Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors

Jike Chong

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-158
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-158.html

December 15, 2010

Acknowledgement

This research was supported in part by an Intel Ph.D. Fellowship, a California Nano-Technology Research Fellowship, funding from the Gigascale Systems Research Center, the Microsoft (Award \#024263) and Intel (Award \#024894) funding and by matching funding by U.C. Discovery (Award \#DIG07-10227).

Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors

by

Jike Chong

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt W. Keutzer, Chair
Professor David A. Patterson
Professor Nelson Morgan
Professor Terrence Hendershott

Fall 2010

Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors

Abstract

Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors

by

Jike Chong

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt W. Keutzer, Chair

Manycore microprocessors are powerful computing engines that are architected to embrace the use of parallelism to extract computational throughput from the continued improvements in the semiconductor manufacturing process. Yet the performance of the software applications running on these microprocessors is highly sensitive to factors such as data layout, data placement, and synchronization. These factors are not usually part of an application domain experts daily concerns, as they look to utilize the powerful compute capabilities of manycore microprocessors for their applications, but failure to carefully address these concerns could mean an order of magnitude of loss in application execution latency and/or throughput. With the proliferation of manycore microprocessors from servers to laptops and portable devices, there is increasing demand for the productive development of computationally efficient business and consumer applications in a wide range of usage scenarios. The sensitivity of execution speed to software architecture and programming techniques can impede the adoption of the manycore microprocessors and slow the momentum of the semiconductor industry.

This thesis discusses how we can empower application domain experts with *pattern-oriented application frameworks*, which can allow them to effectively utilize the capabilities of highly parallel manycore microprocessors and productively develop efficient parallel software applications. Our *pattern-oriented application framework* includes an *application context* for outlining application characteristics, a *software architecture* for describing the application concurrency exploited in the framework, a *reference implementation* as a sample design, and a set of *extension points* for flexible customization.

We studied the process of accelerating applications in the fields of machine learning and computational finance, specifically looking at automatic speech recognition (ASR), financial market value-at-risk estimation (VaR), and financial potential future exposure (PFE). We present a *pattern-oriented application framework* for ASR, as well as efficient *reference implementations* of VaR and PFE. For the ASR framework, we demonstrate its construction and two separate deployments, one of which flexibly extends the ASR framework to enable lip-reading in high-noise recognition environments. The framework enabled a Matlab/Java

programmer to effectively utilize a manycore microprocessor to achieve a 20x speedup in recognition throughput as compared to a sequential CPU implementation.

Our *pattern-oriented application framework* provides an approach for crystallizing and transferring the often-tacit knowledge of effective parallel programming techniques while allowing for flexible adaptation to various application usage scenarios. We believe that the *pattern-oriented application framework* will be an essential tool for the effective utilization of manycore microprocessors for application domain experts.

To Yue, Xuetong, and Peiji,
my wonderful wife and supportive parents.

To the application domain experts,
who are bravely developing applications on the
highly parallel manycore microprocessors.

Contents

List of Figures

Acknowledgments

I would like to thank all the people who have guided and supported me on this journey. In particular, I owe my deepest gratitude to my advisor, Kurt Keutzer, whose constant encouragement and unwavering support fill me with confidence even during the most difficult times. By following his guidance and observing him as a role model, I learned not only the process of conducting research, but also the process of sincerely engaging highly talented people to work together in producing world-class contributions. This thesis also would not have been possible without the inspirational leadership of Professor David Patterson, who spearheaded the establishment of the Parallel Computing Lab at University of California, Berkeley, which provided an environment for long term collaboration between application domain experts and parallel programming experts. I am also grateful for Professor Nelson Morgan and Professor Terrence Hendershott, for their generous guidance and support.

It is an honor for me to have met and studied under professors Andrew Isaacs, Henry Chesbrough, Steve Blank, Jihong Sanderson, and the late Dean Richard Newton, as they inspired me to look beyond theories and algorithms, and to use the opportunities from the Mayfield Fellowship and perspectives from the Haas Business School to observe how tools and concepts get deployed in the real world. I would like to give special thanks to Dr Pradeep Dubey, Dr Yen-Kuang Chen, Dr Mikhail Smelyanskiy, Dr Christopher Hughes at Intel Corporation, and Roger W., Dr Alejandro H., Colin W., and Brian C. in other companies for their guidance and support in shaping this research.

I am indebted to many of my colleagues including Dr Matthew Dixon and Dr Dorothea Kolossa, whose support made possible the numerous real world case studies; Arlo Aria, Dr Nadathar Satish, Dr Youngmin Yi, Ekaterina Gonina, Dr Kisun You, Michael Anderson, Dr Andreas Stolcke, Steffen Zeiler, Dr Gerald Friedland, Dr Adam Janin, and Fares Hedayati for their generous support and close collaborations. They each made critical contributions to the research presented here. I would also like to thank Bryan Catanzaro, Matt Moskewicz, Mark Murphy, Narayanan Sundarum, Bor-Ying Su, Chao-Yue Lai, David Sheffield, Dr Wei Zheng, Dr Kelvin Lwin, Dr Abhjit Devare, Dr Qi Zhu and Dr Douglas Densmore for countless stimulating discussions on research as well as on the meaning of life.

I am grateful to my uncle, Dr Xing Zhu, my father in-law, Dr Xiubao Chang, and my mentors, Dr Li Gong and Dr Prakash Hebalkar for their encouragement for me to pursue a PhD; to Dean Pradeep Khosla and Professor Andrjez Strojwas for their generous recommendations that enabled me to come to Berkeley. This research would not have been possible without the sacrifice from my parents, Xuetong Zheng and Peiji Chong, and my grandparents, Weimei Dong and the late Dechen Zheng, Yuhuan Li, and Yidong Chong, who made every effort to provide me with opportunities to receive the best education possible. Finally, I am forever indebted to my wife, Yue Cathy Chang, who has supported me intellectually and emotionally through a journey of six years during the process of this research.

This research was supported in part by an Intel Ph.D. Fellowship, a California Nano-Technology Research Fellowship, funding from the Gigascale Systems Research Center, the Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

Chapter 1

Introduction

The evolution of computing technology is at an inflection point where microprocessors are forced by underlying physics to use parallelism to take advantage of the increasing scale of silicon integration [17, 81]. Manycore microprocessors have emerged as powerful computing engines that can efficiently extract computational throughput from the continued improvements in the semiconductor manufacturing process. Manycore processors are architected to embrace the use of parallelism by allowing software applications to utilize tens to hundreds of threads per core, and tens of cores per chip concurrently. This paradigm is in stark contrast to traditional threaded processing, where often only a few threads are used in an application. As a result, the inflection point in the hardware platform is causing a disruption in the software development process, such that the traditional sequential hardware abstraction is no longer sufficient for the development of performance-sensitive end-user applications [18].

To further complicate the situation, the performance of the software applications running on manycore microprocessors can be highly sensitive to factors such as data layout, data placement, and synchronization. Failure to carefully address these factors could result in an order of magnitude of loss in application execution latency and/or throughput [26]. At the same time, the types of applications that can take advantage of the manycore microprocessors are expanding. Significant application domain expertise is required to develop new application capabilities for new application usage scenarios[1]. We consider application domain experts as professionals who are trained in specific domains such as machine learning and computational finance. As application domain experts develop new application capabilities on highly parallel manycore processors, they are usually not familiar with how to take care of concerns in data layout, data placement and synchronization, which makes it challenging to develop efficient implementations for new application usage scenarios.

Without an effective methodology to allow applications to quickly leverage the benefits of the increasing scale of silicon integration, the adoption of manycore microprocessors can be severely limited. Such a slow-down in new technology adoption can stunt the growth of

[1]Sections 5.1 and 6.1.3 illustrate that extensive application domain expertise that is required for the development of new application capabilities. The example in Section 6.1.3 involves the implementation of an audio-visual speech recognition application that can perform lip-reading.

the entire semiconductor industry.

This thesis assumes that the productive development of applications for an emerging generation of highly parallel microprocessors is the preeminent programming challenge of our time. We believe that the productive development of applications begins by empowering application domain experts with tools that allow them to utilize the capabilities of the highly parallel manycore microprocessors effectively and to develop efficient parallel software applications productively.

With this perspective in mind, we have developed the *pattern-oriented application framework* to enable application domain experts to achieve application execution efficiency, development productivity, and the portability of development efforts. Our *pattern-oriented application framework* contains four components, which includes an *application context* outlining application characteristics and exposing application concurrency, a *software architecture* describing the application concurrency exploited in the framework, a *reference implementation*, which a fully functional, efficient sample design, and a set of *extension points* for flexible customization.

We demonstrate efficient reference implementations of applications in the fields of machine learning and computational finance, specifically studying automatic speech recognition (ASR), financial market value-at-risk (VaR) estimation, and financial potential future exposure (PFE) estimation.

In the field of machine learning, we analyzed ASR in-depth and optimized it for execution efficiency. ASR allows multimedia content to be transcribed from acoustic signals to word sequences. It is emerging as a critical component in data analytics for a wealth of multimedia data that is being generated on a daily basis. A *pattern-oriented application framework* was developed in order to assist application domain experts to productively utilize highly parallel computing platforms to deploy ASR based applications. For the ASR pattern-oriented application framework, we demonstrate its construction and two separate deployment case studies. Extensive optimizations were applied in the construction of the ASR reference implementation, including application algorithm selection, input speech model structural transformations, hardware platform feature sensitivity analysis, and implementation efficiency tuning. In one of the deployment case studies, the audio-only ASR framework was extended to an audio-visual speech recognition application that takes in video information and use lip-reading to improve recognition accuracy in noisy recognition environments. The ASR pattern-oriented application framework enabled a programmer with only prior experience in Matlab/Java to effectively utilize a manycore microprocessor to achieve a 20x speedup in recognition throughput as compared to a sequential CPU implementation.

In the field of computational finance, VaR and PFE are the two applications being explored in this thesis. VaR is a measure of market risk for a financial portfolio and refers to the maximum loss expected under normal market conditions over a period time at a given confidence level. It is the preferred risk metric recommended in the Basel II international banking standard [5]. For the VaR implementation, we applied optimizations from multiple perspectives, including task-centric function refactoring to reduce necessary computation, numerical-centric module selection to accelerate algorithm convergence, and data-centric module merging to improve memory bandwidth utilization. The optimized implementation

achieved a speed up of 169x on the GPU when compared to a baseline GPU implementation, making it an efficient *reference implementation* for the construction of an application framework. The implementation of this application framework is on-going research.

PFE is a measure of the financial default risk that quantifies the counterparty[2] risk posed by future fluctuations in market prices during the lifetime of the transactions in a bank's portfolio. The exploration of an efficient PFE software architecture is based on the production code base of an industry partner that is a global financial information company. For the PFE application, we investigated multiple application programming interfaces to offload batch of computation from a Central Processing Unit (CPU) to a manycore microprocessor based accelerator. When compared to a reference implementation on the CPU, the optimized implementation of the PFE application that is presented here achieved a speed up of 750x on the GPU, making it an efficient reference design for the construction of an application framework. The implementation of this application framework is also part of the ongoing research.

Our *pattern-oriented application frameworks* are tools that can be widely deployed in the industry, allowing application domain experts to productively develop and deploy software applications for the new generations of highly parallel manycore microprocessors. We analyze an ecosystem for *pattern-oriented application frameworks* in terms of its lead users, developers, building blocks, and the ecosystem components to help the frameworks find initial adoption and allow them to evolve to meet the needs of the industry.

1.1 Thesis Contributions

The contributions this research offers include:

1. Proposing four components that when used together can allow a *pattern-oriented application framework* to address the efficiency concerns of the application domain experts and help them productively develop software applications for the highly parallel manycore microprocessors[3]

2. Demonstrating that both application domain expertise and parallel programming expertise are required to develop high-performance *pattern-oriented application framework*

3. Optimizing implementations of applications in machine learning for automatic speech recognition and in computational finance for market value-at-risk estimation and financial potential future exposure estimation, achieving orders of magnitude speed up in execution time compared to sequential execution

[2]Counterparties are brokers, investment banks, and other securities dealers that serve as the contracting party when completing an over-the-counter financial security transaction. The details are explained in Section 5.2.2.

[3]The *pattern-oriented application framework* concept proposed here is inspired by the framework concept discussed in [18].

4. Demonstrating a *pattern-oriented application framework* for automatic speech recognition with deployments in multiple usage scenarios, enabling application domain experts to achieve 20x speedup on the highly parallel manycore microprocessors

5. Proposing an ecosystem in which *pattern-oriented application frameworks* can find adoption in industry and evolve to meet the needs of the application domain experts

1.2 Thesis Outline

The chapters in this thesis are presented as follows:

- Chapter 2 provides the background and motivation that highlights the implementation gap in parallel application development.
- Chapter 3 surveys the existing tools and environments for the productive development of parallel applications and introduces the concept of *pattern-oriented application frameworks* for domain experts.
- Chapter 4 illustrates the proposed *pattern-oriented application frameworks* for domain experts to more effectively program manycore microprocessors.
- Chapter 5 presents the construction process for the *pattern-oriented application frameworks*.
- Chapter 6 demonstrates how *pattern-oriented application frameworks* can be deployed in the field.
- Chapter 7 proposes an ecosystem in which *pattern-oriented application frameworks* can find adoption and evolve to meet the needs of the application domain experts.
- Chapter 8 provides a summary of key lessons learned in the process of developing *pattern-oriented application frameworks* for application domain experts.

The following chapter explains the industry trends in both hardware and application software that are creating an *implementation gap* for the development of software applications. It then goes on to propose a solution process that will be elaborated upon in the remainder of this thesis.

Chapter 2

Background and Motivation

The evolution of computing technology was recently at an inflection point where the industry is transitioned from sequential computing platforms to parallel computing platforms. This inflection point in the hardware platform is causing a disruption in the software development process. The traditional sequential hardware abstraction is no longer sufficient for the development of performance-sensitive end-user applications. This chapter explains the industry trends, with respect to both hardware and application software, that are creating an *implementation gap* for software application development. It then proposes a solution process that will be elaborated upon in details in the remainder of this thesis.

2.1 Computing Technology Trends

For the past four decades, the computer industry has been driven by Moore's Law, which predicted that the density of integrated circuits can double approximately every two years. Moore's Law has become a synchronizing force for all levels of the semiconductor industry. This is most clearly seen in the microprocessor industry, where Moore's Law has synchronized low-level research and development (R&D) efforts, such as the chemistry necessary for chip manufacturing steps, up to the high level R&D of the practically achievable end-user applications performance on the microprocessors. Figure 2.1 illustrates the effect of Moore's Law on microprocessor designs over the past four decades [144]. The data points are based on microprocessor specifications that are plotted according to their release dates. Starting at the top line, which shows the number of transistors that are integrated on-a-chip, it is clear that the industry is on track to increase the scale of integration approximately every two years. Since the early 1970s, the exponential growth during these three decades has enabled more than one billion transistors to be integrated on one microprocessor.

At the application development level, software developers have depended on the assumption that exponential growth in transistor density will result in similar exponential improvement in the execution performance of a single stream of application code executing on one processor. In the last decade, however, the scaling process has reached physical limits in microprocessor clock speed, power consumption, and performance per clock. We see in Figure 2.1 that many of these metrics have plateaued. Asanovic *et al.* in [17] has described

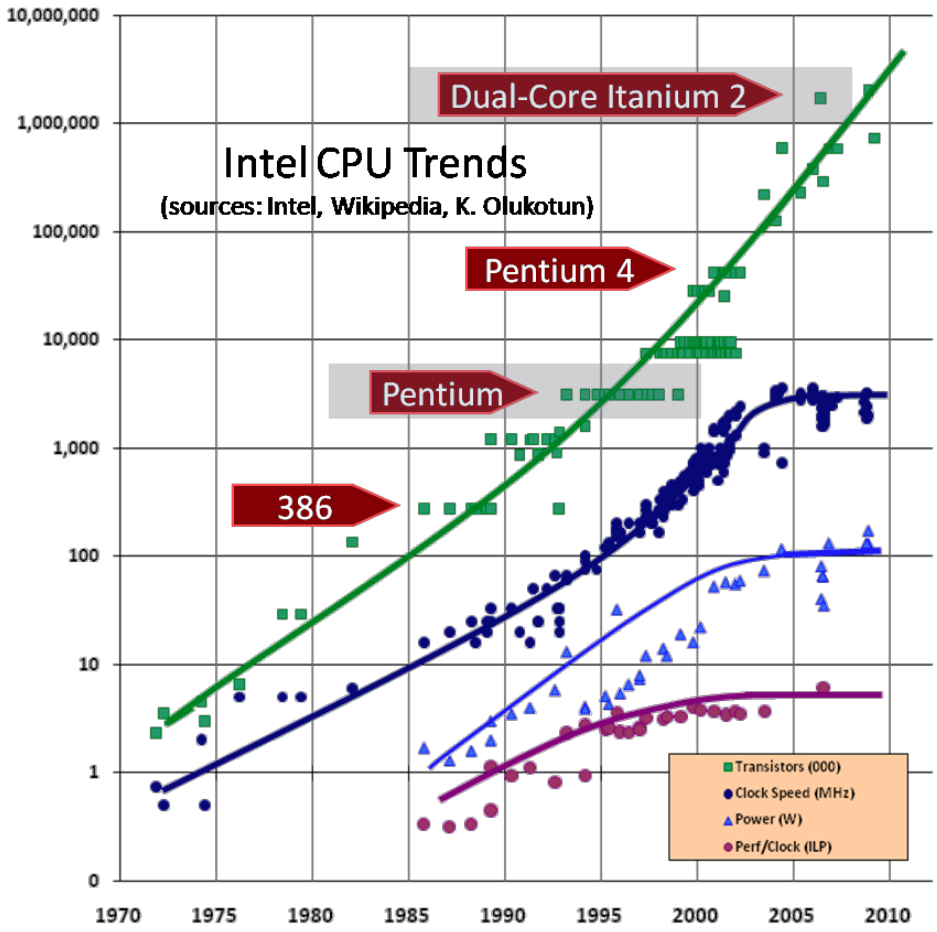


Figure 2.1: Microprocessor specification trends over the past four decades (Prepared by Herb Sutter in [144])

this effect as the aggregation of three performance scaling challenges:

1. "Power Wall": Although power is a scarce resource for computation, transistors are "free" That is, we can put more transistors on a chip than we have the power to turn on.

2. "Memory Wall": Load and store instructions are slow, but multiply is fast [155]. Modern microprocessors can take 200 clocks to access Dynamic Random Access Memory (DRAM), but even floating-point multiplies may take only four clock cycles. Many applications are becoming bandwidth-limited.

3. "ILP Wall" : There are diminishing returns with respect to finding more instruction-level parallelism (ILP) via compilers and architecture innovations including branch prediction, out-of-order execution, speculation, and Very Long Instruction Word systems [81].

The aggregate effect of these three "Walls" is that sequential processing performance is becoming increasingly difficult to improve. While the total number of transistors that one can integrate on a chip continues to increase, we are forced to respect the limitations of physics and to organize microprocessor designs around the physical limitations of power dissipation, memory device proximity, and the limited scope of implicit instruction level parallelism in software implementation.

At the same time, a new breed of "manycore" microprocessor architectures has emerged. Such architecture sacrifices the performance of any single stream of instructions and use many simpler and more power-efficient processor cores in parallel in order to achieve higher overall throughput under any specific power budgets. The more power-efficient core architectures allow manycore processors to mitigate the "Power Wall". The manycore architecture concurrently maintains the context of numerous threads and allows low overhead context switches to occur between threads. This effectively hides long memory latencies by allowing stalled processor pipelines to switch to other ready threads so as to be able to continue, thus mitigating the effects of the "Memory Wall". Lastly, special data-parallel languages are proposed and used to expose and represent more parallelism in applications, thereby mitigate the "ILP Wall".

Having to adapt new applications to a new data-parallel language is not a preferred move in the industry. As summarized in [17], this move has been forced upon us by our desire to continue the scaling of microprocessor performance while respecting the laws of physics:

> This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.
> – Berkeley View, December 2006

Many major microprocessor vendors have general-purpose manycore processors either in production or on the roadmap. For example, as of 2010, NVIDIA has brought to market the G80/GTX200/GTX400 general-purpose manycore processor architectures. AMD/ATI offers the Radeon 4000/5000/6000 series manycore programmable processor architectures. And Intel is developing its many-integrated-core (MIC) processor architecture under the name Aubrey Isle [131], previously known as Larrabee [134].

Manycore Processor Architecture

Manycore processors are expected to be an increasingly important component in computing technologies. Current and emerging many-core platforms such as the GPUs from NVIDIA and AMD/ATI, as well as the Intel MIC processor are built around an array of processors each running many threads of execution in parallel. As shown in Figure 2.2, each core employs variations of the Single Instruction Multiple Data (SIMD) architecture, where the same instruction can operate on multiple pieces of data at the same time. Amortizing the instruction-processing overhead among many data calculations is an effectively way to reduce power consumption [98]. The cores are then connected together using levels of

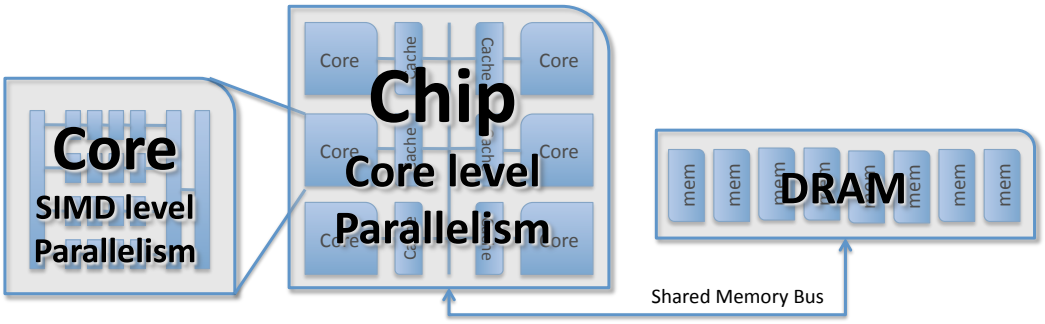


Figure 2.2: The generic manycore architecture.

shared memory hierarchy, allowing synchronization to occur between the cores on a chip. The cores also share the same DRAM memory controller, thus enabling a shared memory space abstraction for ease of application development.

In this thesis, the highly parallel manycore microprocessors we use are the NVIDIA graphics processing units (GPU), which are programmed using the Compute Unified Device Architecture (CUDA) [122]. NVIDIA GPUs were chosen because they are representative of an emerging generation of highly parallel microprocessors that have as many as 15-30 of cores and 8-16 SIMD lanes in each core. At the same time, CUDA has a mature software development environment for the shared memory manycore on-a-chip architecture, which allows large user applications to be effectively developed. Although the tools demonstrated in this thesis is based in CUDA, the concepts presented are not limited to CUDA and are applicable to other development environments.

2.2 Software Application Trends

A software application is a solution that solves a problem when it is implemented in software and executed on a computer. The types of applications that could benefit from highly parallel manycore platforms are expanding. In 2005, Pradeep Dubey put forth a vision that by 2015, computing will be increasingly applied to a broad range of applications involving Recognition, Mining, and Synthesis [64]. These applications are expected to demand and gain more utility from the increasing processing capabilities offered by manycore parallel architectures.

At the same time, the application software that is running on top of the highly parallel computation platforms is becoming increasingly complex. Many large projects often involve tens to hundreds of software developers at a time, making the design of succinct software architectures crucial for the success of the projects. With the growing maturity of open source software available, many software projects are designed as a composition of existing optimized software libraries and infrastructures in order to minimize the effort necessary to construct a new application from scratch.

The expanding variety of applications that can take advantage of parallel microproces-

sors and their increasing complexity are the two important points to keep in mind as we develop technologies to assist software developers to be able to better utilize highly parallel microprocessors.

2.3 Current Parallel Software Development Best Practices

The development of a complex software application involves the coordinated efforts of multiple groups of people with diverse areas of expertise. To successfully construct the end application, they must have a coherent high-level view of a system's structure and organization. Through the process of developing efficient application implementations in diverse fields such as machine learning and computational finance, we found that a three-step process worked well in the construction of efficient parallel application. Figure 2.3 illustrates this three-step process as: *specify*, *architect*, and *implement*.

1. In the *specify* step, the application characteristics are described in terms of the type, size, and requirements of the computation required, as well as performance goals that must be met or would be nice to meet. The parallelization opportunities in the application are also exposed, as well as the amount of parallelism that each opportunity entails.

2. In the *architect* step, the design space to be explored is defined. The design space is the set of alternative implementations of the solution that solves the end-user's problem. It is associated with the parallel opportunities that are exposed in the *specify* step. In this step, the potential performance bottlenecks are also explored and prototyped. The end result is a set of data types and application programming interfaces (APIs).

3. In the *implement* step, the functions of the application are implemented by translating the high level descriptions of the application into a software code base, and unit tests are defined and deployed in order to verify functional correctness and to evaluate performance requirements.

The purpose of specifying this three-step process is to partition the design process such that one set of activities should be completed before another set of activities begins. The parallelism opportunities in the *specify* step should be explicitly enumerated before one embarks on the exploration of the design space in the *architect* step. Failure to do so could result in a partially defined design space where better performing implementation alternatives may not be duly explored. All potential bottlenecks in the *architect* step should be analyzed before one starts the detailed implementation in the Implement step. Failure to do so could result in late-stage performance or integration problems that prevent the on-time deployment of a software project from occurring.

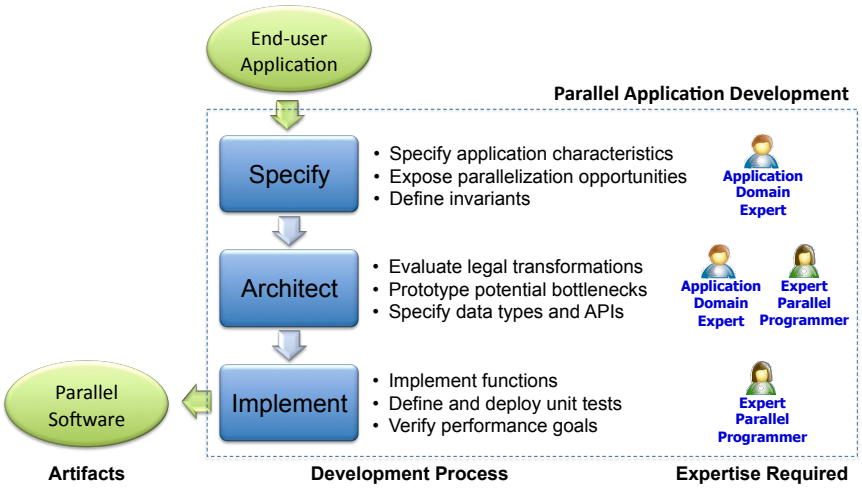


Figure 2.3: The three-step process in a parallel applications development flow.

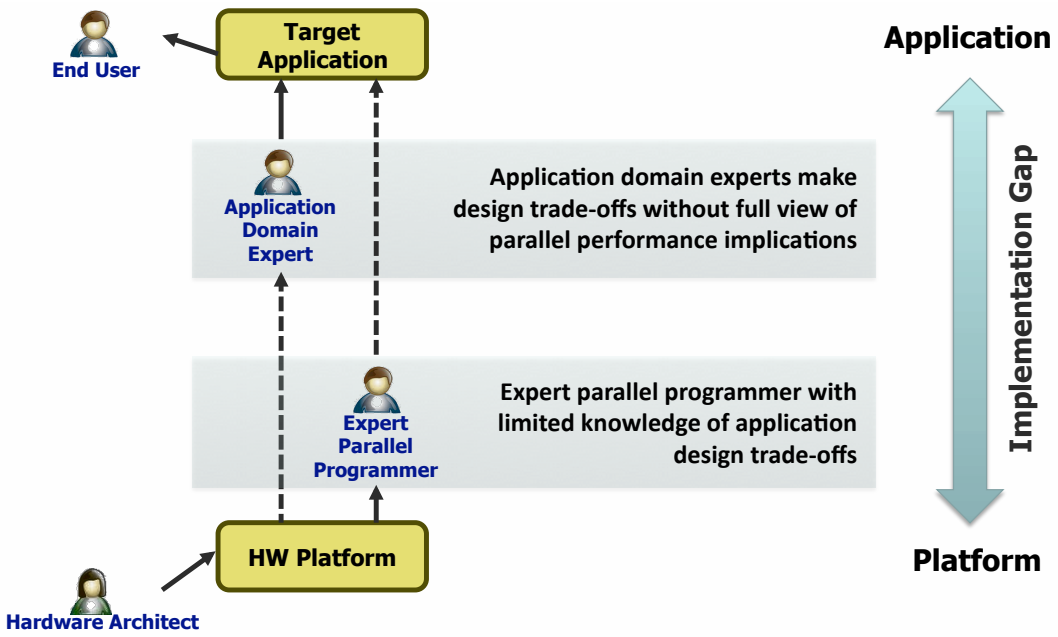


Figure 2.4: The parallel application *Implementation Gap*.

2.4 The Implementation Gap

One can observe two groups of programmers for application development: the parallel programming experts and the application domain experts. The parallel programming experts are often the staff programmers and IT professionals in an organization. They have a deep understanding of the parallel computing platforms. Application domain experts are often the researchers and practitioners in diverse fields like automatic speech recognition, computer vision, and financial risk analytics. They understand application usage characteristics, algorithm variations, and legal transformations of data and control well enough to be able to derive various implementation alternatives to improve application performance.

To achieve an efficient implementation of an application on a highly parallel platform, applications must be designed taking into account the characteristics of the specific application, as well as the capabilities of the underlying computing platform.

Although application domain experts have an in-depth understanding of the application characteristics and requirements, they are often ill equipped to deal with application parallelization challenges because they lack the computing platform insights to anticipate and avoid execution efficiency bottlenecks in various implementation alternatives. For example, Section 5.1.5 demonstrates the nuances in the multicore and manycore computing platforms that a speech recognition domain expert must be aware of to achieve good performance. Specifically, being able to utilize hardware-supported atomic operations can provide a 3x performance gain compared to relying on software-based synchronization mechanisms on the GPU.

Although parallel programming experts have intimate knowledge of the implementation platforms, they often lack the application level perspective to implement application level transformations that can result in significantly more efficient implementations for the application at hand. For example, Section 5.1.4 demonstrates that performing speech model transformations based on application domain expertise can provide up to 2x performance improvements on recognition network traversal speed, which translates to up to 40% improvement in application-level recognition latency.

An application development team that is looking to adopt parallel processing technology must have a collaboration between both application domain experts and parallel programming experts so as to reduce this *implementation gap* (Figure 2.4) in deploying parallel applications.

The implementation gap represents a significant barrier to the successfully construction of efficient implementations. There are very few developers who have expertise in both the application domain and the implementation platform to lead these projects. In order to make parallel programming widely applicable, it is necessary to have a set of application development tools and the infrastructure required to bridge the implementation gap and meet the needs of the end-users of software applications who are demanding an expanding variety of ever more complex applications.

Referring back to Figure 2.3, in the current best practices for parallel software development, parallel programming experts are involved in the Architect and Implement step of every application development project for each usage scenario. In the automatic speech

recognition related application, for example, parallel programming experts must be involved in the development of usage scenarios of an in-car speech recognition system as well as a meeting transcription speech recognition system. While there are many application domains that can benefit from parallel implementations, the different application usage scenarios in each application domain are even more numerous. If parallel programming experts have to be involved in the development of every parallel application, the deployment of highly parallel microprocessors will be severely limited.

2.5 Summary

The "Power Wall", "Memory Wall" and "ILP Wall" are forcing microprocessor architectures to go parallel. The observation here is that the future of computer architecture is heading in the direction of manycore microprocessors, with tens of cores on a chip, and tens of SIMD lanes concurrently executing on each core. This causes a disruption in the software development process, where the traditional sequential hardware abstraction is no longer sufficient for the development of performance-sensitive end-user applications.

Meanwhile, there is an expansion in the variety of recognition, mining, and synthesis based end-user applications that can take advantage of the emerging manycore microprocessor architectures. However, the software is becoming increasingly complex to build.

To effectively implement the great variety of complex software applications with the highly parallel manycore microprocessors, the current best practice is to follow the *specify*, *architect*, and *implement* three-step process. Yet, going through this process requires both application domain expertise and parallel programming expertise, as having either one or the other alone is not sufficient if one wants to develop efficient parallel software applications. This problematic situation is the parallel application *Implementation Gap*.

Unfortunately, building a team that has both areas of expertise so as to bridge the gap for the development of every end-user application will be cost-prohibitive. At the same time, failure to enable the deployment of efficient end-user applications and demonstrate the value of highly parallel microprocessors will severely stunt the growth of the entire semiconductor industry.

To this end, the establishment of an alternative approach to productively develop efficient and highly parallel software applications is essential. Thus, the next chapter presents a review of the existing work in this area that have attempted to solve this problem, and the following chapters demonstrate a new effective solution toward the closing of the *implementation gap*.

Chapter 3

Tools for Closing the Implementation Gap

A tremendous amount of work has been done in software engineering in order to allow larger and more complex problems to be solved using software. This chapter focuses on the techniques that can help resolve today's challenge in closing the implementation gap and enabling "effective" implementation of software on highly parallel microprocessors. Firstly, the metrics for "effectiveness" are defined. Then, three sets of tools are surveyed so as to evaluate their effectiveness in closing the implementation gap. The three sets of tools are: software architecture narratives, software implementation support, and parallel software implementation tools. Finally, a summary of the key lessons learned from prior work is given.

3.1 Metrics for Evaluating Effectiveness

The metrics for the effectiveness of the tools for closing the *Implementation gap* are presented here from the perspective of an application domain expert who is looking to productively implement efficient applications on highly parallel microprocessors.

Application domain experts are researchers and practitioners in diverse fields such as automatic speech recognition, computer vision, and financial risk analytics. While they have a deep body of knowledge in the specific application domain, they may not have a clear sense of the relative efficiency of application architecture alternatives with respect to highly parallel software implementations. In fact, there is often some notion of implementation optimizations that was successful in sequential processing, but may no longer be effective in parallel processing[1].

A tool to effectively assist application domain experts to more productively implement

[1]One example of this is the technique of memoization, used to eliminate redundancies in computing identical duplicate sub-problems in algorithms like dynamic programming and branch-and-bound. Here the program maintains a hash table to keep track of the results that have been computed. In sequential processing, this eliminates the need to do redundant work. In parallel processing, parallel threads may start working on the same result at the same time, and still duplicate the work.

efficient applications on highly parallel microprocessors must provide: *efficiency*, *productivity*, and *portability*.

To an application domain expert, *efficiency* measures the ability of the application developed to obtain high performance using the available resources in a computing platform. It can be demonstrated in two ways:

1. Producing higher performance in an application-specific metric, such as higher accuracy for speech recognition under some time constraints

2. Obtaining high degree of utilization of the underlying hardware platform

To an application domain expert, *Productivity* means a fast path to a functionally correct solution that implements a feature or an application while meeting some performance constraint. One can categorize the metrics into the following classes:

1. The background required, which includes additional background knowledge such as application characteristics that are more amenable for parallelization, and efficient software architectures for parallel implementations.

2. Lines of code (LOC), which is used to measures the LOC that must be comprehended before one can utilize existing libraries or frameworks, as well as the LOC required to construct new features.

3. Potential for bugs, which includes errors due to misunderstanding the concept, and errors from the inevitable process inherent to implementing the many lines of code.

To an application domain expert, *Portability* means how easy it is to get their code to execute on another hardware platform. There are two aspects of portability:

1. *Functional portability*: Can the code execute correctly on an alternative platform?

2. *Performance portability*: Can the code leverage the available resources on the alternative platform? This is also called "Parallel scalability" of an implementation [157].

With these metrics, the tools for *software architecture narrative*, *software implementation support*, and *parallel software implementation infrastructures* are analyzed.

3.2 Software Architecture Narrative

When software developers have a discussion about the architecture of a piece of software, they often use a set of vocabulary to assist them in describing the structure and organization of that piece of software. There are many different types of narratives that can be used, including: idioms and patterns. While idioms are language-specific, patterns are language-agnostic. These software architecture narratives are described in details in this section.

3.2.1 Idioms

Idioms are small, language-specific coding techniques. They allow features that are not native to a programming language to be expressed. Examples of these features include the memory management model, input and output (I/O), and object initialization. The usage of language extensions beyond the core language features becomes *idiomatic*.

This concept comes from idioms in natural languages, where idioms are expressions whose meaning is unpredictable from the usual meanings of its constituent elements. An example of this is *"to kick the bucket"* which in English is often used to mean *"to die"*. As early as in 1989, Coplien used the concept of idioms in programming languages to teach C++ at AT&T Bell labs. To Coplien, idioms are fundamental building block of reuse in a specific (programming) language. He writes in his 1992 book [50] (page vi):

> In programming, as in natural language, important idioms underlie the suitability and expressiveness of linguistic constructs even in everyday situations. Good idioms make the application programmer's job easier, just as idioms in any language enrich communication. Programming idioms are reusable "expressions" of programming semantics... The idioms usually involve some intricacy and complexity, details that can be written once and stashed away. Once established, programming language idioms can be used with convenience and power.

A simple example of this is the technique of overloading of the << operators in C++ in order to load data into an abstract data type. The operations invoked by << operator is not defined in C++ natively. The resulting code generated according to the idiom would be constructed by the software developer, and would only be relevant to the users of the specific abstract data type.

A more complex example here is the technique of "reference counting," which is used to track the number of references to an object in an object oriented language. The technique can be used to assist in automated garbage-collection in order to free up unused memory spaces during the execution of a program for languages that do not support garbage-collection natively. They exist in the context of a language property such as object-oriented language, and are for particular purposes, such as automated garbage-collection. A detailed example of this can be found in [50] (page 58).

Idioms are productivity tools that can help software developers in establishing the necessary background knowledge to effectively realize building blocks in the software architecture of an application. While the software developer still has to design the structure of the software, write all of the lines of the source code, and test the software for functional correctness and performance characteristics, the potential for programmer errors is greatly reduced.

Although the proper usage of idioms can lead to more efficient code, this is not guaranteed. More productive implementations may also be less efficient to execute, e.g. applications that are implemented with automated garbage-collection incur additional overhead during execution for counting references to objects, and applications with explicitly-managed memory space can execute faster because they do not have to pay for the reference-counting overhead.

3.2.2 Patterns and Pattern Languages

A *design pattern* is a generalizable solution to a class of recurring problems that occurs in the design of software. A design pattern attaches a name to a well-analyzed solution that encapsulates the way an expert in the field solves a problem. Compared to idioms, design patterns are not language specific, and hence are more general than idioms[2].

While design patterns describe point solutions to problems encountered in a design space, a *pattern language* is an organized way of navigating through a collection of design patterns. This helps a software developer solve a set of inter-related challenges in the process of developing a software application.

Patterns, as defined in the dictionary, refer to forms or models proposed for imitation. The concept of patterns in design has a long history that goes back to the early 1960s. In his seminal book on systems engineering [80], Hall relates patterns to the recurring characteristics of system processes that can be expressed in such a way as to facilitate the ability of non-experts to learn how to improve system engineering. He writes:

> Of all the possible ways of defining the systems engineering function, the most significant and explicit is an operational one, which gives a description of the general pattern of work from formulation of a program of projects to completion of a specific project. For it is the pattern, more than anything else, which gives the function its essential structure and characteristics.

Christopher Alexander, a civil architect and Professor Emeritus of Architecture at the University of California, Berkeley, used the concept of patterns in the design of civil architecture [10]. For Alexander, patterns are knowledge written in a style that helps to lead a designer from a recurring architectural design problem to a set of solutions for the design of buildings and towns. He writes:

> A pattern is a careful description of a perennial solution to a recurring problem within a building context, describing one of the configurations which brings life to a building... Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Alexander organized his 253 patterns of civil architecture in a linear sequence (1) to (253) according to the scope the design patterns affect, from the largest scope in *Independent Regions* (1), and *The Distribution of Towns* (2), to *House for a Couple* (77), to *Bathing Room* (144), to the smallest scope in *Things from Your Life* (253). At the same time, a design pattern in this language relates to and helps complete selected larger patterns that comes before it, and relates to and is completed by selected smaller patterns that come after

[2]While the concept described by a common idiom such as integer increment can be implemented in multiple languages: $(InPascal : Inc(x))$, $(InC++ : x+ = 1; or x++;)$ they are often considered to be different idioms, as the implementation realization may be different.

it. Alexander envisions that a design process would involve the selection of a small subset of the patterns and the generation of a design around the solutions suggested in the patterns.

Inspired by these concepts, design patterns were applied to software development by Kent Beck at Apple Computer, and Ward Cunningham at Tektronix in 1978 and publicly discussed at a panel discussion at the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) Conference in 1978 [25].

Design patterns in software engineering were popularized in 1994 by Gamma, Helm, Johnson and Vlissides in the book "Design Patterns: Elements of Reusable Object-Oriented Software" [68]. This book focuses on design patterns for object-oriented programming (OOP), and includes 23 patterns that are organized first by their *purpose*, which reflects what a pattern does, and then by their *scope*, which specifies whether the pattern applies primarily to classes or objects. The different purposes include: creational, structural, and behavioral. Creational patterns help guide a user in techniques of object creation. Structural patterns help guide a user in the composition of classes or objects. Behavioral patterns help guide a user in the definition of class and object interactions. The different scopes here include Class and Object, which distinguish between whether the patterns deal with relationships between Classes that are static and fixed at compile time, or between Objects that can be dynamically changed at run-time.

At about the same time, Mary Shaw and David Garlan studied leading software industry practices and crystallized a set of architectural styles in their 1996 book "Software Architecture: Perspectives on an Emerging Discipline". For Garlan and Shaw, they defined an architectural style as "a family of systems in terms of a pattern of structural organization" [135].

Frank Buschmann *et al* [36] used concepts from Gamma *et al* and Shaw and Garlan to distinguish between two levels of patterns: 1) architectural patterns that describe the overall structuring principle of viable software architectures; and 2) design patterns that only influence the architecture of a subsystem.

Timothy Mattson *et al*, in their 2004 book "Patterns for Parallel Programming" [110], focused on patterns used for parallel computing. They introduced 19 patterns and organized them into a hierarchy of patterns for addressing three different levels of design space, including *finding concurrency* design space, *algorithmic structure* design space, *supporting structure* design space.

The parallel programming design pattern used in this thesis is based on the parallel design pattern research hosted at the University of California, Berkeley [94]. The working title for this set of patterns is OPL or "Our Pattern Language". A pattern language is set of inter-related design patterns that are organized to provide guidance to a software developer for the process of solving a design problem.

Historically, patterns and pattern languages have been developed to serve the following three purposes:

1. Education: Patterns document effective solutions to recurring problems in order to help to more quickly transfer tacit expert knowledge in parallel computing to new parallel programmers.

2. Communication: Patterns provide a set of common vocabulary, and allow experts in parallel software design to use this vocabulary to further advance the field.

3. Design: Patterns and the pattern languages inform the design of frameworks that support software developers to quickly construct software applications.

In order to help to illuminate problems and provide possible solutions, the OPL is structured as five areas of concerns for a software developer, helping to illuminate problems and provide possible solutions (Figure 3.1):

1. Area (a) contains the *Structural Patterns* and addresses concerns related to the high level structure of the application being developed. The patterns in this area are drawn from the architectural styles by Shaw *et al.* [135].

2. Area (b) contains the *Computational Patterns*, which addresses concerns related to the identification of key computations; they are drawn largely from the thirteen motifs, which are the algorithm methods that capture patterns of computation and communication originally described as "dwarfs" in [17].

3. Area (c) contains the *Parallel Algorithm Strategy Patterns* and addresses the concerns related to high-level strategies that describe how to exploit concurrency in a parallel computation. These patterns are drawn from the patterns in the "Algorithm Structure" design space in [110].

4. Area (d) contains the *Implementation Strategy Patterns* and addresses concerns related to the realization of the source code to support (i) how the program itself is organized and (ii) the common data structures specific to parallel programming. These patterns are a super-set of the patterns in the "Supporting Structures" design space in [110].

5. Area (e) contains the *Concurrent Execution Patterns* and addresses concerns related to the support of the execution of a parallel algorithm. This includes (i) strategies that advance a program counter and (ii) basic building blocks to support the coordination of concurrent tasks.

In the context of this thesis, we focus on two areas of concerns of the software developer, which are described by the Structural Patterns and Computational Patterns. Figure 3.1 shows these two areas of concerns at the top, where they are placed side by side with arrows that illustrate the tight coupling between them. These arrows demonstrate the iterative nature of the way that a software developer uses the Structural Patterns and Computational Patterns.

Software architecture is thus the hierarchical composition of these Structural and Computational Patterns, which can be subsequently refined using the design patterns in the other areas of concerns.

Each pattern in the OPL has a *name* that alludes to the problem or the solution of the pattern that can invoke familiarity for the reader. Each pattern also involves a narrative that illustrates the *problem* that it solves, the *context* in which it is relevant, the *forces* that

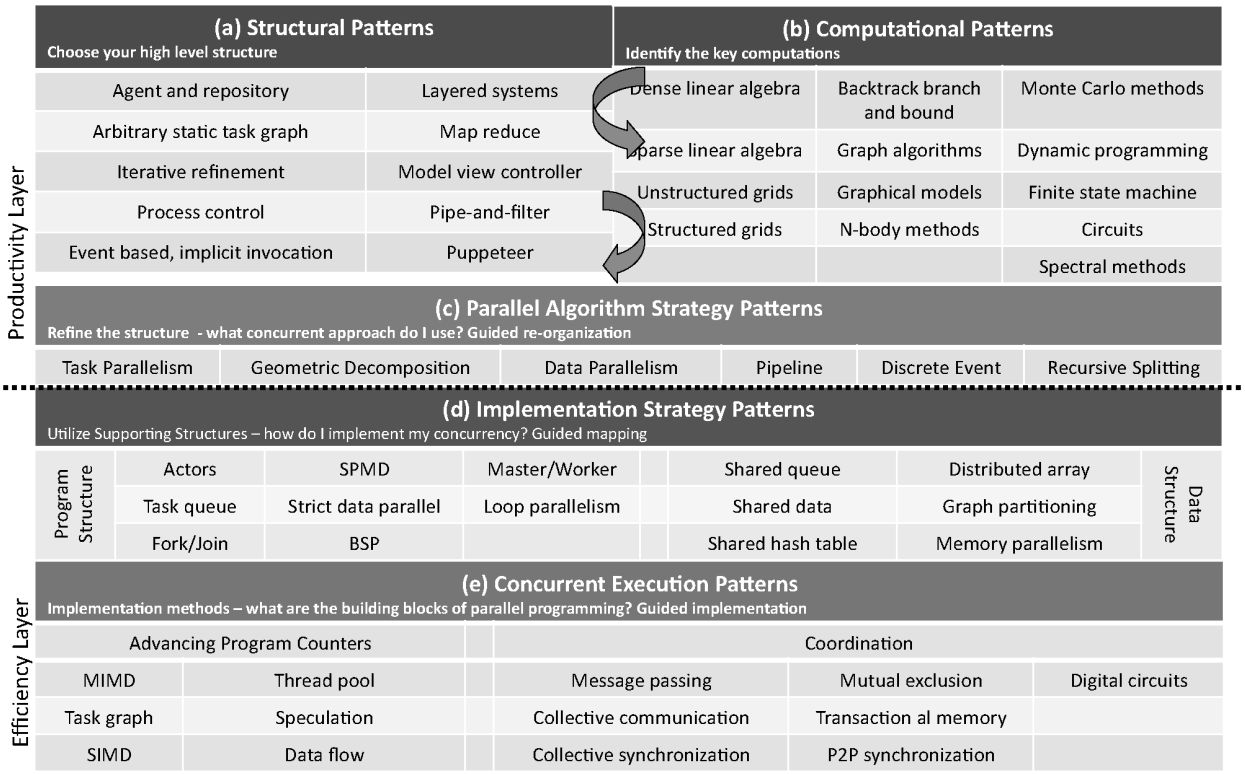


Figure 3.1: Our Pattern Language.

act on the solution, and the *solution* to the problem. It also includes pedagogical *examples*, *known uses*, as well as a list of *related patterns*. An example is provided in Appendix A.

Parallel Programming Design Patterns are productivity tools that can help software developers to establish the necessary background knowledge to effectively organize the software architecture of an application, as well as its mapping to hardware. Although the software developer still has to implement the structure of the software, write all of the lines of the source code, and test the software for functional correctness and performance characteristics, the potential for programmer errors is now greatly reduced.

Patterns are not efficiency tools, as they do not directly affect the efficiency of the end application. They can, however, help illuminate potential performance bottlenecks and help software developers avoid potentially inefficient architectures or implementations.

3.3 Software Implementation Support

In the era of manycore platforms, software programming patterns alone do not provide sufficient assistance for the majority of domain experts to program manycore platforms. While software programming patterns describe the solutions, domain experts are still required to arduously implement the application in their domains. In contrast, libraries, skele-

tons, and application frameworks provide assistance in implementing a solution. Hence, this section examines each of these and how they can help bridge the implementation gap.

3.3.1 Libraries

A software library is a collection of sub-routines or classes used to develop software. They can be constructed for specific application domains, such as the OpenCV (Open Source Computer Vision) library for computer vision [150], or for specific platforms, such as the CUDPP (CUDA Performance Primitives) library for the NVIDIA CUDA-enabled graphics processing units (GPUs) [52], or for specific classes of algorithms, such as the BLAS (Basic Linear Algebra Subroutines) library for manipulating matrices [29].

A library enables software reuse: by implementing standard functions such as "sorting" once, they can be reused again and again in many usage scenarios. Given the amount of reuse, significant effort can be put into making the standard functions as efficient as possible. A library subroutine is called through an interface, which is a list of the input and output operands necessary for a library to perform its functions. To effectively interact with a software library, one is only required to understand the library's intended function and its interface. This allows significant complexity to be packaged into the libraries that provides efficient execution of the library functions, and at the same time it allows the library user to be highly productive.

An application can gain significant computational efficiency by using well-tuned libraries such as the BLAS (Basic Linear Algebra Subroutines) library. Utilization levels of 60-97% of the peak achievable computational throughput have been reported for the BLAS library [148]. Such high levels of utilization are challenging to achieve on today's highly complex computing platform, as an application's performance may be limited by a number of bottlenecks such as the memory bandwidth bottlenecks, workload imbalance, and multi-thread synchronization bottlenecks. Utilizing a library routine to implement an application's compute-intensive sections is a fast way to improve the efficiency of the application.

Application developers can gain significant productivity by utilizing libraries from three main perspectives: 1) They can achieve the intended functions without understanding all of the details of library implementation, and this saves time in regard to the obtaining of background knowledge to implement the require functions; 2) the library user is only required to use the library interface to instantiate the library functions, and this requires much fewer lines of code to be written, reducing opportunities for bugs; 3) the library subroutines' function has been extensively tested and verified by other users, increasing the confidence for the correctness of its operations, allowing faster localization of bugs in modules that are not library routines during debugging.

In terms of the portability of an application based on library routines, the availability of libraries with identical interfaces on multiple platforms allows applications using these interfaces to be functionally portable across platforms.

While libraries play an important part in providing implementation support for application developers, when faced with the implementation of an application, there is still the decision of how to architect the application. Different applications architectures would call

for different subset of library routines. For example, an application that utilizes a breadth-first search on a graph could make use of a graph processing library, such as the BGL (Boost Graph Library) [136], or the adjacency matrix of the nodes in the graph could be represented using sparse matrix and use the OSKI library (Optimized Sparse Kernel Interface) [149]; or the adjacency matrix could be represented in a dense matrix format to make use of the BLAS library.

Decisions about how to transform an application or algorithm in order to make use of the specific optimized library can be handled with higher-level implementation support infrastructures like skeletons and frameworks, which we will examine in the next section.

3.3.2 Skeletons

A software skeleton resembles the outline of an algorithm. It was introduced in some early works by Murray Cole on parallel-programming implementation-support at University of Glasgow, UK. He first introduced the concept of "algorithmic skeletons" in 1988 [48, 47], where "each such skeleton captures the essential computational structures of some familiar class of algorithms". The idea resonated within the software engineering community in Europe and several research groups started working in this direction.

Among the groups were Danelutto *et al.*, from the University of Pisa, Italy, Darlington et al, from Imperial College, London, UK, and Kuchen *et al.* from Aachen University of Technology, Germany. Darlington *et al.* developed functional language embeddings of the skeletons in 1993 [58], and extended it to Fortran in 1995 [57]. In 1995, Danelutto et al designed P^3L, Pisa Parallel Programming Language, which is a skeleton in parallel C [56], and then latter extended it include dynamic run time support in 1999 [55]. Kuchen *et al.* first presented Skil, an imperative language enhanced with higher order functions and a polymorphic type system in 1996 [31] and in 2002 produced a C++ Skeleton Library [99] with two classes of Data Parallel Skeletons: computation skeletons, such as the Map function, and communication skeletons, such as Permute functions, running on distributed data structures. They also produced a class of Task Parallel Skeleton that offers the Farm function for splitting and joining task as well as parallel composition.

Such works adopted the algorithmic skeleton concept by Cole in [48, 47], where:

> The user must describe a solution to a problem as an instance of the appropriate skeleton [a skeleton] is selected and fleshed out with appropriate procedure and type definitions required to customize it to a particular problem.

Software skeletons provide the outline of an algorithm, where a piece of code has to be inserted for compilation. They provide a partially functional system with high-level structures already designed and implemented. A typical example here is the elementary algorithm "MapReduce", where a parallel programs can be constructed by inserting a side-effects-free, data-parallel "Map" function and an associative "Reduce" function.

The main advantage in using skeletons lies in their formal framework for algorithm composition: a skeleton requires the "flesh" code to satisfy strict invariance in order to be fitted into the skeleton; in exchange, the skeleton can perform a complex set of transformations

at compile time and run time to optimize the execution performance on a target computing platform [55].

However, when an application developer is faced with the implementation of an application, and multiple algorithms are required for different phases of execution, there is little guidance on how to compose different skeletons so as to obtain optimal efficiency at the application level.

3.3.3 Frameworks

The term "framework" is often considered overloaded. Ralph Johnson and Brian Foote [89] discussed the concept in as early as 1988. They viewed frameworks used in object-oriented programming as:

> ...a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes.

As classified in [65], there are generally three types of software frameworks for software development:

1. System infrastructure frameworks, which simplify the development of efficient system infrastructure in operating systems [37] or communication frameworks [132]
2. Middleware integration frameworks, which are used to integrate applications and components
3. Application frameworks, which directly address specific application domains.

In this thesis, we focus the discussion on application frameworks. To us, application frameworks are developed as a tool for propagating proven software designs and implementations in order to reduce cost and improve the quality of the software. This concept has been discussed since the 1980s [137] and early application frameworks were built with macro processors and conditional compilation techniques [23].

In the 1990s, as object-oriented programming enabled increasingly complex software to be developed, software reuse though the use of application frameworks became popular. [66] provides a snap-shot of 24 application frameworks proposed in a variety of domains, such as Business and Artificial Intelligence. The concept also matured and was described by Fayad *et al.* in [65] as:

> ...a reusable design of a system that describes how the system is decomposed into a set of interacting objects.

Application frameworks defined this way have four main benefits, as described by [65]:

1. *Modularity*: Application frameworks encapsulate the volatile implementation details behind stable interfaces. The stable interfaces help localize the impact of design changes and reduce the efforts required to understand and maintain existing software.

2. *Reusability*: The stable interfaces provided by the application framework allows plug-ins that are developed for one application to be used in another that is based on the same application framework. This avoids re-creating and re-validating common solutions for recurring application requirements and software design challenges.

3. *Extensibility*: Application frameworks allow stable interfaces to be extended to ensure for the timely customization of new application services and features.

4. *Inversion of control*: The application framework provides a software environment where the overall program's control flow is dictated, not by the caller of the framework, but by the framework itself. This allows the framework to apply transformations to the execution in order to make it more efficient.

The four benefits of application frameworks for an object-oriented application framework are all targeted toward productivity enhancement. The efficiency enhancement of the application frameworks is less clear. While more efforts generally go into the optimization of the components that can be reused, the fact that an application framework is used does not guarantee execution efficiency.

In contrast, the primary concern of application frameworks for parallel programming is the applications' execution efficiency. The application frameworks being developed in this thesis strive to help application domain experts to more effectively program highly parallel manycore platforms, where the primary motivation for domain experts is the platforms' execution performance potential.

Application frameworks for parallel programming have been proposed for scientific computing on large computing clusters. Examples include Cactus [75] for structured grids based applications, and CHARMS [77] for computational chemistry applications. Their development often involved multi-year efforts by a large team of developers. As multi-GPU workstations today often pack the compute throughput that used to be common in computing centers a few years ago, the demand of application frameworks is shifting towards desktop applications in a more diverse set of application domains. As such, we are already seeing frameworks emerging for GPU-based computing that target machine learning and computer vision applications [40, 13]. By providing reference implementations and defining the extension points on top of the proven software designs, we hope to create light weight application frameworks that are easy to construct and easy to use. We expect this type of application framework to become more important as more computer-literate application domain experts develop applications for high performance workstations or even portable devices.

In terms of successful application frameworks deployed in the industry, Hudson is an exemplar of an application framework that is used for software build-automation [93]. Hudson provides a web interface for managing hundreds of third party tools used to performing functions such as automated self-testing, bug analysis, and documentation generation with an extension point interface. Hudson allows anyone to implement plug-ins for various third-party tools such that users can easily manage tools in one automated software production application framework. Although Hudson is implemented in Java, many of the concepts like

narrow extension point interfaces are language agnostic and have influenced the design of the pattern-oriented application framework discussed in this thesis.

3.3.4 Pattern-oriented Application Frameworks

In this thesis, the application framework is developed in order to help application domain experts more effectively program highly parallel manycore platforms. The development of the application framework is guided by the software architecture, which is the hierarchical composition of the computational and structural patterns used in an application. A *pattern-oriented application framework* is defined as:

> A software environment in which user customizations may only be in harmony with the underlying software architecture, which is the hierarchical composition of the computational and structural patterns, for a class of applications.

For example, if the application framework is based on pipe-and-filter, then customization only involves the modification of pipes and/or filters. As outlined in [14], an application domain expert can be served by application frameworks in two ways: First, the application domain expert works within a familiar environment using concepts drawn from the application domain. Second, the application framework prevents the expression of many annoying problems with respect to parallel programming such as race hazards[3], deadlocks[4], and starvation[5].

An exemplar of a pattern-oriented application framework is the Ruby-on-Rail application framework for web application programming [19, 104]. Its development was based on the Model-View-Controller structural pattern, which was first referenced in the late 1970s with the Smalltalk-80 user interface design [73].

In the Model-View-Controller structural pattern, the problem is expressed in terms of the challenge of designing a flexible user interface to allow the frequent adjustment of the presentation, the safe manipulation of complex models, and a modular approach in the implementation; the solution involves three interacting subsystems, including the model, the view, and the controller. This three-subsystem approach enforces the separation of roles and restricted accesses, permits independent development, as well as the testing and maintenance of each subsystem. In particular, the *Model* can only be modified by the *Controller* using specific routines; the *View* can only query the state of the model; and the *View* can only be selected by the *Controller*.

Ruby is a programming language that is dynamically typed, interpreted, and can be modified at runtime. It originated in the mid-1990s and was first developed and designed

[3]Race hazards: a condition where the result of a parallel execution is dependent on the sequence or timing of the program execution. A potentially wrong result can be generated through some ordering of the program execution. Bugs caused by race hazards are notoriously hard to reproduce, making them difficult to resolve.

[4]Deadlock: when two or more competing threads are each waiting for the other to finish

[5]Starvation: the condition when one of the many threads is perpetually denied necessary resources, such that the application cannot complete the necessary tasks.

by Yukihiro Matsumoto [105]. Rails is a "gem", or a library in Ruby that was developed by David Heinemeier Hansson to assist in constructing web applications, providing classes for saving to the database, handling URLs and displaying HTML. It was extracted from David's work on Basecamp, a project management tool by 37signals (a web application company) [76].

A developer defines the *Model* using Rails' scaffolding mechanism, which creates a set of default *Controller* actions (i.e. URLs to visit) and a *View* (forms to fill out) to interact with the specified model. The scaffolding approach is typical of an application framework, which put in place a reference design, such that the subsystems can then be customized by the developer as the end-user application for the specific usage scenario is constructed.

Rails uses a specific directory structure with *Models* in *app/models*, *Controllers* in *app/controllers*, and *Views* in *app/views*. These naming conventions let Rails applications find its components without additional configuration. It also encourages the customizations to the framework to be in harmony with the model-view-controller software programming pattern. When a developer stays within the scope of the pattern, much of the complexity of interacting with a webserver and executing maintenance tasks is abstracted away.

The pattern-oriented application frameworks for parallel programming developed as part of this thesis were inspired by pattern-oriented application framework for web programming tools like Ruby-on-Rail. However, some of the applications described in this thesis have a much more complex software architecture that is hierarchically composed of multiple parallel programming patterns. An example of this is the pattern-oriented application framework for automatic speech recognition. Its software architecture is discussed in details in Section 4.4.2.

3.4 Parallel Software Implementation Tools

In order to effectively utilize highly parallel computing platforms, the tools described in the previous section for software implementation support can be implemented using a variety of languages and APIs. Some common infrastructures include: processes [147], threads [111], MPI [67], OpenMP [54], CUDA [122], and OpenCL.

A process is an instance of a computer program that is being executed. The concept first appeared in computer control software in the early 1960s, when multiple programs running as separate processes were able to share the scarce resources of a uniprocessor computer [147]. In a modern operating system, the process retained its characteristic as an independent instance with a separate memory address space, with considerable state information and resources include memory, file handles, sockets, device handles and windows. Such independence allows each process to accomplish a variety of functions. It also allows an operating system to efficiently manage multiple processes to share the same hardware resources. On a parallel computing platform, computation can be separated into processes and mapped to different hardware computing resources. If processes need to coordinate actions, they can communicate through system-provided inter-process communication mechanisms. When multiple processes share a computing resource, the operating system is able to preemptively determine when to context-switch between processes.

Processes are used for parallelizing large applications at the scale of computing clusters, as the scope of a process nicely abstracts the resources on a computing cluster node. However, the challenge of using processes lies in the significant overhead of creating and managing a process, as each process carries a considerable amount of state information, and its resources must be allocated by the operating systems. A block of computation must contain millions instructions in order for it to be worthwhile to be parallelized by creating another process.

Threads are small units of execution within a process. The concept of sharing a single execution path among multiple threads of execution appeared as early as in the 1950s [138]. Threads within a process share the states, memory and resources of the process. The sharing of states and memory means that the threads within a process can communicate with each other through the use of share variables at particular memory locations, which is much more efficient than going through system-provided inter-process communication mechanisms. With less states associated with each thread, the creation of a thread and context switching between threads involves much less overhead. An application can be partitioned into finer-grained computation and constructed with threads in a process. These different threads can then make use of the multiple computing resources that support the shared-memory-space abstraction, such as on a multicore microprocessor.

There are two main challenges in the use of threads for the deployment of programs on manycore multiprocessors. First, writing shared-memory programs is an error-prone process, with many opportunities for race hazards, deadlocks, and starvation, even for simple setups. Second threads as implemented in POSIX threads still involve significant creation overhead that require the computation granularity to be as large as 10,000 to 100,000 instructions to achieve compute efficiency.

To provide a more user-friendly abstractions to make parallel application programming less error-prone, application programming interfaces (API), such as MPI and OpenMP, have been proposed to provide assistance to manage and partition computation, and instantiate common communication patterns. To reduce parallel thread management overhead, other programming infrastructures such as CUDA and OpenCL, along with their associated hardware supports, were developed to allow much finer-grained application concurrency to be exploited.

MPI (Message Passing Interface) is an API specification that allows computers in a computer cluster to communicate and coordinate with each other. First presented in 1994 [67], it is the *de facto* standard in the scientific computing community as an abstraction for writing parallel programs running on distributed memory systems. It uses the Single Program Multiple Data (SPMD) abstraction, which assigns each parallel process a unique process ID for each process in order to identify itself and to coordinate actions among the processes. High performance communication functions, such as MPI_Bcast (for broadcasting data from one process to all processes) and MPI_Reduce (for operations such as summing across all processes), are provided.

OpenMP (Open Multi-Processing) is an API specification introduced in 1997 that supports parallel programming on shared memory platforms [54]. It uses pragmas[6] to allow

[6]Pragma: a directive in a programming language communicating additional implementation-specific

portions of a sequential program to be marked as parallelizable, and automatically forks off threads according to the available resources at runtime. The API also provides assistance for describing critical sections in data sharing among the parallel threads, as well as assistance operations that require global communication operations, such as reductions.

In programming manycore computing platforms, such as the GPU, the performance sensitivity to more nuances in the hardware architecture places additional demands on the programming infrastructure.

The GPU is programmed using the CUDA programming framework [122]. An application is organized into a sequential host program that is run on a CPU, and one or more parallel kernels that run on a GPU, or on multi-core CPUs through tools such as MCUDA [143]).

A kernel executes a scalar sequential program across a set of parallel threads. The programmer organizes these threads into *thread blocks*, where a thread within a thread block can efficiently coordinate actions with each other by means of barrier synchronization. A kernel consists of a grid of one or more blocks, as the size of each block is limited[7]. The threads in a thread block may also share a shared memory space private to that block. The programmer must specify the number of blocks in the kernel and the number of threads within a block when launching the kernel. Each thread block is executed on one streaming multiprocessor (SM). While synchronization within a block is possible using barriers in the shared memory, global synchronization across all of the blocks is possible either at kernel boundaries, or through the use of global atomic instructions. The SIMD structure of the hardware is exposed through thread warps. Each group of 32 threads within a thread block executes in a SIMD fashion on the scalar processors within an SM, where the scalar processors share an instruction unit. Although the SM can handle the divergence of threads within a warp, it is important to keep such divergence to a minimum for best performance.

Each SM is equipped with an on-chip scratchpad memory that provides a private per-block shared memory space to CUDA kernels. This shared memory has very low access latency and a high bandwidth. Along with the SM's lightweight barriers, this memory is an essential mechanism for efficient cooperation and communication amongst threads in a block. In the most recent Fermi architecture from NVIDIA, part of the per-block shared scratchpad memory can be configured as a hardware-controlled cache.

Threads in a warp can perform memory loads and stores from and to any address in memory. However, when threads within a warp access consecutive memory locations, the hardware then can coalesce these accesses into an aggregate transaction for higher memory throughput.

While CUDA is a vendor specific programming infrastructure, OpenCL (Open Computing Language) is an open framework put in place by the Khronos Group in 2008 for writing CUDA-like parallel programs that execute across heterogeneous platforms such as CPUs, GPUs, and other processors [78].

In this thesis, as OpenCL was still being defined during the period of this research, we

information

[7]Depending on the implementation hardware, there can be a maximum of 512 to 1024 threads within a thread block

use CUDA as the development platform. It should be noted that the pattern-oriented application framework concepts discussed here are not limited to CUDA as an implementation platform and can just as well be implemented in OpenCL.

3.5 Summary

This chapter surveyed existing tools that can help application domain experts more effectively program manycore microprocessors. We first defined "effectiveness" as achieving application execution efficiency, development productivity, and the portability of developer efforts.

Existing tools from three areas are explored in sections: *software architecture narrative* tools (Section 3.2), *software implementation support* (Section 3.3), and *parallel software implementation tools* (Section 3.4).

Software architecture narrative tools help application domain experts grasp the background knowledge necessary to understand the opportunities and challenges in developing a parallel application. These tools include idioms, patterns, and pattern languages. While assistance in understanding the background contributes to higher development productivity for an application domain expert, using it alone is not enough to close the implementation gap: an application domain expert still needs significant expertise in the parallel hardware platform to implement the applications.

Software implementation support tools assist application domain experts with the implementation of the applications. These tools include libraries, skeletons, and frameworks. Libraries have concise APIs and are great abstractions for complex tasks. However, they often implement fixed functions and are relatively inflexible to adapting to new application requirements. Skeletons provide the outline of an algorithm in a programming environment, and users can incorporate custom functions within the skeleton as "flesh" code. They are more flexible than libraries, and target applications with a single dominant algorithm well. However, they provide little help with the composition of multiple levels of algorithms in complex applications. The frameworks considered in this thesis are application frameworks that often involve complex compositions of algorithms. They are tools for propagating proven software designs and implementations in order to reduce the cost of developing similar solutions and to improve the quality of software. The pattern-oriented application framework imposes an additional constraint on the concept of application framework such that customizations of the framework may only be in harmony with the underlying software architecture. Ruby-on-Rails was presented as an exemplar application framework for web application development implementing the Model-View-Controller structural pattern. Such prior work influenced the development of pattern-oriented application frameworks for highly parallel computing platforms, where efficiency, productivity, and portability considerations are addressed.

Parallel software implementation tools are used to construct tools for *Software implementation support*. The common languages and APIs, such as processes, threads, MPI, OpenMP, CUDA and OpenCL have been discussed. Concepts from processes and threads form the basis of parallel processing paradigms. MPI and OpenMP build on top of pro-

cesses and threads with SPMD, loop level, as well as task level parallelism abstractions to assist developer in utilizing cluster and multicore platforms. In order to target the level of parallelism within a manycore microprocessor, CUDA and OpenCL provides the necessary abstraction to effectively exploit fine-grained parallelism.

In this thesis, we focus on pattern-oriented application frameworks for domain experts to effectively utilize highly parallel manycore microprocessors. Although the application framework are constructed with CUDA on NVIDIA GPUs, the techniques described here are not limited to the CUDA environment or the GPUs, and are applicable to the broad challenge of the effective utilization of highly parallel microprocessors and systems.

Chapter 4

Pattern-Oriented Application Frameworks for Parallel Programming

In order to help application domain experts to more effectively utilize highly parallel manycore microprocessors and productively implement efficient applications in a portable way, we see four types of assistance to be critical:

1. To provide assistance to better understand the landscape of parallelization opportunities or application concurrency
2. To provide assistance to better understand well-thought out techniques for exploiting the application concurrency, as well as the trade-offs possible with the implementation techniques, along with the common bottlenecks to avoid.
3. To provide a reference design to demonstrate how the techniques for exploiting application concurrency can be integrated together in order to achieve efficient execution.
4. To allow flexible customization of the reference design to implement a full class of applications so as to target different usage scenarios.

These types of assistance are precisely what a pattern-oriented application framework provides for a domain expert. A pattern-oriented application framework for domain experts is a *software environment* built around an underlying *software architecture* for a class of applications in a domain (such as Figure 4.5 on page 41) in which user customizations may only be applied in harmony with the software architecture. In our approach, the software environment includes the *application context*, the *software architecture*, a *reference implementation*, and a set of *extension points*. Software architecture is defined as a hierarchical composition of parallel programming patterns [94], which are solutions to the recurring problems in parallel programming.

The application framework for parallel programming crystallizes in a reference implementation the parallel programming optimizations that are applicable to a class of applications based on their common software architecture. It also provides extension points to allow

flexible customizations of the reference implementation with plug-ins that target specific application usage scenarios.

This chapter introduces the four components of an application framework (Chapter 4.1), discusses the design philosophy behind the application framework (Chapter 4.2), illustrate the implications of the application framework in a parallel software development flow (Chapter 4.3), and gives a detailed explanation of each of the components (Chapter 4.4). Each component is illustrated with a sample pattern-oriented application framework for automatic speech recognition (ASR).

4.1 The Four Components of an Application Framework

In our approach, there are four main components in the pattern-oriented application framework targeted for parallel programming, including the *application context*, the *software architecture*, the *reference implementation*, and a set of *extension points*.

The *application context* is a description of the characteristics and requirements for the class of applications. This component exposes the parallelization opportunities of an application that are independent of the implementation platform. For application domain experts, it provides the context with which they can understand the motivation of parallelization decisions made in the software architecture of an application framework. Section 4.4.1 describes the application context for an automatic speech recognition application.

The *software architecture* description is a set of concise documentations of the organization and structure of the class of applications that is described using the software parallel programming. This component presents a hierarchical composition of the parallel programming patterns that assist in the navigation of sample implementation. For application domain experts, the description of the software architecture allows them to quickly gain an understanding of the opportunities and challenges in the implementation of an application. This helps the domain experts better organize their efforts around the fundamental limitations and constraints of the implementation of the application with highly parallel microprocessors. Section 4.4.2 describes a software architecture used for a speech recognition application in great detail.

The *reference implementation* is a fully functional, efficiently implemented, sample parallel design of the application. This component provides a concrete example of how each component in the application framework can be implemented, as well as how they can be integrated. It is also a proof of the capability of a computing platform for the class of applications that the application framework is designed for. For application domain experts, such implementation relieves the burden of constructing functionally correct baseline implementation before introducing new features. With the aid of the sample implementation, application domain experts can focus on the particular modules that have to be designed in order to meet the needs of specific end-user products. A detailed example of a reference implementation for ASR is described in Section 4.4.3.

The *extension points* are a set of interfaces that are defined to summarize the interactions

between the application framework and any potential new plug-in modules. They invite developers to customize specific modules using pre-determined interfaces. They also allow for customizations that would not jeopardize the execution latency or any throughput sensitive features in an application framework. For an application domain expert, the extension points provide a flexible interface for the implementation of plug-ins while maintaining the efficiency of the final implementation. Examples of extension points for the speech recognition application framework are described in Section 4.4.4.

Section 4.3 offers some ways to utilize the application framework components in a parallel application design flow. In brief, Figure 4.1 shows the *application context* and *software architecture* assist an application developer to *specify* the application and its concurrency opportunities to be exploited. The *reference implementation* provides a functional design to help developers evaluate the performance potential on a computing platform, as well as to *match* the end-user application to an application framework. The *extension points* help abstract away the rest of the application and allow the developer to focus on *customizing*, which involves extending the functionality and implementing new features for an application.

4.2 Design Philosophy

Design philosophy clarifies the goals and their importance in the design process. The application framework is developed to be part of a software development flow that is coherent with the concept of "marginal utility," which observes that people assess value only on the "margin," where the benefit to them is new and uncommon. Informal conversations with application domain experts have shown that a tool for application development is usually considered worth trying when one can set it up and get initial results in one afternoon and be able to use it effectively to implement new capabilities in one week.

The value of an application framework is therefore dependent on:

1. How quickly an application domain expert can use it to effectively start utilizing a highly parallel manycore computing platform

2. How quickly an application domain expert can start developing new functions that efficiently execute on these platforms

With the availability of *reference implementation* in the application framework, an application domain expert can immediately work with a functional and efficient starting point to apply an application to an usage scenario. The *application context* description and the *software architecture* in the framework can provide the necessary guidance to the application domain experts so they can utilize the relevant *extension points* to apply his/her own customizations. Each of the four main components of the application framework is essential in maximizing "marginal utility" for an application domain expert looking to utilize highly parallel microprocessors.

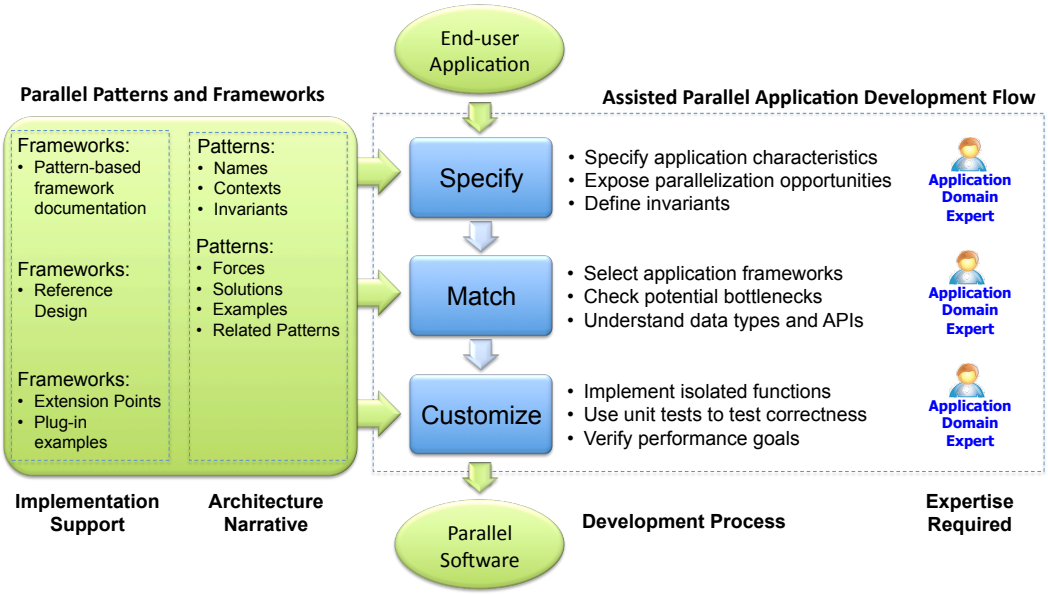


Figure 4.1: The three-step process in an assisted parallel application development flow

4.3 The Implications of Application Development Flow

In order to solve the dichotomy of the increasing demand for parallel application and the scarcity of developers with both application domain and parallel programming expertise who can effectively develop parallel applications, we can provide tools to help developers take care of their application development concerns.

For this thesis, we assume that application domain knowledge is essential for the implementation of software applications. The assistance proposed here is in the form of parallel programming patterns and parallel application frameworks that allow a software developer to describe, define and implement the architecture of a piece of software.

Figure 4.1 shows that with the assistance provided by programming patterns and frameworks, an application domain expert can follow a new three-step development flow where the steps are to: Specify, Match, and Customize. The Specify step is where one describes application characteristics, exposes application concurrency (or parallelization opportunities), and defines the invariants. The Match Step is where one selects the application framework to use, checks for potential performance bottlenecks, and gets an understanding of the data types and APIs of the extension points to customize. The Customize step is where one develops the plug-in modules so as to obtain new functions for the end application.

By using the parallel programming patterns, an application domain expert is made aware of the hidden concerns that must be taken care of in developing parallel applications. There are two important benefits of using the parallel programming patterns. Firstly, it brings forth potential implementation challenges that typically plague the solution of a pattern such that the engineering efforts can be coordinated to prototype the right bottlenecks to remove

implementation risks. We demonstrate this process in Section 4.4.2, where we evaluate parallelization opportunities in the ASR application with the known potential bottlenecks in the corresponding parallel programming patterns. Secondly, parallel programming patterns allow the application domain expert to seek application-level transformation that reduces or even eliminates the limitations imposed by a particular style of implementation. We demonstrate this process in Section 5.1.4, where we applied a complex application-specific data format transformation to trade-off the fixed overhead of data-parallel iterations with extra work per data-parallel iteration to improve overall application performance.

When an application framework is available, the application framework can take care of the hidden concerns, that is, as long as the application domain expert applies his/her customizations with respect to the provided extension points. A parallel programming expert is no longer required in the development process, opening up new opportunities for classes of applications to be developed by more application experts. This removes implementation platform expertise from being a significant limiting factor in the demand for highly parallel computation platforms.

When an application framework is not available, it can be constructed from a reference implementation. Figure 4.2 shows how the Specify-Architect-Implement three-step process can be extended with a fourth step: the *leverage* step. This step links the Specify-Architect-Implement process to the Specify-Match-Customize process by producing the parallel programming patterns and the application frameworks necessary to enable the Specify-Match-Customize process. The *leverage* step involves re-factoring the implementation structure, implementing the extension points such that additional plug-ins can be developed.

In terms of the prerequisites for the Leverage Step, the performance goals in the Implement Step should have been verified before one starts building an application framework in the Leverage step. Failure to do so will lead to wasted effort in building an application framework for which there is no demand for reuse.

4.4 Application Framework Component Details

This section offers an overview of the *features* of the four components of application framework, how they *function* within a parallel application development flow, and how these functions *benefit* application domain experts as they leverage highly parallel computing platforms to produce end applications for users. The explanations here reference components from an ASR application framework as examples.

ASR is emerging as a critical component in data analytics for a wealth of multimedia data [153]. Speech recognition technology allows multimedia contents to be transcribed from acoustic waveforms into word sequences (Figure 4.3). Commercial usage scenarios for ASR are now appearing in both data centers and portable devices[1]. We present the ASR application framework components in this section and demonstrate how a speech expert has achieved a 20x speedup with the framework in recognition throughput as compared to the sequential CPU implementation in Section 6.1.

[1]iPhone applications such as Jibbigo provide speech recognition in the client device as part of a traveler's speech-to-speech translator

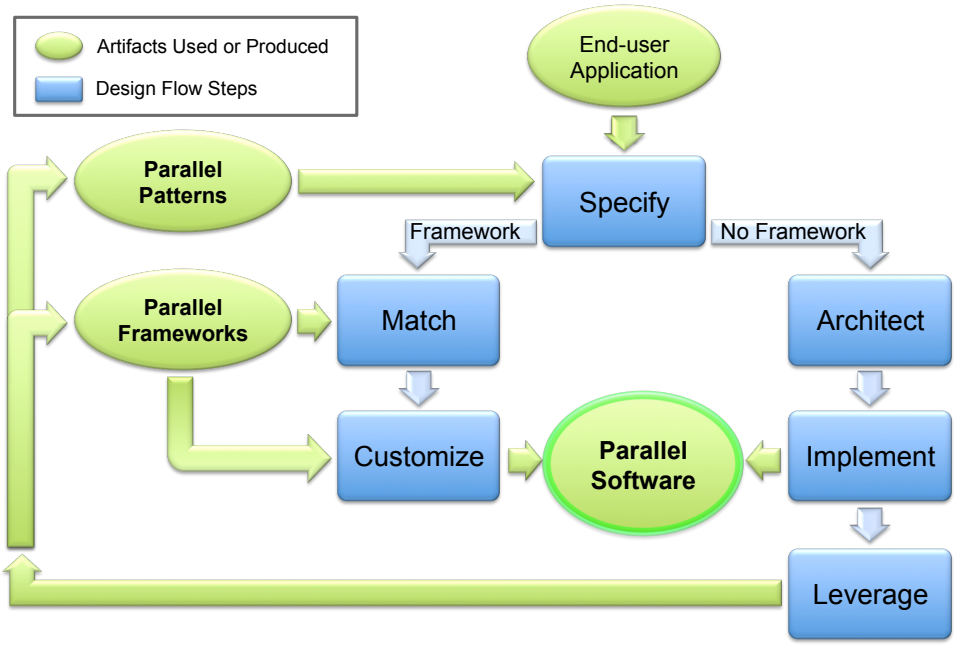


Figure 4.2: Overall parallel application development flow.

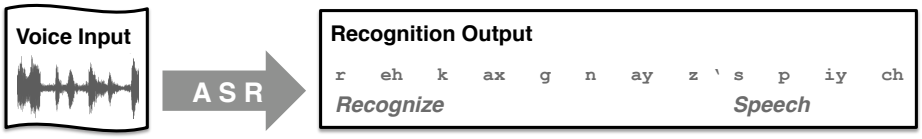


Figure 4.3: Automatic Speech Recognition (ASR) extracts phonetically-relevant features from a speech signal, estimates phone likelihoods, and infers word sequences.

4.4.1 Application Context

In this section, we discuss the *application context*, which is first of the four components of a *pattern-oriented application framework* by presenting its *features*, *functions*, and *benefits* to the application domain expert. We then use a component from an ASR application framework as an example.

Feature

The description of an application context contains an elaboration of the characteristics of a class application. This includes the typical input and output data types, the typical working set sizes, the typical modules and their inter-dependence, as well as the constraints that must be met for the application to be viable for the end-user. The constraints could be defined in many areas such as in latency, throughput, power, and maintainability.

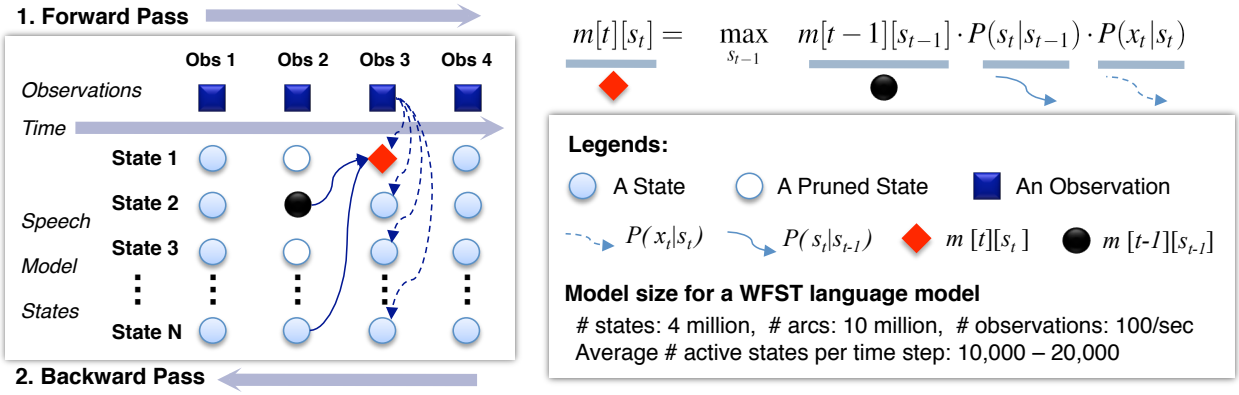


Figure 4.4: Application characteristics: the inner-workings of the performance critical Viterbi forward and backward pass steps

Function

The goal of application context description is to expose the available concurrency in a class of applications. Concurrency can also be described as parallelization opportunities. They are the state of a computation in which more than one task is active and able to make progress at one time. In a software application, concurrency allows the application to make forward progress through executing multiple tasks at the same time.

Benefits

An application domain expert can examine the context in which the application framework is built and evaluate whether the application he/she would like to build is a variant of the class of applications that the application framework is designed for. Before investing more time learning about the framework, a decision can be made about the *relevance* of the framework for the specific application usage scenario at hand.

Example

Application Characteristics: A Large Vocabulary Continuous Speech Recognition (LVCSR) application analyzes a human utterance from a sequence of input audio waveforms to interpret and distinguish the most likely words and sentences intended by the speaker (see Figure 4.3). The analysis involves iteratively comparing a sequence of features extracted from the input audio waveform (as observations) to a speech model that has been trained using powerful statistical learning techniques. To adapt to different languages, acoustic environments, and vocabulary domains, only the speech model needs to be replaced, with the recognition process staying the same.

The process of recognizing speech is a type of temporal pattern recognition, which is a well-known application of the hidden Markov model (HMM) [129]. The states in the HMM for speech recognition are components of words in a vocabulary. They are hidden because they can only be observed through interpreting features in an acoustic waveform.

The Viterbi algorithm [121], with a forward-pass and a backward-pass step, is often used to infer the most likely sequence of words given the observations from the acoustic waveform.

Figure 4.4 shows that there are two main phases in the forward-pass of the inference algorithm. Phase 1 is shown in Figure 4.4 as dashed arrows between observations and states. It evaluates the observation probability of the hidden state. This matches the input information to the available acoustic model elements and only takes into account the instantaneous likelihood of a feature matching an acoustic model element. Phase 2 is shown in Figure 4.4 as the solid arrows between states of consecutive time steps. It references the historic information about what the most likely alternative interpretations are in the utterance heard to that point; it computes the likelihood of incorporating the current observation given the pronunciation and language models. The computation for each state s_t at time t (with the diamond-shaped state as an example) records the state transition from the prior time step $t-1$ that produced the greatest probability $m[t][s_t]$.

Inputs and Outputs: In a standard speech recognition application, the acoustic inputs are features that are extracted from acoustic waveforms, typically at 10ms time steps. The number of features used varies among different information sources, languages and acoustic environments in the recognition scenario. For example, when recognizing English with a single microphone in meeting rooms, a common feature vector size is 39. The speech models used in this application framework contain information from acoustic models, pronunciation models, and language models. They are statically combined using weighted finite state transducer (WFST) techniques into a monolithic graph structure [112]. Depending on the language models, the speech model graphs often contain millions of states and tens of millions of arcs.

Working Set Size: When using the speech model during inference, one can achieve good accuracy by tracking a small percentage of the total number of states representing the most likely alternative interpretations of the utterances. In our experiments, we found that tracking more than 1% of the most likely alternative interpretations provides diminishingly small improvements in accuracy while requiring a linear increase in execution time. Thus, the working set of active states is kept below 1% of the total number of states in the speech model, which is, on average, 10,000 to 20,000 active states.

Concurrency: There are four main levels of concurrency in the inference process. We have provided an application description and have highlighted the amount of concurrency available at each level. Detailed descriptions of these opportunities are treated in more depth in Section 5.1.1. The opportunities and challenges posed by these levels of concurrency in an implementation are explored in Section 4.4.2.

1. Different speech utterances can be distributed to different machines for batch-mode processing. A typical conversational utterance can be 5-30 seconds long, and a one-hour audio input can be distributed to hundreds of machines for processing. Each inference process requires billions of instructions and can last seconds. This approach uses the MapReduce structural pattern and is effective in improving recognition throughput. However, it will not improve recognition latency for single-user on-line applications such as live sub-title generation.

2. For a set of utterances, if the forward and backward passes in the Viterbi algorithm are handled by dedicated computing resources such as by different cores or different processors, the two passes can be pipelined. When utterance A has completed the forward-pass and proceed to compute the backward pass, utterance B can initiate its forward-pass. This approach uses the Pipe-and-filter structural pattern, and can improve both recognition throughput as well as latency.

3. In the forward-pass, if phases 1 and 2 are handled by dedicated computing resources, the two phases can be pipelined: i.e. Phase 2 of one time step can execute on one computing resource while Phase 1 of the next time step can execute on another computing resource. This approach uses the Pipe-and-filter structural pattern, and can improve both recognition throughput as well as latency.

4. Within each of the functions in phases 1 and 2 of the forward pass, there are thousands of observation probabilities and tens of thousands of alternative interpretations of the utterance to track. Each can be tracked independently with some amount of synchronization required after each function, with each unit of work usually being no larger than tens to hundreds of instructions. This approach uses the MapReduce structural pattern, and can improve both recognition throughput as well as latency.

Performance Constraints: The primary goal of automatic speech recognition is to transcribe a sequence of utterances as fast as possible with as high an accuracy as possible. For commercial applications, there is usually an accuracy threshold that makes the usage scenario realistic and practical. For example, for in-car command and control applications, one may tolerate a 5% command error rate in the interpretation of non-essential commands. For data analytics, where one searches for the keywords in a recorded telephone conversation, a 50% word error rate (WER) may be tolerable and still yield useful results. In both usage scenarios, higher recognition accuracy will help reduce user frustration caused by recognition errors. Other goals such as more reliable automatic recognition confidence estimation for the recognized results can allow human based back-off systems to be used to improve user experience.

4.4.2 Software Architecture

In this section, we discuss the *software architecture*, which is second of the four components of a *pattern-oriented application framework* by presenting its *features*, *functions*, and *benefits* to the application domain expert. We then use a component from an ASR application framework as an example.

Feature

Software architecture is the organization of a software program expressed as the hierarchical composition of structural and computational programming patterns.

This composition is often hierarchical: There can be one pattern matched for the top-level architecture, and more patterns matched for the sub-modules of the top-level architecture.

For example, the top-level architecture for a speech recognition application matches a pipe-and-filter pattern that has two "filters," or modules. The first module is the feature extractor, which transforms an input audio waveform into a stream of feature vector[2]. The second module is the speech inference engine which analyzes the stream of feature vectors and infer the most-likely word sequence present in the audio.

The first module can be further associated with a the pipe-and-filter pattern with a set of feature extraction techniques as "filters" and the various intermediate domain of representations such as amplitude, spectrum, and power spectrum as "pipes". The second module can be further associated with the bulk synchronous pattern, where each iteration integrates the information presented by a feature vector in a particular time step with the state derived from all of the preceding feature vectors. The sub-modules of the feature extractor and the speech inference engine can be further associated with other parallel programming pattern.

The composition may not be unique: there can be many ways to architect an application implementation and multiple sets of associations of application context to software architecture are possible. The software architecture for an application framework focuses on describing the structure and organization for the application framework.

The composition references parallel programming patterns: The parallel programming patterns are known, well-analyzed solutions to recurring problems in parallel programming. A set of such patterns is outlined in Section 3.2.2. Each pattern features a problem that is often encountered in parallel programming, the context in which the problem is encountered, the forces guiding the trade-offs in implementing a solution, the solutions that have previously been successfully used to solve the problem, and examples illustrating the solution process. The set of patterns also form a pattern language where alternative patterns are referenced.

Function

The software architecture is used to introduce the structure and organization of the application framework and to convey the opportunities and challenges in parallelizing this structure and organization. The information is introduced and conveyed using parallel programming patterns in English (or any other natural language) prose. The goal here is to guide the application level design trade-offs to focus on the alleviation of critical performance bottlenecks.

Benefits

By identifying the parallel programming patterns that are used, a host of well-thought out solutions are brought forth for the application domain expert, as well as the impor-

[2]In this case, the feature vectors abstract information contained in the waveform that is relevant for speech recognition.

tant forces that highlight design trade-offs. The level of details required is associated with whether an extension point for which the application exists.

Example

A software architecture is the organization of a software program expressed as the hierarchical composition of patterns [94]. In the application framework, the software architecture expresses the composition of the reference implementation and reflects the decisions made when mapping the application concurrency to the parallel hardware resources.

The hardware resource that is targeted in our ASR application framework is the GPU, which is an offload device with the CPU acting as the host engine. Programs running on the GPU are written in CUDA [4]. CUDA programs are invoked from the host CPU code, and operands must be explicitly transferred between the CPU and the GPU. Recent GPUs like the GTX480 contain 15 cores each with dual issue 16-wide SIMD units, with non-coherent caches and scratch space memories that are available in each core. In order to efficiently program the GPU, one must efficiently leverage the wider SIMD units, the GPU memory hierarchy, and the synchronization primitives within and between the cores.

Concurrency Exploited: In our ASR application framework for highly parallel implementations on manycore processors, we have selected the fourth type of concurrency to exploit: the fine-grained parallelism within each of the functions in Phase 1 and Phase 2 of the forward pass of the inference algorithm. While this choice may be the most complex to implement, it offers the most efficient utilization of the manycore platform.

When mapping the application onto a manycore platform, the following thought experiments were performed to help eliminate the choosing of any of the first three types of concurrency:

1. Concurrency among speech utterances can be exploited over multiple processors and is complementary to the more challenging fine-grained concurrency explored among the cores on a chip and among the vector lanes within a SIMD unit. However, exploiting concurrency among speech utterances among cores and vector lanes is not practical. With tens of cores sharing the same memory sub-system, the available memory capacity and memory bandwidth in a GPU cannot accommodate the working set sizes of tens of concurrent speech inference processes. Also, while this level of concurrency is useful for batch recognition systems, it is not effective to speedup single-user online recognition systems.

2. When different forward and backward passes are mapped onto different resources, referring to the pipe-and-filter computational parallel programming pattern, load balancing becomes the most significant factor in achieving efficient utilization. Since backward pass performs less than 1% of the work that is done by the forward pass, the source of concurrency is not suitable for exploitation.

3. Depending on the model parameters that are used, phases 1 and 2 of the forward pass do similar amounts of work. However, referring to the pipe-and-filter computational parallel programming pattern, communication between the "filters" along the "pipes"

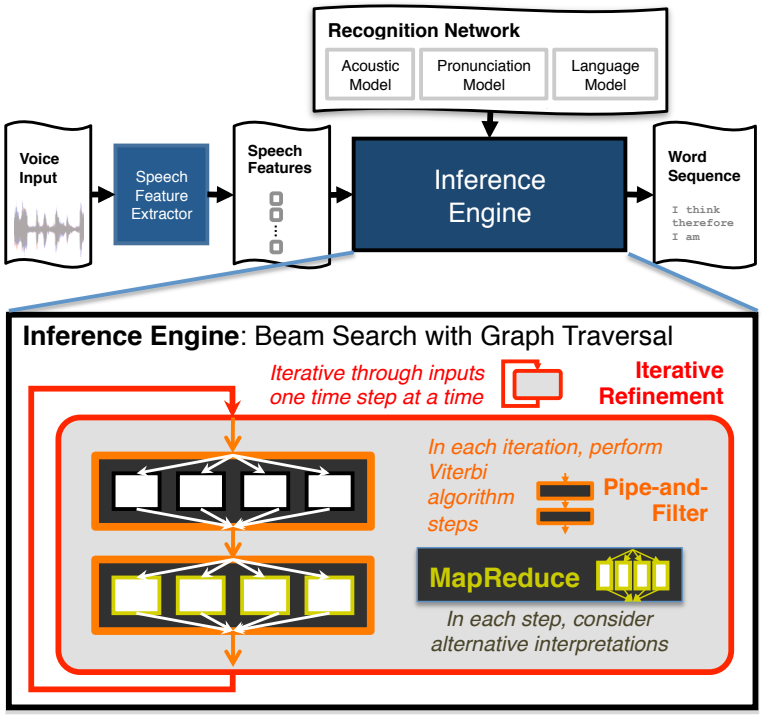


Figure 4.5: The software architecture of a large vocabulary continuous speech recognition application. At the top level, the application can be seen as an instance of the *Pipe-and-filter pattern*, with the speech feature extractor and the inference engine as filters, and the intermediate results between them on pipes. Inside the inference engine, the iterations over each time step in the application is based on the *Iterative Refinement pattern*, where each iteration handles one input feature vector corresponding to one time-step. Inside each iteration, phases 1 and 2 can be seen as filters in a *Pipe-and-filter pattern*. Within each phase, the computations can be executed in parallel following the *MapReduce pattern*.

may limit performance. In such a case, if phased 1 and 2 are implemented in the GPU and CPU, respectively, the amount of intermediate results that must be transferred between them can lead to a performance bottleneck. Indeed, this situation is observed in [63].

Architecture Representation: Figure 4.5 shows the software architecture for the ASR application framework as a hierarchical composition of parallel programming patterns. The top level of the inference engine may be associated with the Iterative Refinement Structural Pattern [94], where each iteration handles one input feature vector corresponding to one time-step. The computation for the entire iteration is mapped onto the GPU device, such that the computation throughput will not be bottlenecked by intermediate results that would be transferred between the CPU and the GPU. The work within each iteration can be associated with the Pipe-and-filter Structural Pattern [94], where the phases can be considered as filters. The computation in each of the two phases can be associated with the MapReduce Structural Pattern [94], where thousands of observation probabilities are com-

puted in parallel, and tens of thousands of alternative interpretations of a speech utterance are tracked in parallel.

Challenges: While this software architecture maps well with respect to the many computing resources on the manycore GPU devices, it also presents significant challenges in regard to global synchronizations between different algorithm steps. For example, in Phase 2 of the forward pass of the Viterbi algorithm, the inference process is based on parallel graph traversal, a well-known and challenging problem in parallel computing [107], especially in the context of speech recognition [86]. The parallel graph traversal operates on an *irregular subset* of 10,000 states of the speech model representing the most likely alternative interpretations of the utterance; it frequently updates *conflicting memory locations*. Correctly implementing such challenging tasks in parallel while optimizing for metrics such as memory bandwidth are often beyond what most application domain experts would like to undertake. Section 4.4.3 elaborates on how this challenge can be resolved by the use of application framework, which provides a reference implementation that encapsulates an efficient solution to these implementation challenges.

4.4.3 Reference Implementation

In this section, we discuss the *reference implementation*, which is third of the four components of a *pattern-oriented application framework* by presenting its *features*, *functions*, and *benefits* to the application domain expert. We then use a component from an ASR application framework as an example.

Feature

The *Reference Implementation* component of the pattern-oriented application framework is a fully functional, efficient sample parallel design of the application. It provides a description of the data structures and application modules, an explanation of their implementation decisions, and an illustration of how data structures and application modules are mapped onto the implementation platform.

Function

The reference implementation here allows the application framework to achieve four objectives. First, it demonstrates that the application concurrency can be exposed while maintaining functional correctness. Second, it demonstrates that an efficient software architecture can be designed to exploit specific levels of application concurrency. Third, it demonstrates that an efficient software implementation can be constructed on a computing platform. Lastly, it demonstrates that the computing platform is capable of efficiently delivering the desired performance objectives.

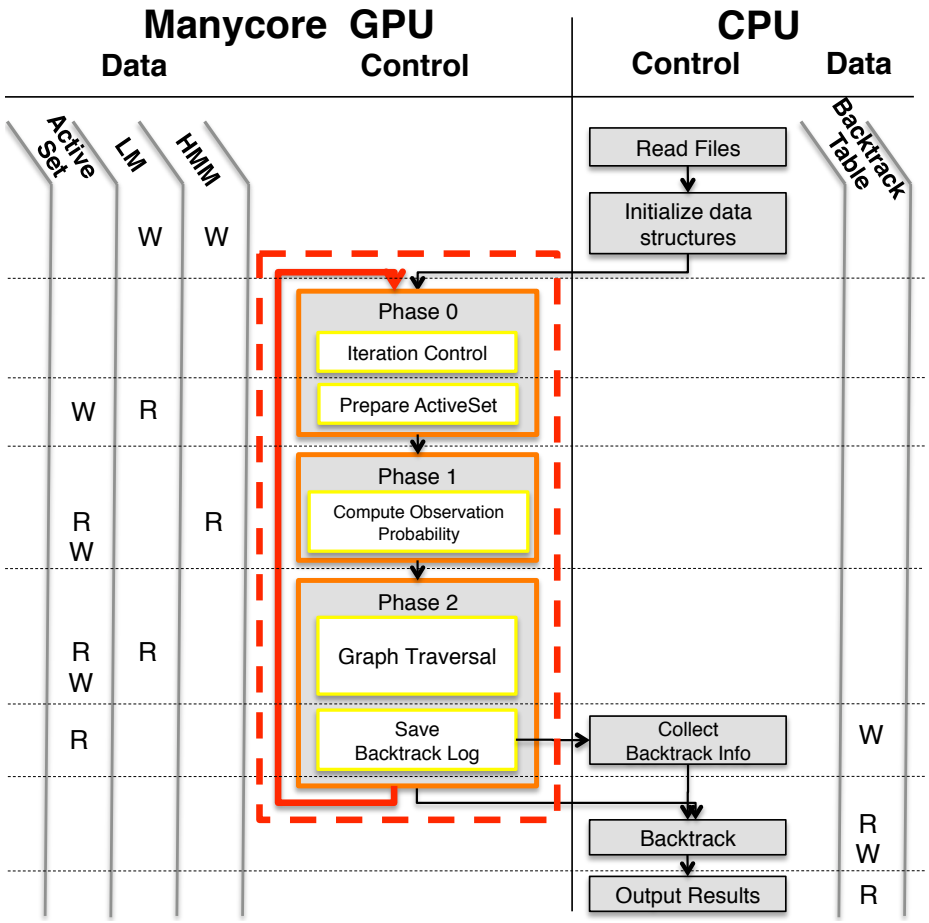


Figure 4.6: A summary of the data structure access and control flow of the inference engine on the manycore platform

Benefits

By providing the reference implementation, application domain experts can examine the specific application concurrency that is exploited and how it fits in an efficient software implementation. They can also experience the performance achievable on a hardware platform. Such information can allow domain experts to quickly evaluate the suitability of the application framework for their specific usage scenario. Moreover, all of this can be achieved without going through weeks of exercise learning about a hardware platform, and months of development and debugging on a proof-of-concept design.

Example

Figure 4.6 illustrates the reference implementation of the speech inference engine. As specified in the software architecture, the forward pass inference loop is implemented on the GPU accelerator, and the backward pass is implemented on the CPU.

The main data structures can be classified into two types: read-only model data structures and read/write runtime data structures. Referring to Figure 4.6, the read-only model data structures include the acoustic model parameters (shown as "HMM" for hidden Markov model) and the WFST graph structure (shown as "LM" for language model). Read/write runtime data structures include storage for intermediate buffers for the inference process (shown as "Active Set" for the set of most likely alternative interpretations actively tracked at runtime) and backtrack information (shown as "Backtrack Table"), which is a log of the forward pass of the Viterbi algorithm that is used for the backward pass.

In terms of the program flow, the models and inputs are read from several files and placed into the memory on the GPU. The forward pass of the Viterbi algorithm is computed on the GPU and the logs are sent back to the CPU. When the forward pass is complete, the CPU performs the backtrack operations from the overall most-likely path, and outputs the results to file.

On the GPU, the forward pass occurs in three phases. An additional Phase 0 is inserted to implement the performance optimization techniques. Phase 1 evaluates the observation probability, and Phase 2 tracks the most likely interpretations of the input utterance over time.

In Phase 0, the *Iteration Control* step is introduced so as to manage the memory pools that are allocated in the ActiveSet data structure. In highly parallel shared memory architectures, any memory allocation and freeing action will introduce a point of serialization in shared memory management. The technique to avoid serialization is that of allocating a memory pool at the beginning of a program, and carefully managing program behavior such that memory usage stays within the pre-allocated space. In speech recognition, we track a set of active states that represent the set of most likely alternative interpretations in each time step at run time. The decision of whether a state is active or not is based on a *pruning threshold*. States whose likelihood is greater than the threshold are tracked, while others are pruned away in Phase 2. The challenge is that at the beginning of a time-step in Phase 0, it is unclear what pruning threshold will allow the right amount of states to be active. The *Iteration Control* step makes a prediction based on the history of the threshold values and the number of active states in order to predict what threshold to set in the current time-step so as to keep the traversal process within the pre-allocated space.

During the *Prepare ActiveSet* step in Phase 0 we populate runtime data buffers to maximally regularize data accesses. The recognition network is irregular and the traversal through the network is guided by user input that is available only at run time. To maximize the utilization of the memory bandwidth, the data required in each iteration is gathered into consecutive vectors acting as runtime data buffers, such that the algorithmic steps in the iteration are able to load and store results one cache line at a time. This maximizes the utilization of the available memory bandwidth.

The *Compute Observation Probability* step in Phase 1 is a compute-intensive step. It involves matching the input waveform feature vector at the current time step to a large set of acoustic model parameters in the form of Gaussian mixture models (GMM). Depending on the types of features used and the corresponding acoustic model utilized, the computation performed at this step may be different across distinct usage scenarios.

Phase 2 is a performance critical phase. There are two steps in this phase, the *Graph Traversal* step and the *Save Backtrack Log* step. The *Graph Traversal* step involves data-parallel graph traversals of the irregular WFST graph structure, where the working set is guided by inputs known only at runtime. The graph traversal routines execute in parallel on different cores and frequently have to update the same memory locations. Such frequent write conflicts between cores must be resolved in an efficient manner. Four techniques were employed to optimize the graph traversal for speech inference on GPUs, and each of these is discussed in details in Chapter 5:

1. Constructing efficient dynamic vector data structures to handle irregular graph traversals.

2. Implementing an efficient find-unique function to eliminate redundant work by leveraging the GPU global memory write-conflict-resolution. policy

3. Implementing lock-free access to a shared-map leveraging advanced GPU atomic operations in order to enable conflict-free reduction.

4. Using hybrid local/global atomic operations and local buffers for the construction of a global queue to avoid sequential bottlenecks in accessing global queue control variables.

These techniques allow for the graph traversal step with *irregular* data access patterns and *irregular* task synchronization patterns to be implemented on a data parallel platform. This way the application will not be bottlenecked as a result of the sharing of intermediate data between the CPU and the GPU in the inner loop of the speech inference algorithm in a hybrid GPU-CPU implementation. The implementation of these techniques also includes basic capabilities for *introspections* on the dynamically changing data working set size that is induced by the input data. The framework is able to *automatically adapt* the routines' runtime parameters by adjusting their task distribution parameter (the thread block size) based on the amount of work available.

The *Save Backtrack Log* step in Phase 2 transfers traversal data from the GPU to the CPU. This step incurs a sequential overhead of 13% of the total execution time after parallelization.

4.4.4 Extension Points

In this section, we discuss the *extension points*, which is last of the four components of a *pattern-oriented application framework* by presenting its *features*, *functions*, and *benefits* to the application domain expert. We then use a component from an ASR application framework as an example.

Feature

An *extension point* is an interface for creating families of functions that extend the capability of the application framework for a specific application. It is implemented as an

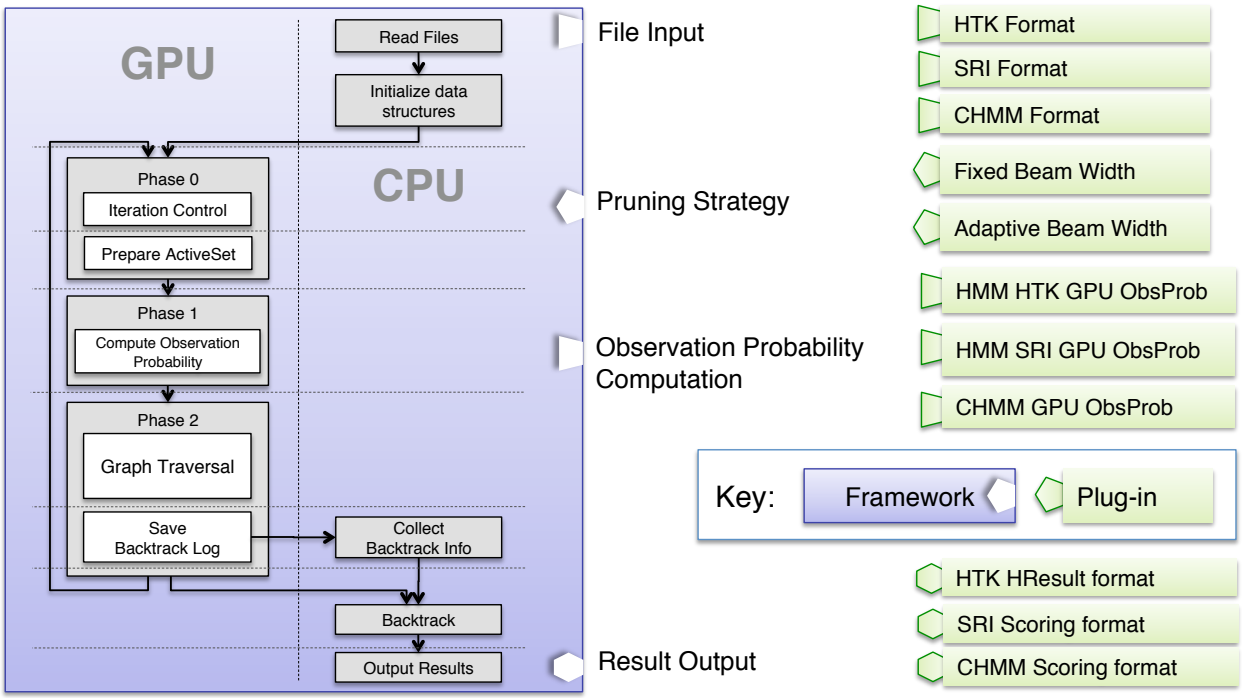


Figure 4.7: An application framework for automatic speech recognition with extension points

instance of an *Abstract Factory* creational object-oriented programming pattern, as specified in [68]. This pattern involves an abstract class definition that specifies a set of methods with names, as well as a list of parameters accompanying their data types.

Function

The *extension points* specify how a set of plug-ins should interact with the application framework. From the perspective of the framework, the extension points abstract the functions provided by the plug-in modules. From the perspective of the plug-ins, the extension points abstract the framework's requirements for the plug-in functions.

Benefits

For the application domain expert, the extension points provide an environment from which to develop new functions on an application framework. One does not need to rewrite any of the generic modules associated with the infrastructure of an application, and can leverage the expertise crystallized within the performance critical portions of the reference implementation. The extension points allow the application domain experts to extend the function of the framework without jeopardizing its execution efficiency.

```
1  string getName();
2  void   build_model(const Runopt * options);
3  void   save_model (const char *filename);
4  void   load_model (const Runopt * options);
5  void   free_model ();
6
7  void   observationProbComputation(
8         const int         frame,
9         const Utterance *in,
10        const int        *LabelFlagHash,
11              float      *LabelProbHash);
```

Figure 4.8: The observation probability computation extension point definition

Example

Based on optimized reference implementation, we can construct extension points to allow the implementation to be flexibly adapted to a variety of usage scenarios. The application framework with its extension points and many plug-ins is presented in Figure 4.7. The extension points are illustrated as notches in the application framework. Their position is aligned with the functions in the program flow that they affect. The shapes of the extension points, or the notches, correspond to the associated plug-ins that match them.

There are three extension points implemented in the ASR application framework: the Observation Probability Computation, the Pruning Strategy, and the Output File Format.

The **Observation Probability Computation extension point** is complex. It is used to manage not only the computation of the observation probability, but the data structure of the acoustic model associated with the computation as well. As different acoustic models may have different data representations, the data structure itself is not specified at the interface. This is also the reason why the File Input extension point is part of this interface. The interface is specified in the following manner:

In the extension point interface for observation probability computation shown in Figure 4.8, line 1 is a self-identifying function for the plug-in function. The *build_model* interface on Line 2 reads a text format input file into the internal data format. The filename to read is specified in the run options in the parameter. The *save_model* interface on Line 3 dumps the internal format to a binary file for fast loading in future runs. The *load_model* interface on Line 4 loads previously dumped internal format from a binary file. The *free_model* interface frees a model from the memory.

The *observationProbComputation* interface on line 7 is where Phase 1 of the inference engine takes place. It has four parameters. The *frame* parameter specifies what frame, or time step, the Phase 1 calculations target. The *in* parameter provides the input data. The *LabelFlagHash* parameter provides a vector of "0"s and non-"0"s as flags, where a "0" means the calculation should be skipped. The *LableProbHash* is a vector of output where for each non-"0" in the input flag vector, a valid observation probability is expected.

```
1 string getName();
2 void   iteration_control(
3        const int       frame,
4              iControl *history,
5              float     pruneThreshold);
```

Figure 4.9: The pruning strategy extension point definition

```
1 string getName();
2 void   Result_Output(
3              FILE        *outFile,
4        const gpu_History *hist,
5        const WordTable   *Wordtable,
6        const SegmentList *segmentList);
```

Figure 4.10: The result output extension point definition

The **Pruning Strategy extension point** allows the customization of the algorithm for specifying the pruning threshold to occur. The interface is as follows:

Figure 4.9 shows the In the extension point interface for the pruning strategy, line 1 is a self-identifying function for the plug-in function. The *iteration_control* interface has three parameters. The *frame* parameter specifies what frame, or time step, the plug-in is working on. The *history* parameter provides the history of the past active state count and pruning threshold used. The *pruneThreshold* is the predicted pruning threshold to be used for the current frame.

The **Result Output extension point** allows the results of the inference process to be displayed in any format. The interface is as follows:

In the extension point interface for result output shown in Figure 4.10, line 1 is a self-identifying function for the plug-in function. The *Result_Output* interface has four parameters. The *outFile* parameter indicates the file pointer if the results are to be written to a file. The *hist* parameter provides the results of the backward pass in the Viterbi algorithm. The *Wordtable* parameter provides the word table to look up word IDs and print them out as words. The *segmentList* parameter provides the list of filenames being analyzed.

4.5 Discussion

The pattern-oriented application framework is best suited for applications with a relatively mature software architecture, where extension points can be clearly defined [59], as the framework code must remain stable across many deployments to end-user usage scenarios for the cost of developing a framework to be effectively amortized. With this in mind, applications such as automatic speech recognition (ASR) are ideal candidates.

With respect to the "effectiveness" metrics specified in Section 3.1, the pattern-oriented

application framework, as discussed in this chapter, allows an application domain expert to construct an end-user application in a *productive*, *efficient* and *portable* manner.

Productivity

Productivity is achieved as the application framework allows an application domain expert to follow a fast path to a functionally correct solution that implements the desired feature or application that can meet the performance constraints. We qualitatively describe the productivity benefits here.

The *application context* component of the pattern-oriented application framework provides a more concise set of background information. The component quickly introduces an application domain expert to the sources of concurrency that are inherent to the application domain. This saves him/her a significant amount of time that would other wise be spent on the identification of sources of concurrency to exploit.

The *software architecture* component of the pattern-oriented application framework introduces an efficient software architecture for better implementation of the application, as well as the potential performance bottlenecks to avoid. This saves the time necessary to architect an application from scratch and arduously perform software architecture design space exploration to validate the software architecture.

Although some effort is required to understand an application framework well enough to use it, the pattern-oriented nature of the application framework provides a standard format to expose and exploit application concurrency, greatly reducing the burden to studying the application context and the software architecture to get oriented to the reference implementation and extension points.

The pattern oriented application framework includes a *reference implementation* with *extension points* that allow for the quick customization of the application framework in order to target specific usage scenarios. This reduces the *lines of code* (LOC) that an application domain expert has to write to the specific customizations of the extension points. As an example, this reduces the efforts of implementing a speech recognition application from a ten-thousand-line application to a few hundred lines of plug-in code.

The *application context* and the *software architecture* components of the pattern-oriented application framework carefully explains the sources of application concurrency and the trade-offs in exploiting these sources. This greatly reduces errors in an implementation resulting from any misunderstandings of the parallelization concept.

With significant reduction in the lines of code required to be implemented, the probability of errors from the inevitable coding process is greatly reduced.

Efficiency

Efficiency is achieved through the rigorous process of developing reference implementations that incorporates both application domain expertise as well as parallel programming expertise to take full advantage of the hardware resources. Section 5.1.4 demonstrates the achievable efficiency of a *reference implementation*' by presenting results that achieved

85-98% of peak execution efficiency on a GPU for the compute-intensive portion of an application. Section 6.1 demonstrates a 14-20x speed up on a GPU compared to the performance of the sequential implementation on a CPU.

Portability

Portability for a pattern-oriented application framework is achieved through two means: *functional portability* and *performance portability*.

With respect to *functional portability*, the narrow interfaces defined by the *extension points* naturally partition the application such that the plug-ins of the application framework can be used in other application frameworks. An extreme exemplar that illustrates this portability is that of the observation probability computation plug-in presented in Section 6.1.3. It was originally designed for audio-video speech recognition, and was recently used in another application implementing audio-video speech model training.

With respect to *performance portability*, the pattern-oriented design allows the implementation to achieve "Parallel scalability" [157], enabling an application to efficiently utilize an increasing number of processing elements on future generations of even more parallel platforms.

It is also well known that performance portability across architecturally different microarchitectures is a challenging research topic. As demonstrated in Section 5.1.3 and Section 5.1.5, application framework must be tuned on different microprocessors to achieve high efficiency. Such tuning process can be automated as described in [152]. The implementation of the extension point can utilize programming frameworks such as Copperhead [39] to allow parallel computation to be implemented on architecturally diverse manycore microprocessors.

4.6 Summary

Pattern-oriented application frameworks allow application domain experts to effectively leverage the capabilities of the highly parallel manycore microprocessors. The four main components of a pattern-oriented application framework are: *application context*, *software architecture*, *reference implementation*, and *extension points*. The feature, function, and the benefits of each component were discussed, and examples of each component from the automatic speech recognition application framework were provided.

The four components of the application framework, when used together, allow an application domain expert to specify the needs of the application usage scenario, to match them to an application framework, and to customize the associated reference implementation in order to construct the software for an end-user usage scenario – all without the presence of a parallel programming expert.

Pattern-oriented application frameworks can greatly reduce the barriers for application domain experts to more effectively utilize the manycore microprocessors, encouraging wider deployment of highly parallel computing platforms.

Chapter 5

The Construction of Pattern-Oriented Application Frameworks

When an application developer is faced with developing a parallel application, he/she should look for a pattern-oriented application framework for parallel programming to assist in the application development process. When an appropriate application framework exists, the application developer can customize it and apply it to the usage scenario at hand. However, when no appropriate application framework is available, how can he/she construct an efficient implementation, and re-factor it into an application framework for future use?

Constructing a pattern-oriented application framework for parallel programming requires a four-step process, as illustrated in Figure 5.1. The first three steps: specify, architect, and implement, were addressed in Section 2.3. Once a functional and efficient reference implementation is available, one can *leverage* it to create the application framework. The *leverage* process includes re-factoring the implementation, creating extension points, and developing plug-in modules for the extension points.

This chapter addresses the two main concerns in the development of application frameworks: *efficiency* and *productivity*. Since the primary motivation for application developers to take up the challenge of using parallel platforms is the potential performance gains, the primary concern to address during the construction of a pattern-oriented application framework is *execution efficiency*. Once an efficient implementation exists, we can go on to explore the construction of the extension points in order to allow for better *developer productivity*.

This chapter presents the construction process of three pattern-oriented application frameworks in the respective fields of machine learning and computational finance.

1. The Automatic Speech Recognition (ASR) application framework provides speech recognition application experts with an infrastructure to quickly deploy speech recognition application under different usage scenarios. An efficient ASR application has been completely implemented and was re-factored to enable multiple extension points to be constructed. The completed application framework has been deployed in two usage scenarios, which are further described in Chapter 6.

2. The Value-at-Risk (VaR) application framework is being developed to allow risk managers in banks and hedge funds to quickly deploy VaR engines for their specific usage

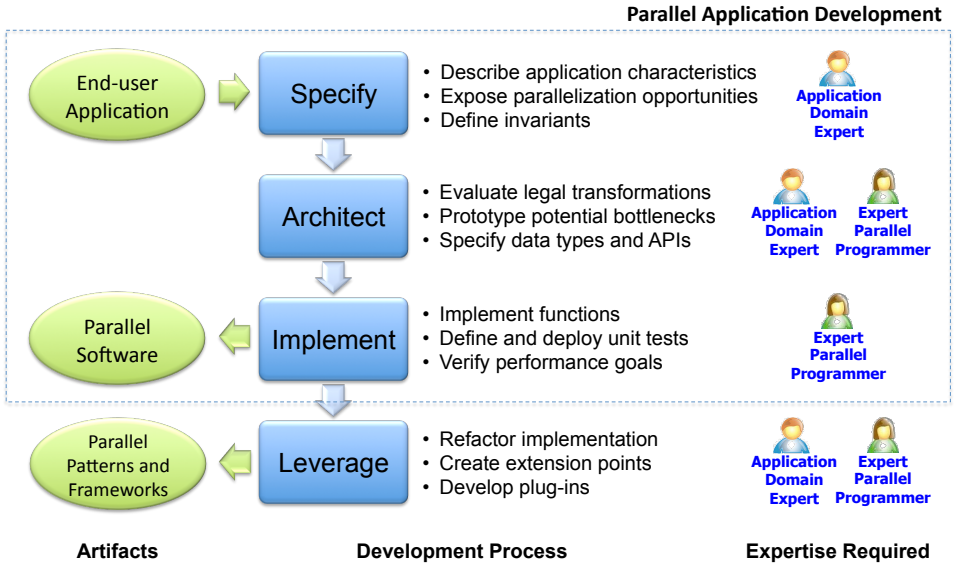


Figure 5.1: The construction of pattern-oriented application frameworks using the *Leverage* step.

scenarios. The configurations needed to achieve efficient execution are reported, and an efficient reference implementation has been developed. Research with respect to the re-factoring and the development of the extension points in the reference implementation is still on-going.

3. The Potential Future Exposure (PFE) application framework is being developed to allow an industrial-scale counterparty exposure estimation application (as described in Section 5.2.2) to be effectively extended and maintained. The initial design space has been explored and the full reference implementation is still being constructed.

5.1 Automatic Speech Recognition Application Framework

Automatic speech recognition (ASR) allows multimedia content to be transcribed from acoustic waveforms to word sequences. This technology is emerging as a critical component in data analytics for the wealth of media data that is being generated everyday. Commercial usage scenarios have already begun to appear in industries, such as customer service call centers for data analytics, where ASR is used to search recorded content, track service quality, and to provide early detection of service issues. Fast and efficient ASR enables economic employment of text-based data analytics to multimedia contents to occur. This opens the door to an unlimited array of potential applications, such as automatic meeting diarization, news broadcast transcription, and voice-activated multimedia systems in home entertainment systems.

The goal of an automatic speech recognition application is to analyze a human utterance from a sequence of input audio waveforms in order to interpret and distinguish the words and sentences intended by the speaker. The mathematics behind statistical inference for probabilistic functions of finite state Markov chains was developed by Baum in the late 1960s [24]. The concept was used in a speech recognition system in 1975 by Baker [22], as well as in work conducted at IBM by Jelinek *et al.* [87]. The concept of beam search, which is a heuristic used to "focus" computation on the most-likely portion of a search space, was used by Lowerre in 1976 [106]. In the 1980s, HMM-based approaches were investigated by labs at BBN [133] and Philips [32], as mentioned in [74]. By the mid-late 1980s, HMMs became the dominant recognition paradigm, adopted by systems at SRI [116], MIT-Lincoln [126], and Carnegie Mellon University (CMU). The CMU system developed by Lee in 1988 [103] is representative of the types of systems developed at the time.

An HMM-based approach has since become the *de facto* approach used in major speech recognition projects such as SPHINX, HTK, and Julius [103, 154, 102]. This approach gained further popularity after the Rabiner published his extensive survey paper [129] on HMM for speech recognition. Figure 5.2 shows the major components of such a system. This system uses a recognition network that is compiled offline from a variety of knowledge sources using powerful statistical learning techniques. Spectral-based speech features are extracted by processing the input audio signal; the inference engine then computes the most likely word sequence based on the extracted speech features and the recognition network.

Inference engine based LVCSR systems are modular and flexible. They are language independent and robust to various acoustic environments [154, 102]: by using different recognition networks and signal-processing kernels, they have been shown to be effective for Arabic, English, Japanese, and Mandarin, in a variety of situations, such as phone conversations, lectures, and news broadcasts.

There is a large body of prior work in mapping LVCSR to parallel platforms, which can be clustered into three categories of efforts:

- *Data Parallel, shared memory implementations on multiprocessor clusters*: These implementations are plagued by high communication overhead in the platform, high sequential overhead in the software architecture, load imbalance among parallel tasks or excessive memory bandwidth. They are limited in scalability as computing platforms become more parallel.

  In 1993, Ravishankar [130] presented a parallel implementation of the HMM-based inference engine with a beam search algorithm on a shared memory PLUS multiprocessor cluster that was developed at Carnegie Mellon University. The implementation statically mapped a carefully partitioned recognition network onto the multiprocessors in order to minimize load imbalance. While achieving a 3.83x speedup over the sequential implementation on five processors, static partitioning approaches will not scale well to tens of cores due to load imbalance among the partitions at run time.

  The parallel LVCSR system proposed by Phillips *et al.* [128] achieved a 4.6x to 6.23x speedup on 16 processors. However, this implementation was limited by the sequential components in the recognizer and load imbalance among the processors. Their private

buffer-based synchronization imposes significant data structure overhead and is not scalable with the increasing number of cores.

You *et al.* [158] have recently proposed a parallel LVCSR implementation on a commodity multicore system using OpenMP. The Viterbi search was parallelized by statically partitioning a tree-lexical search network across cores. However, due to limited memory bandwidth, only 2x speedup was achieved on shared-memory Intel Core2 quadcore processors.

- *Task parallel implementation*: the scaling of these implementations to more parallel platforms requires extensive redesign efforts.

  Ishikawa *et al.* [85] explored coarse-grained concurrency in LVCSR and implemented a pipeline of tasks on a cell phone-oriented multicore architecture. They achieved 2.63x speedup over a sequential baseline version by distributing tasks among three ARM cores. However, due to the small amount of function-level concurrency in the algorithm, it is difficult for this implementation to scale beyond three cores.

- *Data Parallel implementation with manycore-based accelerators in CPU-based host systems*:

  Prior works such as [62, 38, 79] by Dixon *et al.*, Cardinal *et al.*, and Gupta *et al.* leveraged manycore processors and focused on speeding up the compute-intensive phase (i.e., the observation probability computation) of LVCSR on manycore accelerators. Compared to CPU-based implementations, both [62] and [38] have shown approximately 5x speedups in the compute-intensive phase. Both [62] and [38] have also mapped the communication intensive phases (i.e., Viterbi search) onto the host CPU processor. This software architecture incurs significant penalty for copying intermediate results between the host and the accelerator subsystem and does not expose the maximum potential of the performance capabilities of the platform.

The automatic speech recognition inference engine presented in this thesis is a collection of multiple years of efforts in constructing an efficient and scalable application that is fully implemented on the GPU. The following sections illustrate in a host of optimizations that can be encapsulated in a pattern-oriented application framework for domain experts. Specifically:

- Sections 5.1.1 and 5.1.2: Apply parallel programming expertise to expose and exploit application concurrency to more efficiently implement ASR
- Section 5.1.3: Applies application domain expertise to explore multiple recognition network representations in order to more efficiently implement ASR
- Section 5.1.4: Applies application domain expertise to transform input recognition network structure so as to more efficiently implement ASR
- Section 5.1.5: Applies parallel programming expertise in order to experiment with multiple HW implementation platforms to construct more efficient implementations of ASR

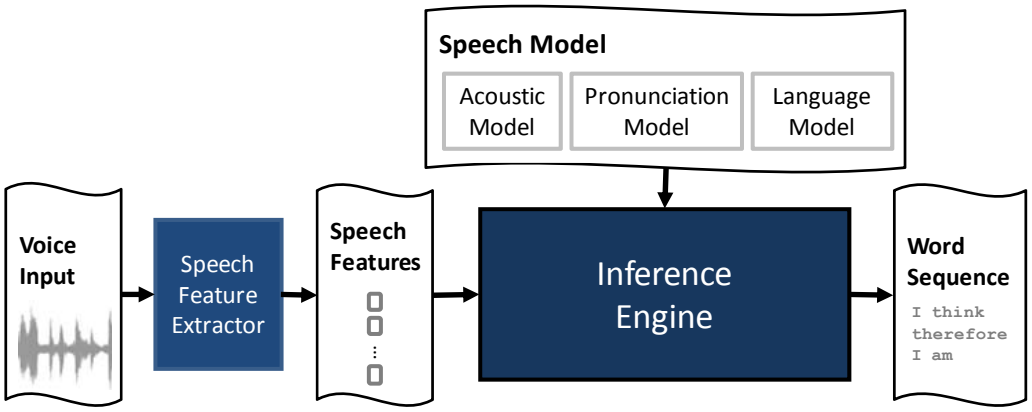


Figure 5.2: The system architecture of a large vocabulary continuous speech recognition application.

5.1.1 Implementation Efficiency Concerns

The recognition of human speech is a complex task, especially when you consider the significant variations in the voice quality, speed, and pronunciation among different speakers. Moreover, differences among languages and the speech recording environments pose further challenges to an effective recognition system. After decades of scientific research, many researchers have converged on the hidden Markov model (HMM) extract and inference system as a standard setup. In this setup, the acoustic signal is treated as the observed sequence of events and the sentences to be recognized are considered the hidden cause of the acoustic signal.

Figure 5.2 illustrates that ASR first extracts the representative features from an input waveform and then decodes the feature sequence to produce a word sequence. The feature extraction process involves a sequence of signal processing steps. It is done to minimize the influence of non-linguistic factors in the speech signal and preserving factors that are most useful in distinguishing word sequences. A feature vector is extracted per 10ms segment of the waveform (a time step). A sequence of feature vectors is used in an inference process by iteratively comparing each feature vector to a probabilistic speech model. The speech model contains an acoustic component, a pronunciation component, and a language component. The pronunciation component comes from a typical dictionary. Both the acoustic and language components are trained off-line using a set of powerful statistical learning techniques.

The acoustic model is often represented as a multi-component Gaussian mixture model, which is a more general model that uses a family of distributions to represent the feature space of each phone label[1]. The pronunciation model describes each word as a sequence of phones that make up its pronunciation. The language model relates words to phrases and is

[1]Phone: An abstract unit of sound

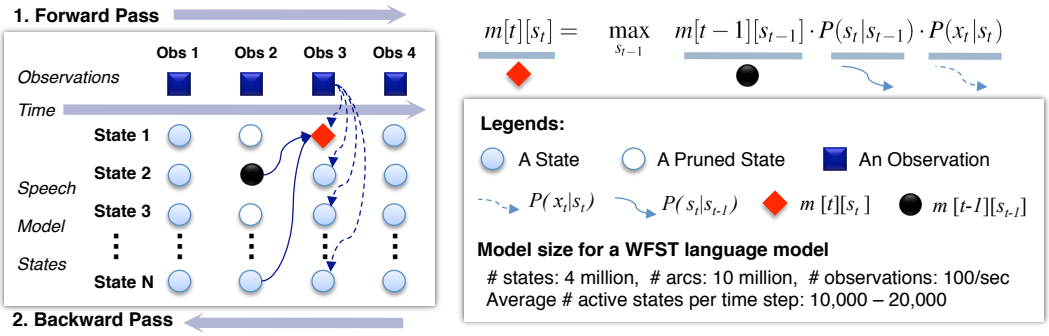


Figure 5.3: Application characteristics: the inner-workings of the performance critical Viterbi forward and backward pass steps.

often represented by unigrams, bigrams, and trigrams, which are the common one/two/three word phrases and their likelihood of appearance in a language. To recognize a different language in a distinct environment, only the speech model and the feature extractor need to be changed. The inference engine remains unchanged producing the most-likely interpretation of the input sequence by traversing the speech model for the specified language.

The pronunciation model and language model can be seen as finite state machines with transition probabilities on the arcs between states. One of the state-of-the-art optimization techniques in automatic speech recognition is the composition of the acoustic model with the language model off-line in order to produce a weighted finite state transducer (WFST) [112], where a single state machine can relate phone sequences to phrases directly. A typical WFST-based recognition network can have millions of states and arcs. The WFST representation could reduce redundancies in the number of state transitions traversal at run time by 22x [44], and simplify the traversal process to reference only one state machine.

Figure 5.3 shows the inference process of ASR. The inference engine uses the Viterbi dynamic programming algorithm [121]. It performs a forward-pass to track the more likely interpretations of the feature vector sequence through each increment in the time steps, and a backward-pass step to backtrack the path taken by the most likely outcome at the end of a sequence.

The forward-pass has two main phases in the inference process. Figure 5.3 shows Phase 1 as dashed arrows between observations and states. This phase evaluates the observation probability of the hidden state, matches the input information to the available acoustic model elements, and only takes into account the instantaneous likelihood of a feature matching acoustic model element. Figure 5.3 shows Phase 2 as the solid arrows between the states of consecutive time steps. Phase 2 references the historic information in regards to what the most likely alternative interpretations of the utterance heard so far are. It also computes the likelihood of incorporating the current observations given the pronunciation and language models. The computation for each state st at time t (with the diamond shaped state as an example) records the state transition from the prior time step $t-1$ that produced the maximum probability $m[t][st]$.

In each time step of the algorithm, the data working set involves thousands of active states that are used to keep track of the most likely alternative interpretations of the speech that has been heard so far. We compute the next set of active states by using Phase 2 of the algorithm, as described above. It is found that by tracking less than 1% of the most likely alternative interpretations and pruning away the rest, negligible differences in recognition accuracy are perceived as compared to tracking all alternative interpretations. The process of pruning less likely interpretations is called the beam search technique as described by [121], is called the beam search technique.

The recent development of the weight-finite state transducer (WFST) [112] has had a further standardization effect in the inference engine. In the traditional HMM-based LVCSR inference engine, the recognition network has a regular structure where each word is translated into a sequence of phone-states to be recognized, i.e. a linear-lexicon network. Many words that have the same prefix have phone states that can be shared, leading to a tree-based structure for the recognition network, or a tree-lexicon network. The tree-lexicon network has a greatly reduced memory footprint and saves computation time when the connectivity between words is not considered. Yet, as word-to-word bigram probabilities are used, a significant number of runtime state needs to be preserved with a tree replication process, making the decoder cumbersome to design. Mohri's work on WFST provides a LVCSR recognition network with features that are pre-compiled into the recognition network. Many of the recognition network optimization techniques are applied during network composition, and the decoding process has a clean, and well-defined behavior.

Application Concurrency

As mentioned in the sample application context for speech recognition in Section 4.4.1, there are four main levels of concurrency in the inference process that can be exposed by examining the application's characteristics. These levels of concurrency are exposed with parallel programming patterns, where the patterns are used to illuminate potential performance bottlenecks that may be encountered while exploiting these parallelization opportunities.

1. **Concurrency among speech utterances:**

   Looking at the process of extracting acoustic features from the input waveform and conducting the inference on the feature stream with the speech inference engine, the input waveform can be split up into numerous segments of speech that can be independently recognized[2]. A typical conversational utterance is usually no longer than 5 to 30 seconds long and is separated by short silences. A one-hour long speech recording can be quickly processed to detect pauses in the speech and can be split up into hundreds of utterances to be recognized in parallel. This can be done utilizing the MapReduce pattern, where the Map function maps the independent utterances to be processed in

[2]One may argue that the topic of the words in one part of the input speech may help with recognition of other parts of the input speech. While it is true that the knowledge of context may help improve the accuracy of the speech, how context is best taken into account in speech recognition remains a topic of research

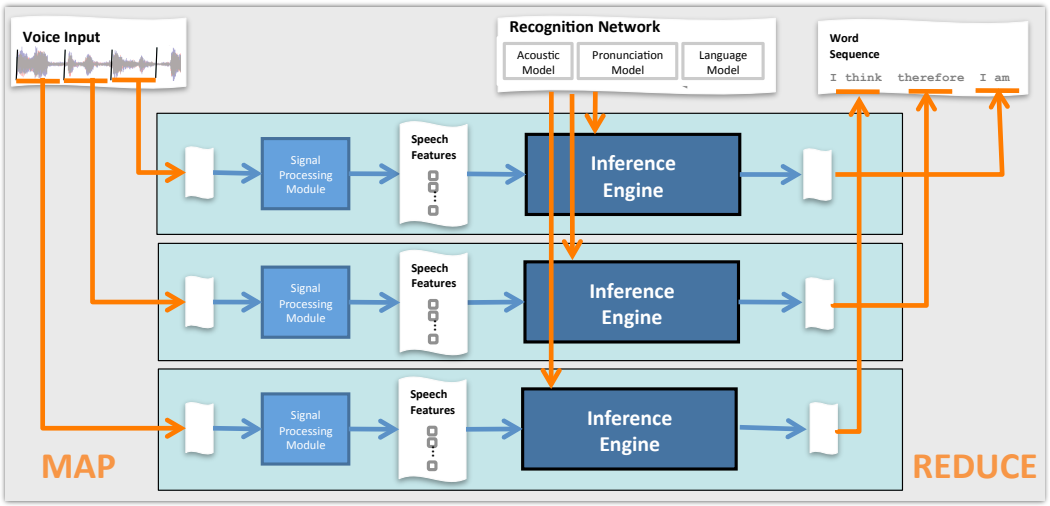


Figure 5.4: Parallelization Opportunity 1: Applying MapReduce parallel programming pattern over the input speech utterances.

parallel. The Reduce function then merges the resulting words according to their time stamps to produce a full transcription of the recording. Figure 5.4 illustrates the Map and Reduce operations.

Figure 5.4 illustrates the concurrency over multiple input speech utterances. While this level of concurrency can be exploited over multiple processors where it can be complementary to the exploitation of more fine-grained concurrency within the detailed algorithm steps, it is not suitable for exploitation within vector lanes on a manycore processor. The Map process in the MapReduce parallel programming pattern duplicates the data working set of the map process. While the read-only operands can be shared, the data working set for an inference process contains tens of megabytes of variables. Duplicating the working set across a large number of mapped processes on the same manycore processor with a shared memory subsystem will create severe memory bandwidth contention, leading to a performance bottleneck at the memory interface. Furthermore, this level of concurrency can not be exploited for latency sensitive applications such as single-user on-line applications,.

2. **Concurrency among the forward and backward passes:**

   The forward and backward pass steps in the Viterbi algorithm can be seen as an instance of the Pipe-and-Filter pattern, where the forward pass and backward pass are filters, and the speech data passes through them and get transcribed into words. Applications that match the Pipe-and-Filter pattern can be pipelined, as independent data flows through the application. In the first of the four application concurrency opportunities, it was explained how a long speech recording can be segmented into short speech utterances. Figure 5.5 illustrates the parallelization opportunity exists to process forward and backward pass using a pipelined approach.

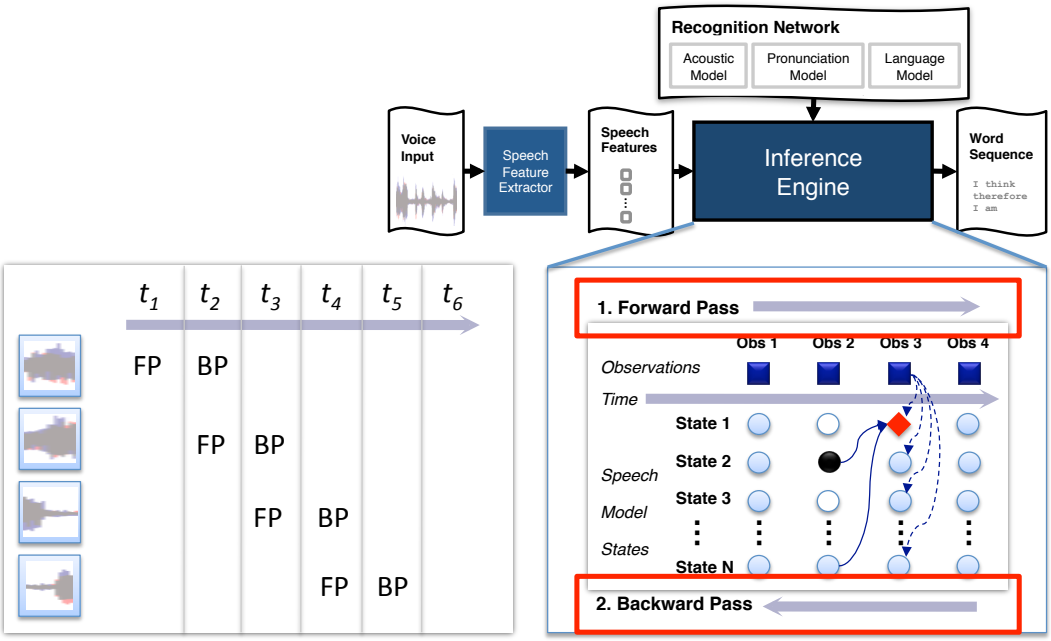


Figure 5.5: Parallelization Opportunity 2: Applying the Pipe-and-Filter parallel programming pattern over a sequence of input speech utterances.

Figure 5.5 shows the progression of time across the page, and the independent speech utterances going down the page. The independent speech utterances can be pipelined, i.e. the backward pass of one utterance can be computed at the same time as the forward pass of the next utterance.

In order to exploit this level of concurrency with the pipelining technique in the Pipe-and-filter parallel programming pattern, the pipeline stages must be balanced. In the Viterbi algorithm used in speech recognition, the backward pass performs less than 1% of the work done by the forward pass. Thus, the source of concurrency is not suitable for exploitation.

3. **Parallelism among the computationally intensive and communication intensive phases:**

   The Viterbi algorithm iterates through the input one time step at a time. Within each iteration, it performs two phases of computation. Phase 1 compares the input observation at the current time step with the acoustic models in the recognition network. This provides an instantaneous match between the input and the acoustic model, and involves the *computationally intensive* Gaussian mixture model (GMM) based computation. Phase 2 then takes the historical information into account by considering the accumulated likelihood from the previous time steps along with the state-to-state transition probabilities from the pronunciation and language models. Looking up the previous likelihood and state-to-state transition likelihood is highly *communication*

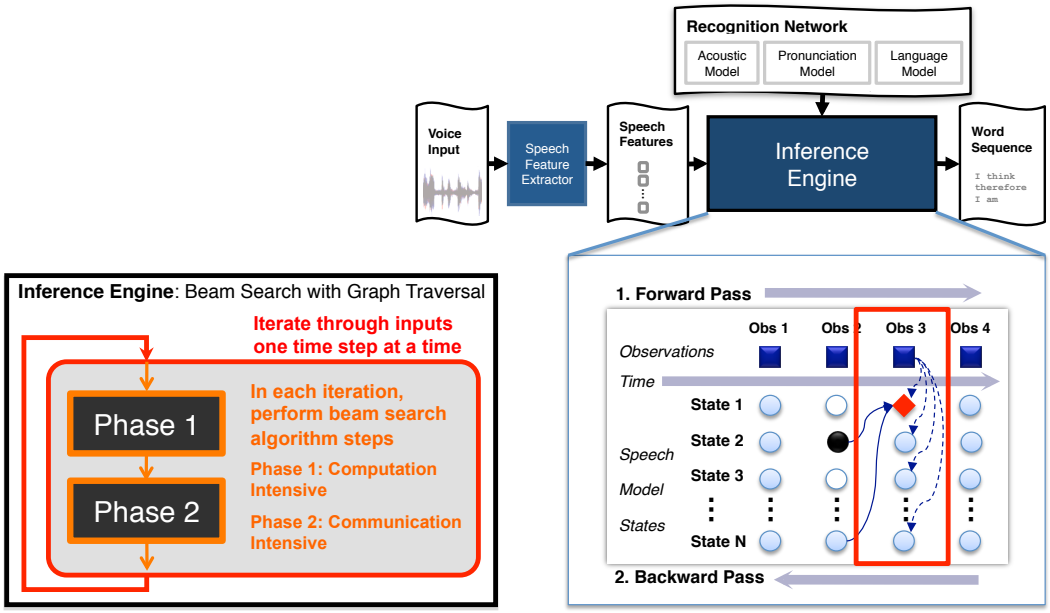


Figure 5.6: Parallelization Opportunity 3: Applying the Pipe-and-Filter parallel programming pattern over two phases of execution.

intensive.

The two phases can be seen as a Pipe-and-Filter parallel programming pattern, where the phases are the "filters", and the data passes through the "pipes" between the "filters". However, the two phases are encapsulated within an iterative-loop, with sequential dependencies between loop iterations. Such dependencies can be removed by the technique of speculation, where Phase 1 can speculatively process more work than necessary in order to start the work load of the next iteration before the current iteration has completed processing. Without these sequential dependencies between the loop iterations, the two phases may be parallelized through the pipelining approach across loop iterations.

To exploit this level of concurrency with the pipelining technique in the Pipe-and-filter parallel programming pattern, the pipeline stages must be balanced. While Phase 1 can take 80-90% of the total sequential execution time, when it is parallelized on the GPU, its runtime can be comparable with Phase 2's execution time running on the CPU. One issue with respect to partitioning Phase 1 and Phase 2 across the CPU and the GPU is that there are thousands of operands that must be passed from Phase 1 to Phase 2 for each time step. Communication between the GPU and CPU can become a performance bottleneck.

Figure 5.7 illustrates the alternative approaches for the implementation of phases 1 and 2 on a CPU-GPU hybrid system. Approach 1 was implemented by Cardinal *et al.* in [38], where Phase 1 was mapped to the GPU and Phase 2 was mapped to the CPU.

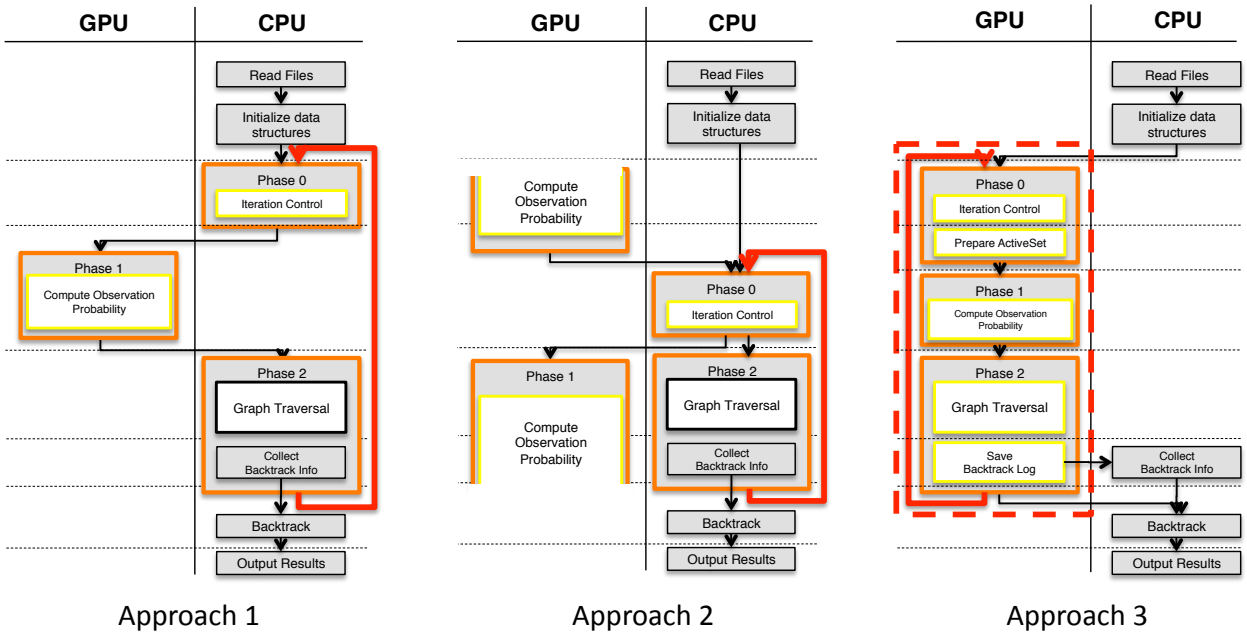


Figure 5.7: Alternative approaches for the implementation of Phase 1 and Phase 2 on a CPU-GPU hybrid system.

No pipelining was implemented. With the overhead coming from the transferring of operands and results back and forth, Phase 1 got a 5x speedup, and the application got only 33% speed up. Approach 2 was implemented by Dixon *et al.* in [62], where the dependency between Phase 1 and Phase 2 was removed by speculatively computing more results than necessary. Phase 1 was mapped to the GPU and Phase 2 stayed on the CPU. When the two phases were pipelined, Phase 1 got a speedup of 2-4x, and the application got a 60-90% speed up. Approach 3 was implemented using the techniques described in this thesis, where both phases 1 and 2 of the inner loop got mapped to the GPU. By eliminating the operand transfer overhead, Phase 1 got a 16-19x speed up and Phase 2 received a 3-4x speed up. Overall, more than a 10x speed up was achieved for the application compared to an optimized sequential implementation on the CPU.

4. **Parallelism among the thousands of observation probabilities and tens of thousands of alternative interpretations of the utterance being tracked**:

   Within each function in Phase 1, thousands of observation probabilities are being computed at each time step; within each function in Phase 2, tens of thousands of alternative interpretations of the input utterance are tracked. This is the ideal amount of parallelism that can be effectively utilized on a manycore computing platform.

   In this case, the MapReduce parallel programming pattern can be used in this case. Each observation probability computation can be mapped to an independent thread

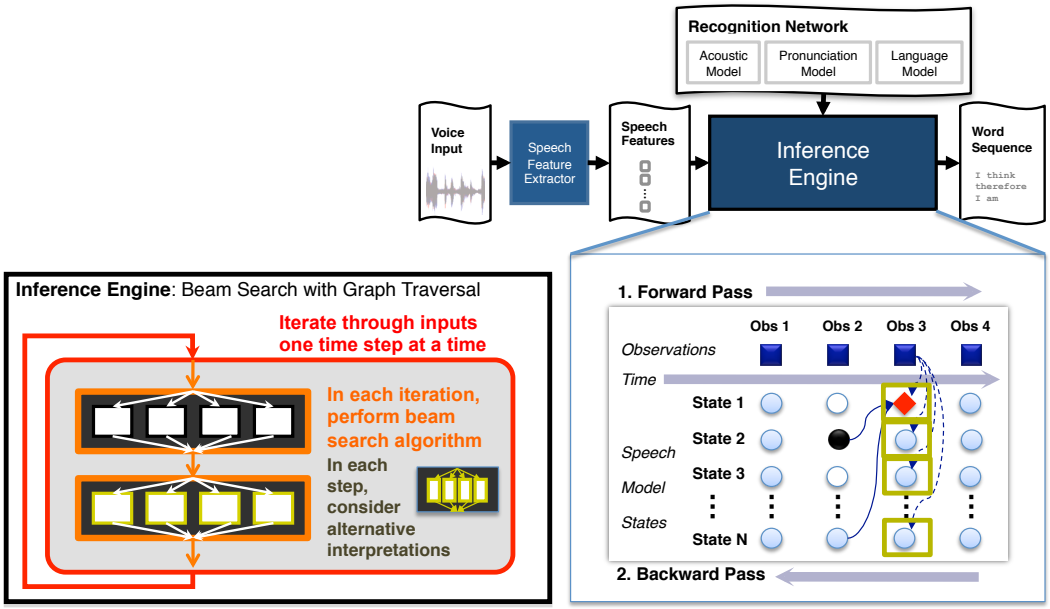


Figure 5.8: Parallelization Opportunity 4: Applying the MapReduce pattern over the functions that implement Phase 1 and Phase 2 in the Viterbi algorithm.

for computation, and the results can be reduced into a result array. Each alternative interpretation can be mapped to a thread responsible for tracking its likelihood over a time step, and the results are reduced to the destination of the tracking process.

While this software architecture maps well on to the many computing resources on the manycore GPU devices, it also presents significant challenges in regards to the implementation of all algorithm steps with data-parallel functions. More specifically, there are four main challenges that must be resolved with the following solution techniques:

(a) **Challenge:** *Handling irregular graph structures* with data parallel operations
Solution: *Constructing efficient dynamic vector data structure* to handle irregular graph traversals

(b) **Challenge:** *Eliminating redundant work* that occurs when threads are accessing an unpredictable subset of data based on input
Solution: *Implementing efficient find-unique function* by leveraging the GPU global memory write-conflict-resolution policy

(c) **Challenge:** *Performing Conflict-free reductions* in graph traversal in order to implement the Viterbi beam search algorithm
Solution: *Implementing lock-free accesses* of a shared map leveraging advanced GPU atomic operations with arithmetic operations so as to enable conflict-free reduction

(d) **Challenge:** *Parallel construction of a global queue* causes sequential bottlenecks when atomically accessing queue control variables

Solution: *Using hybrid local/global atomic operations and local buffers* for the construction of a global queue to avoid sequential bottlenecks in accessing global queue control variables

5.1.2 Detailed Efficiency Optimizations

1. *Constructing an efficient dynamic vector data structure to handle irregular graph traversals*

   There exist extensive concurrency in the automatic speech recognition application. As shown in Figure 4.5, there are 1,000s to 10,000s concurrent tasks that can be executed at the same time in both Phase 1 and Phase 2 of the recognition engine.

   In a sequential implementation of the speech inference process, more than 80% of execution time is spent in Phase 1. The computation in Phase 1 is a *Gaussian mixture model* evaluation to determine the likelihood of an input feature matching specific acoustic symbols in the speech model. The structure of the computation resembles a vector dot product evaluation, where careful structuring of the operand data structures can achieve 16-20x speedup on the GPU. With Phase 1 accelerated on the GPU, Phase 2 dominates the execution time. Phase 2 performs a complex graph traversal over a large irregular graph with millions of states and arcs. The data working set is too large to be cached and is determined by input available only at run time.

   Implementing Phase 2 on the GPU is challenging [107], as operations on the GPU are most efficient when performed over densely packed vectors of data. Accessing data elements from irregular data structures can cause an order of magnitude performance degradation. For this reason, many have attempted to use the CPU for this phase of the algorithm with limited success [62, 38]. The sub-optimal performance is mainly caused by the significant overhead of communicating intermediate results between phases 1 and 2.

   One can implement both phases of the inference engine on the GPU. This is achieved by using a technique for dynamically constructing efficient vector data structures at run time. As illustrated in Figure 5.9, this technique allows the intermediate results to be efficiently handled within the GPU subsystem and eliminates unnecessary data transfers between the CPU and the GPU.

   Problem:

   One often has to operate on large, irregular graph structures with unpredictable data access patterns in machine learning algorithms. How does one implement the data access patterns efficiently on the GPU?

   Solution:

   The solution involves dynamically constructing an efficient data structure at run time. Graph algorithms often operate on a subset of states at a time. Let this working set be

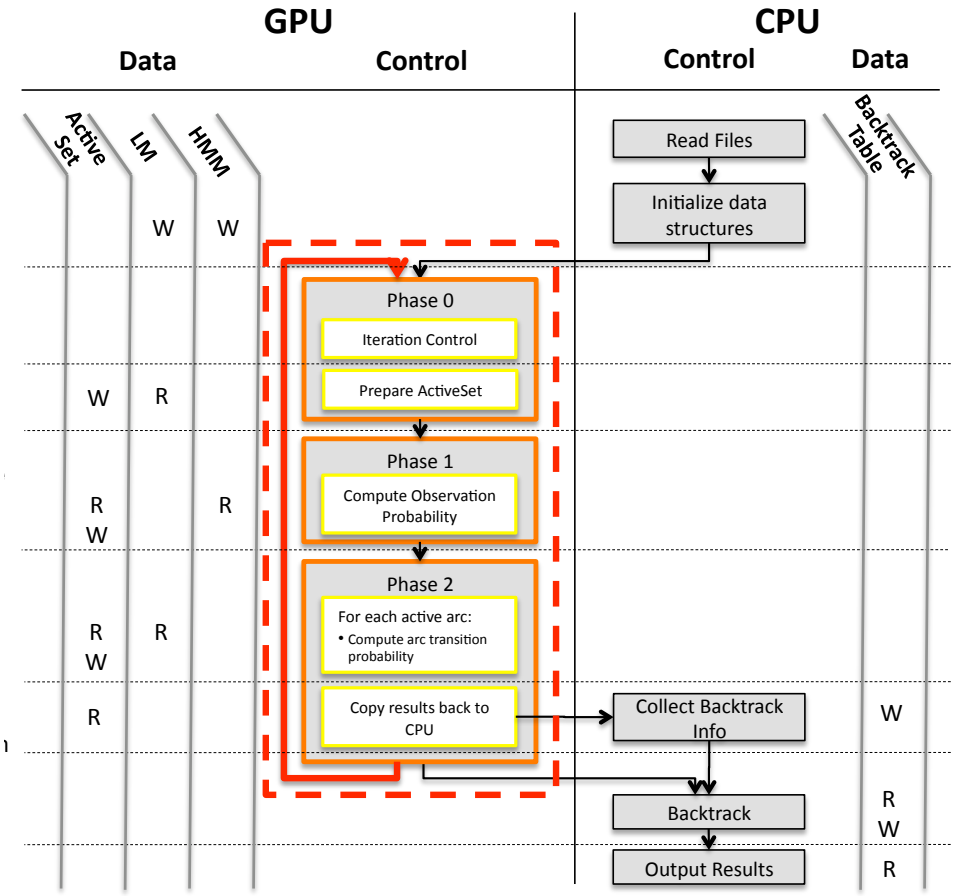


Figure 5.9: Summary of the data structure access and control flow of the inference engine on the manycore platform

the "active set". This active set information is often accessed multiple times within a sequence of algorithmic steps. This is especially true for information about the structure of the graph, such as the set of out-going arcs emerging from the active states.

Active set information can be dynamically extracted from the overall graph and stored in a contiguous vector for efficient access. The cost of gathering the information is then amortized over multiple uses of the same piece of information.

In automatic speech recognition, this technique is used during aggregation of state and arc properties. As shown in Figure 5.9, the "Prepare ActiveSet" step gathers information from the speech model and dynamically populates a set of vectors summarizing the data working set. This way, the following operations can execute on packed vectors of active states for the rest of the iteration.

The following code segment illustrates the usage of a dynamic runtime buffer:

```
1 // Gathering operands
2 int curr_stateID = stateID[i];
3 state_property0[i] = SpeechModel_state_property0[curr_stateID];
4 state_property1[i] = SpeechModel_state_property1[curr_stateID];
5 ...
6 // Utilizing gathered operands
7 stepA(stateID, state_property0);
8 stepB(stateID, state_property0, state_property1);
```

Figure 5.10: A demonstration of data-parallel data gathering routines

In Figure 5.10, there is a *stateID* array that stores a list of the active states. On lines 3 and 4, the properties of an active state i are gathered from the graph data structure *SpeechModel_state_property*0 and *SpeechModel_state_property*1 into a consecutive vector of active state properties in the *state_property*0 and *state_property*1 arrays. In subsequent algorithm steps on lines 7 and 8, one can use the dynamically constructed data structure reusing the efficient data layout.

2. *Implementing efficient find-unique function to eliminate redundant work*

   In Phase 1 of the inference process, the algorithm computes the likelihood of a match between an input feature vector and elements in a database of known acoustic model units called "triphone states". In the speech model we used, there is a total of 51,604 unique triphone states making up the acoustic model. On average, only 20% of these triphone states are used for the observation probability computation in one time step.

   In a sequential implementation, one can use memoization to avoid redundant work. As the active states are evaluated in Phase 1, one can check if a triphone state likelihood has already been computed in the current iteration. If so, the existing result will be used. Otherwise, the triphone state likelihood is computed and the result recorded for future reference.

   In the parallel implementation, active states are evaluated in parallel and implementing memoization would require extensive synchronization among the parallel tasks. Table 5.1 shows three approaches that could be used to compute the observation probability in each time step. One approach is to compute the triphone states as they are encountered in the graph traversal process without memoization. This results in a 13× duplication in the amount of computation required. Even on a GTX280 platform, computing one second of speech would require 1.9 seconds of compute time. A slight improvement to this approach is to compute the observation probability for all triphone states, including the ones that may not be encountered in any particular time step. This results in 4.9× redundancy in the necessary computation, which in turn results in 0.709 second of computation for Phase 1 for each second of speech.

   The more efficient approach is to remove duplicates in the list of encountered triphone states and compute the observation probabilities for only the unique triphone states

encountered during the traversal. This approach requires a fast data parallel find-unique function.

Table 5.1: Comparison of three approaches to determine observation probabilities to compute

Approaches	Real Time Factor*
1. Compute only encountered triphone with duplication and no memoization	1.900
2. Compute all triphone states once	0.709
3. Compute only encountered unique triphone states	0.146

* Real Time Factor:The number of seconds required to compute for one second worth of input data

Problem:

How does one efficiently find a set of unique elements among tens of thousands of possible elements to eliminate redundant work on the GPU?

Solution:

Traditionally, the find-unique function involves a process of "List sorting" and "duplication-removal". As shown in the left half of Figure 5.11, the "sort" step clusters identical elements together, and the "duplication-removal" process keeps one unique element among consecutive identical elements. On a data-parallel platform, Figure 5.11 illustrates further how the "duplication-removal" process expands into *cluster-boundary identification*, *unique-index prefix-scan*, and *unique list gathering* steps. Specifically, the *cluster-boundary identification* step flags a "1" for the elements that are different than their prior neighbor in the sorted list, and leaves all other positions as "0". The *unique-index prefix-scan* step computes the index of the cluster-boundary elements in the shortened unique list. The *unique list gathering* step transfers the flagged cluster boundary elements to the unique list. In this sequence of steps, sort is the dominant step with 89% of the execution time of the find-unique function. The sort step alone would take 0.310 seconds to process every second of input data.

An alternative approach is to generate a flag array based on the total number of unique triphone states in a speech model. This is illustrated in the right half of Figure 5.11. For the "Hash insertion" process, we leverage the semantics of conflicting writes for non-atomic memory accesses. According to the CUDA Programming Guide [4], at least one conflicting write to a device memory location is guaranteed to succeed. For the purpose of setting flags, the success of any one thread in write conflict situations can achieve the desired result. For the "Duplicate Removal" process, the flag array produced by the *hash write* step can be used directly for *unique-index prefix-scan*

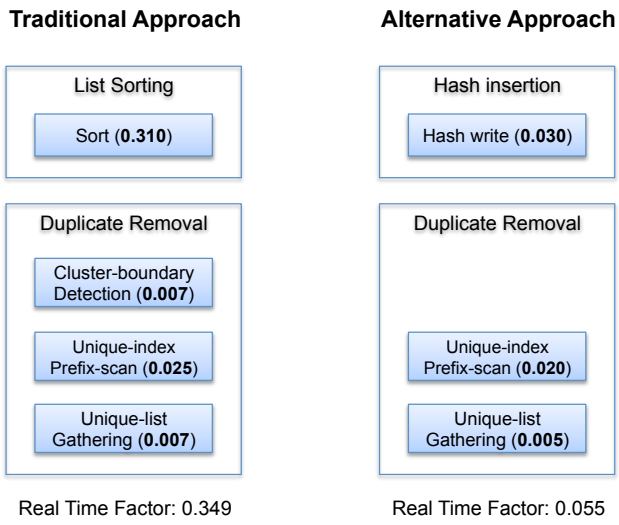


Figure 5.11: Find unique function approaches

and *unique-list gathering* steps. This significantly simplifies the find unique function, providing an efficient way to eliminate redundant work with a small overhead.

3. *Implementing lock-free accesses of a shared map to enable conflict-free reduction*

   Phase 2 of the automatic speech recognition algorithm performs a parallel graph traversal that is equivalent to a one level expansion during breadth-first search. Starting from a set of current active states, one follows the outgoing arcs to reach the next set of active states. Each state represents an alternative interpretation of the word sequence recognized so far.

   The generation of the next set of active states is done in two steps: 1) evaluating the likelihood of a word sequence when an arc is traversed, 2) recording the maximum likelihood of the traversed arc at the destination states (note that maximum likelihood is equal to minimum absolute value in negative log spaces). The first step involves computing the likelihood of particular transitions and is trivially data parallel. The second step involves a reduction over the likelihoods over all transitions ending in a destination state and recording only the transition that produced the most likely word sequence. This is the more challenging step to describe in a data parallel execution environment.

   One can organize the graph traversal in two ways: by *propagation* or by *aggregation*. During the graph traversal process, each arc has a source state and a destination state. Traversal by *propagation* organizes the traversal process at the source state. It evaluates the outgoing arcs of the active states and propagates the result to the destination states. As multiple arcs may be writing their result to the same destination state, this technique requires write conflict resolution support in the underlying platform. Traversal by *aggregation* organizes the traversal process around the destination state.

```
1 float res = compute_result(tid);
2 int stateID = get_stateID(tid);
3 Lock(stateID);
4 if(res < stateProbValues[stateID]) {
5     stateProbValues[stateID] = res;
6 }
7 Unlock(stateID);
```

Figure 5.12: Pseudo code for traversal by *propagation*

The destination states update their own information by performing a reduction on the evaluation results of their incoming arcs. This process explicitly manages the potential write conflicts by using additional algorithmic steps such that no write conflict resolution support is required in the underlying platform.

There is significant overhead in performing the traversal by *aggregation*. Compared to traversal by *propagation*, the process requires 3 more data parallel steps to collect destination states, allocate result buffers for the incoming arcs, evaluate arcs to write to designated result buffer index, and a final reduction over the destination state. The implementation of each step often requires multiple data parallel CUDA kernels to eliminate redundant elements and compact lists for more efficient execution.

For traversal by *propagation*, one can implement a direct-mapped table to store the results of the traversal for all states, indexed by the state ID. Conflicting writes to a destination state would have to be resolved by locking the write location, checking if the new result is more likely than the existing result, and selectively writing into the destination state. The approach is illustrated in the code segment below.

In the code snippet in Figure 5.12, each transition evaluation is assigned to a thread. The likelihood of the transition is computed by each thread (line 1). Then the $stateID$ of the destination state is obtained by each thread (line 2). The $stateProbValues$ array provides a lookup table storing the most likely interpretation of the input waveform that ends in a particular stateID. To conditionally update this lookup table, the memory location in the $stateProbValues$ array is locked (line 3) to prevent write-conflicts when multiple threads access the same state. The location then is updated if the probability computed represents a more likely interpretation of the input waveforms (lines 4-6), which mathematically is a smaller magnitude in log-likelihood. Finally, after the update, the memory location for the state is unlocked (line 7).

This approach is not efficient, as recording a result involves two atomic operations guarding a critical section and multiple dependent long-latency memory accesses. In some platforms, because atomic and non-atomic memory accesses may execute out-of-order, the pseudo-code as shown may not even be functionally correct.

Problem: How does one efficiently leverage CUDA capabilities to implement conflict-free reductions when handling large graphs?

```
1 float atomicMin(float* myAddress, float newVal);
```

Figure 5.13: The CUDA atomic operation with a logic operation

```
1 float res = compute_result(tid);
2 int stateID = StateID[tid];
3 int valueAtState = atomicMin(&(destStateProb[stateID]), res);
```

Figure 5.14: Using the CUDA atomic operation with a logic operation

Solution: Instead of having a locking/unlocking process for each state update, we take advantage of the *atomicMin()* operation in CUDA and implement a lock-free version of the traversal by *propagation*. The definition of *atomicMin()* is as follows:

From the CUDA manual [4]: The instruction reads the 32-bit word *oldVal* located at the address *myAddress* in global or shared memory, computes the minimum of *oldVal* and *newVal*, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns *oldVal*.

By using the *atomicMin()* instruction we implement a lock-free shared array update. Each thread obtains the stateID of the corresponding state and atomically updates the value at that location. The accumulation of the minimum probability is guaranteed by the *atomicMin()* semantics.

In the preceding code snippet, each thread computes the likelihood of the transition ending at $stateID$ (line 1) and obtains the $stateID$ of the destination state for its assigned transition (line 2). Then, the destination state transition probability is atomically updated by keeping the smallest magnitude of log-likelihood using the the *atomicMin()* operation. This technique efficiently resolves write-conflicts among threads (in the *destStateProb* array on line 3) and thus allows for lock-free updates to a shared array, selecting the minimum value at each array location. The resulting reduction in the write-conflict contention is at least 2× since we only have one atomic operation instead of two locks. The actual performance gain can be much greater, as the multiple dependent long-latency accesses in the original critical section between the locking and unlocking processes are also eliminated.

4. *Using hybrid local/global atomic operations and local buffers for the construction of a global queue*

   During traversal of the recognition network in the speech inference engine, one needs to manage a set of active states that represent the most likely interpretation of the input waveform. Algorithmically this is done by following arcs from the current set of active states and collecting the set of likely next active states. This process involves two steps: 1) computing the minimum transition probability and 2) collecting all new states encountered into a list. Section 3 discusses efficient implementation of the first

```
1 float res = compute_result(tid);
2 int stateID = StateID[tid];
3 float valueAtState = atomicMin(&(destStateProb[stateID]), res);
4 //first time seeing the state, push it onto the queue
5 if(valueAtState == FLT_MAX) {
6     int head = atomicAdd(&Q_head, 1);
7     myQ[head] = stateID;
8 }
```

Figure 5.15: A global queue based implementation of active state list generation with pruning

step on the GPU. If we implement this as a two-step process, after the first step, we would have an array of states where some of the states are marked "active". The size of this array equals the size of the recognition network, which is on the order of millions of states, and the number of states marked active is on the order of tens of thousands. To efficiently handle graph traversal on the GPU, one needs to gather the set of active states into a dense array to guarantee efficient access of the state data in consequent steps of the traversal process.

One way to gather the active states is to utilize part of the routine described in Subsection 2, by constructing an array of flags that has a "1" for an active state and a "0" for a non-active state and then performing a "prefix-scan" and "gather" to reduce the size of the list. However, the prefix-scan operation on such large arrays is expensive, and the number of active states is less than 1% of the total number of states.

Instead, one can merge the two steps and actively create a global queue of states while performing the write-conflict resolution. Specifically, every time a thread evaluates an arc transition and encounters a new state, it can atomically add the state to the global queue of next active states. The resulting code is as follows:

In Figure 5.15, each thread evaluates the likelihood of arriving at a state (line 1) and obtains the $stateID$ of the destination state for its assigned transition (line 2). Each entry of the $destStateProb$ array is assumed to have been initialized to FLT_MAX, the largest log-likelihood magnitude (representing the smallest possible likelihood). In line 3, the value in the $destStateProb$ array at the $stateID$ position is conditionally and atomically updated with the evaluated likelihood. If the thread is the first to encounter the state at $stateID$, the returned value from line 3 would be the initial value of FLT_MAX. In that case, the new state is atomically added to the queue of destination states. This is done by atomically incrementing the queue pointer variable using the *atomicAdd* operation on the Q_head pointer (line 6). The value of the old queue head pointer is returned, and the actual $stateID$ can be recorded at the appropriate location (line 7).

While the merged implementation is an efficient way to avoid instantiating numer-

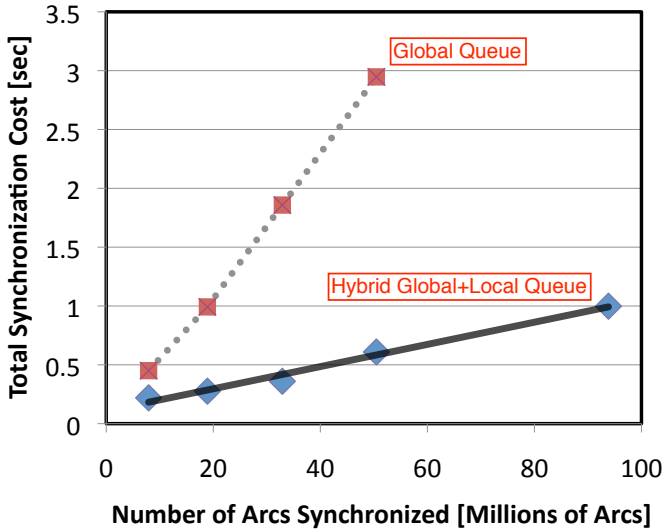


Figure 5.16: Comparison of the synchronization cost for global queue and hybrid global and local queue implementations

ous data parallel kernels for collecting active states in the two-step process, there is one sequentializing bottleneck that prevents this implementation from being scalable: when thousands of threads are executing this kernel concurrently, there are thousands of threads atomically updating the *Q_head* variable. As shown in Figure 5.16, the synchronization cost of a global queue implementation increases sharply as the number of state transitions evaluated increases.

Problem: How does one efficiently construct a shared global queue?

Solution: In order to resolve the contention on the head pointer, one can use a distributed queue implementation. Each CUDA thread block creates its own local shared queue and populates it every time a thread encounters a new state. After each block constructs its local queue of states, each block then merges its local queue into the global queue. The possible contention on the local queue head pointer is limited to the number of threads in a thread block (usually 64 to 512), and the contention on the global queue head pointer is thus reduced by 64× to 512×, greatly improving performance. Figure 5.16 shows that the hybrid implementation scales gracefully with an increasing number of arcs. The resulting code is shown below:

In Figure 5.17, lines 3-6 declare the shared memory variables: local queue *myQ* and local queue head *myQ_head* index and the head index of the global queue, *globalQ_index*. Thread 0 initializes the local queue head for each block in lines 8-10. The thread ID is computed and the bounds are checked in lines 13 and 15. Each thread obtains the destination state for its transition and computes the transition probability in lines 16 and 17 respectively. In lines 19-29, the probability for the destination state is updated as described above and in Section 3. This time, however, the new states encountered

```
1  // Local Q: shared memory data structure
2  // ----------------------------------------
3  extern __shared__ int sh_mem[];  // Q size, at most #threads
4  int *myQ = (int *)sh_mem;        // memory for local Q
5  __shared__ int myQ_head;         // local Q head ptr
6  __shared__ int globalQ_index;    // globalQ_index
7
8  if(threadIdx.x==0) {
9      myQ_head = 0;
10 }
11 __syncthreads();
12
13 int tid = blockIdx.x*blockDim.x + threadIdx.x;
14
15 if(tid<nStates) {
16     int stateID = StateID[tid];
17     float res = compute_result[tid];
18
19     if (res > pruneThreshold) {
20         res = FLT_MAX;
21     } else {
22         //if res is smaller than prune threshold
23         int valueAtState = atomicMin(&(destStateProb[stateID]), res);
24         if(valueAtState == FLT_MAX) {
25             //first time seeing the state, push into local queue
26             int head = atomicAdd(&myQ_head, 1);
27             myQ[head] = stateID;
28         }
29     }
30
31     // Local Q -> Global Q transfer
32     // ----------------------------------------
33     __syncthreads();
34
35     if (threadIdx.x==0) {
36         globalQ_index = atomicAdd(stateHeadPtr, myQ_head);
37     }
38     __syncthreads();
39
40     if (threadIdx.x < myQ_head)
41         destStateQ[globalQ_index+threadIdx.x] = myQ[threadIdx.x];
42
43 } // end if(tid<nStates)
```

Figure 5.17: A hybrid global/local queue based implementation of active state list generation with pruning

by the threads in each block are stored locally in the local queue (line 26). Once all threads in a thread block have completed their enqueuing operations to the local queue, the thread block adds the content of the local queue to the global queue as a block. Lines 35-37 update the global queue index, and line 41 copies the contents of the local queue to the global queue in parallel.

5.1.3 Application-Specific Algorithm Selection

A host of general optimization techniques have been described in Section 5.1.2. These techniques are part of a parallel programming expert's tool kit and are broadly applicable in a variety of highly parallel algorithms for implementing ASR.

A general approach for implementing ASR is the token-passing approach where a set of active tokens is maintained to represent all alternative interpretations being tracked in the speech input. Significant research efforts have focused on optimizing the representation of the recognition network used during inference. The baseline representation is the Linear Lexical Model (LLM), where each word is simply represented by a chain of phone states. Figure 5.18 shows the recognition network, which consists of a collection of chains with one chain representing each word in the vocabulary. The figure also shows that the word-to-word transitions are stored separately.

The Tree-Lexical Model (TLM) and WFST are the two widely adopted optimizations of the LLM recognition network. Tree-organization of the pronunciation lexicon reduces the number of states being traversed during recognition [120] by sharing pronunciation prefixes in a prefix tree structure in representing words in the pronunciation model. In TLM, language model look-ahead can be applied for efficient pruning [124]. More recently, the WFST representation has seen wide adoption [112] because of its application of powerful finite state machine transformations to remove redundant states and arcs during offline network compilation. As reported in [92], the WFST representation is faster than the tree-lexical representation as it explores even less search space. Both TLM and WFST show faster recognition speed than conventional linear lexical search network on a sequential processor.

Network Structure

The network structures for the LLM and the WFST representations differ significantly. Figure 5.18 illustrates the construction of the LLM representation, which involves a chain of triphone states for each pronunciation to be recognized. For example, a dictionary of 5,000 word pronunciations would have 5,000 chains of triphone states in the recognition network. There are many duplicates as each pronunciation chain is constructed using a separate copy of the triphone states for the chain's phone sequence. The possibility of sharing common prefix is not considered in the LLM representation. The language model captures the likelihood of word-to-word transitions. It is used when the token-passing recognition process reaches the end of a chain of triphones. Since a word can be followed by any word, one must evaluate the possibility of transitions from one word to all words in the vocabulary.

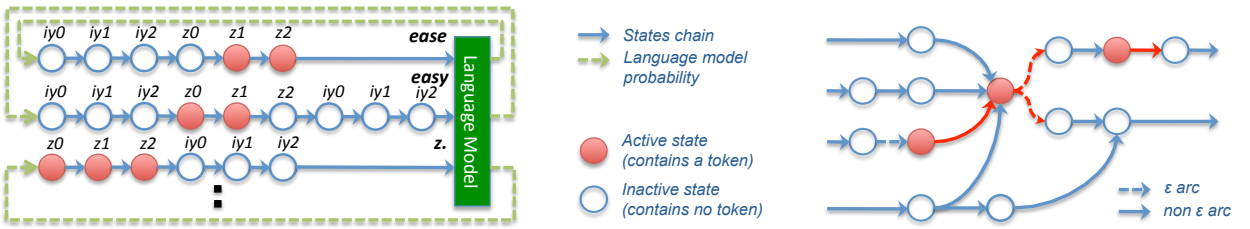


Figure 5.18: Structure of the recognition network for the LLM representation and WFST representation.

In contrast, the WFST representation of the recognition network is constructed by composing the pronunciation model and the language model using powerful finite state machine (FSM) composition techniques. Both within-word transitions and across-word transitions are explicitly represented in the composed network. The sparsity structure of natural languages and the minimization techniques employed in [112] allow the WFST representation to avoid the state explosion problem when composing the models. As reported in [112, 92], the WFST representation is a concise representation that encapsulates a large amount of information with little redundancy. As we will show in our results, compared to the LLM representation, many fewer tokens are required to be maintained for the WFST representation during inference to achieve a target recognition accuracy.

The separate pronunciation and language models in the LLM representation allow for highly optimized computation kernel design, compensating for the performance lost from the redundancies in the representation.

Traversal Implementation

The two inference engines are implemented on the GPU, with one using the LLM representation of the recognition network and the other using the WFST representation. Figure 5.19 illustrates the application structure that was used for both inference engines. In this structure, both the observation probability computation and the token passing phases are implemented on the GPU. This structure has been shown to provide the best recognition speed on GPUs for both types of recognition networks [45, 157]. The implementations are designed to utilize hardware support for atomic operations on the GPU. In Phase 1, each Gaussian Mixture Model (GMM) in the acoustic model is evaluated in a thread-block; in Phase 2, each arc in the recognition network is evaluated by a thread. Backtracking is performed on the CPU. We highlight the key differences between the graph traversal process on recognition networks using the LLM representation and the WFST representation.

The LLM representation: Efficient implementation of the graph traversal of a network with LLM representation depends on explicitly handling two types of transitions in the LLM representation: within-word transitions and across-word transitions. This distinction was originally used in a parallel implementation by [130], and is elaborated here to include three specialized data structures for the GPU implementation. The three structures effi-

ciently represent the chains of triphone states as the *first*, *middle* and *last* state in a word pronunciation. Within-word transition computation uses *first* and *middle* states as inputs and updates the *middle* and *last* state. Across-word transition computation uses *last* states as inputs and updates the *first* states. The distinct memory access patterns for within-word and across-word transitions can be efficiently implemented using specialized kernel routines and data layouts. Specifically, the first triphone states are stored consecutively in memory to optimize for across-word transitions and the middle and last states for each word are stored as a chain of consecutive states. These optimizations were originally proposed in [45] and have been re-implemented to take advantage of the new capabilities in the current generation of GPUs.

The WFST representation: An inference engine using the WFST representation does not distinguish between within-word and across-word transitions, as the pronunciation and the language models are compiled into a monolithic weighted finite state transducer. However, the recognition network does distinguish between non-epsilon and epsilon arcs as a result of state machine minimization during the network construction. The implementation optimizations of the WFST inference engine are described in [157].

Both implementations have been optimized with fast hardware atomic operations as well as using dynamically constructed efficient runtime buffers to gather operands that are dispersed in memory. Figure 5.19 illustrates the operand gather step in Phase 0. While the LLM representation has a regular data structure that enables the utilization of specialized data layout to achieve highly efficient implementations, such optimization is not possible with the WFST representation, as the WFST representation involves highly irregular data structures. The across-word transitions are a significant computation bottleneck for the LLM representation, as a typical recognition process evaluates more than 20 million transitions each second. In our implementation, we represent these computation as dense matrix operations that can be executed on the GPU extremely efficiently. For the WFST representation, the irregular graph structures require many of the operations to be handled as sparse matrix operations.

Experimental Platform and Setup

We used both the NVIDIA GTX480 (Fermi) GPU and the NVIDIA GTX285 GPU with a Intel Core i7 920 based host platform. GTX480 has 15 cores each with dual issue 16-way vector arithmetic units running at 1.40GHz. Its processor architecture allows a theoretical maximum of two single-precision floating point operations (SP FLOP) per cycle, resulting in a maximum of 1.35 TeraFLOP of peak performance per second. GTX285 has 30 cores with 8-way vector arithmetic units running at 1.51GHz. The processor architecture allows a theoretical maximum of three SP FLOP per cycle, resulting in a maximum of 1.087 TeraFLOP of peak performance per second. For compilation, we used Visual Studio 2008 with nvcc 3.0 and nvcc 2.2 respectively.

The acoustic model was trained by HTK [159] with the speaker independent training data in the Wall Street Journal 1 corpus. The frontend uses 39 dimensional features that have 13 dimensional MFCC, delta and acceleration coefficients. The trained acoustic model consists of 3,000 16-mixture Gaussians. Two language model sizes are used to illustrate the

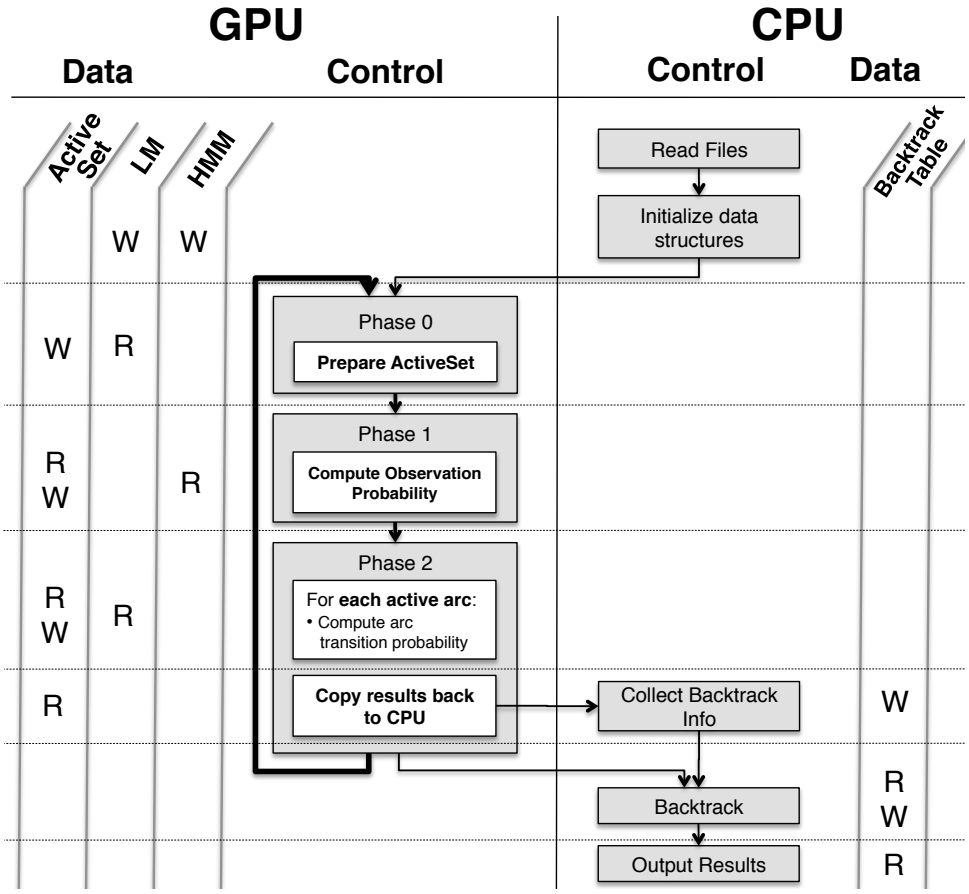


Figure 5.19: Control flow for the CPU/GPU software implementation and associated data structure access (R: read, W: write).

effect of a reduced language model size on the recognition performance. The language model weight is set to 15. The WFST network is an $H \circ C \circ L \circ G$ model compiled and optimized offline with the dmake tool described in [11]. The test set consists of 330 sentences totaling 2,404 seconds from the Wall Street Journal test and evaluation set. The serial reference implementation using the LLM representation has a word error rate (WER) of 8.0% and runs with a 0.64 real time factor. The number of states and arcs representing the recognition networks used are shown in Table 5.2. WFST representation explicitly stores all across-word transitions, and LLM representation skips many across-word transitions as they are implied by the format. This leads to the differences in the actual number of arcs stored.

Table 5.2: Recognition network sizes used in experiments

	LLM Pruned	LLM	WFST Pruned	WFST
# State	123,246	123,246	1,091,295	3,925,931
# Arcs	537,608	1,596,884	2,955,145	11,394,956

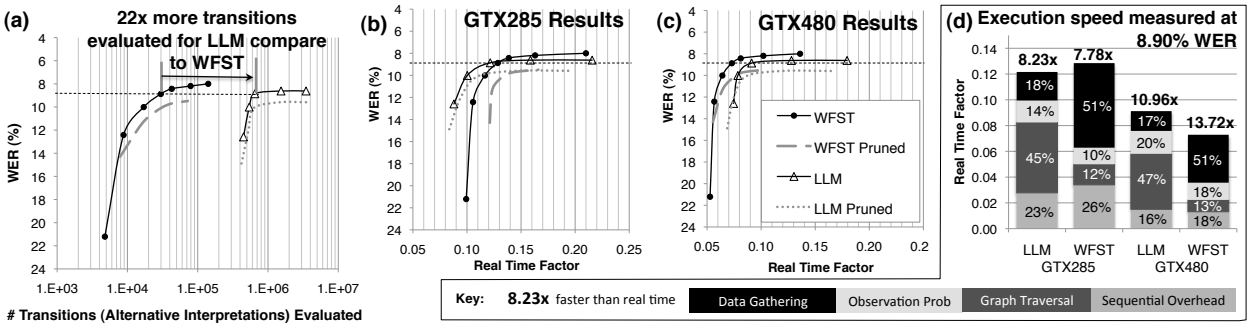


Figure 5.20: WER w.r.t. # of transitions evaluated (a), execution time in Real Time Factor (b/c), and speed analysis at 8.90% WER (d).

Computation Load, Accuracy, and Speed

We measured the number of transitions evaluated by an inference engine using both the LLM and the WFST representations of the recognition network. Figure 5.20(a) illustrates the accuracy vs computation effort curves for both representations. Each curve represents a set of recognition accuracy results by adjusting the average number of states maintained active. A dynamic threshold prediction scheme is used for pruning. We chose an operating point at 8.90% WER for the following comparison with an 11% relaxation (8.0% to 8.9%) in accuracy to gain almost a doubling in decoding speed (from 7.4× to 13.7× faster than real time for the WFST representation on GTX480).

At 8.90% WER, by using the LLM representation, an inference engine must evaluate 22× more transitions compared to using the WFST representation. For a sequential implementation where the execution is usually compute-bound, using the WFST representation could provide a significant speed advantage over the LLM representation.

The speed and accuracy trade-off shifts significantly for implementations on highly parallel platforms. Figure 5.20(b) illustrates the speed of the speech inference engine on the GTX285. The speed is shown as *Real Time Factor*, which is the number of seconds required to process one-second of speech input. The simplicity of the LLM representation allows the inference process to surpass the speed of the WFST representation for target WER of more than 8.8%.

Figure 5.20(c) illustrates the speed of the speech inference engine on the GTX480. Going from GTX285 to GTX480, the new processor micro architecture provided an average of 25% speed improvement for the LLM representation and 79% improvement for the WFST representation. On the GTX480, with the availability of data caches, the WFST representation becomes the faster implementation. Its irregular *Data Gathering* phase and *Graph Traversal* phase enjoys a 74% boost to its performance, while the same phases in the LLM representation only received a 27% performance improvement, which were already well parallelized using algorithm specific data structures.

In Figure 5.20(d), we see significant differences in the run time characteristics when using of the two representations of the recognition network across the two highly parallel

platforms. To achieve 8.90% WER, the inference process using the LLM representation evaluates on average 649k transitions per time step, and obtains this at 8.23× and 10.96× faster than real time when executed on the GTX285 and the GTX480 respectively. The inference process with the WFST representation evaluates on average 29.8k transitions per time step to get 8.90% WER, and achieves this at 7.78× and 13.72× faster than real time when executed on the GTX285 and the GTX480 respectively.

The regular data layout of the LLM representation greatly reduces the cost of the *Data Gathering* phase of inference engine, taking less than 18% of the overall run time, while the WFST representation spends more than half its runtime in *Data Gathering*. For the *Graph Traversal* phase, although performing inference with the LLM representation takes 3-5× more time compared to the WFST representation, it is evaluating 22× the number of state transitions. The speed of the LLM representation is competitive with the WFST representation on highly parallel platforms because per state transition, it is 53-65× faster in *data gathering* and 4.7-6.4× faster in *graph traversal*.

Comparing performance between GTX285 and GTX480, we see an 85% and a 159% improvement in handling *sequential overhead* for the LLM and the WFST representations respectively. These are dominated by transfers of backtracking data from GPU to CPU.

Conclusions and Discussions

The emergence of highly parallel computing platforms brings forth the opportunity to reevaluate the computational efficiency of different approaches in speech recognition. In particular in this paper, we consider the LLM and WFST representations of the recognition network in a speech inference engine. We found that despite the advantages of the sophisticated WFST representation for a sequential implementation, the simpler LLM representation performs competitively on highly parallel platforms. The LLM representation required the traversal of 22× the number of state transitions. However, per state transition, it gathers data 53-65× faster and evaluates the transitions 4.7-6.4× faster than the WFST representation. Depending on the choice of the implementation platform, for the same 8.90% WER, the LLM representation is faster on the GTX285, and the WFST representation is faster on the GTX480. Going forward, the WFST representation is selected for further optimization and analysis.

5.1.4 Application-Specific Input Transformations

Given the advantages of using the WFST recognition network representation for ASR implementation, with application domain knowledge, one can transform the recognition network representation to achieve more efficient execution on the GPUs.

Communication-intensive Phase Optimizations

The communication-intensive phase involves a graph traversal process through an irregular recognition network. There are two types of arcs in a WFST-based recognition network: arcs with an input label (non-epsilon arcs), and arcs without input labels (epsilon arcs). In

order to compute the set of next states in a given time step, one must traverse both the non-epsilon and all the levels of epsilon arcs from the current set of active states. This multi-level traversal can impair performance significantly. The modification of flattening the recognition network is explored to reduce the number of levels of arcs that need to be traversed and observe corresponding performance improvements. To illustrate this, Figure 5.21 shows a small section of a WFST-based recognition network. Each time step starts with a set of currently active states representing the alternative interpretations of the input utterances. In Figure 5.21, the active states are represented by states (1) and (2). The traversal then proceeds to evaluate all out-going non-epsilon arcs to reach a set of destination states, e.g. states (3) and (4). Next, the traversal extends through epsilon arcs to reach more states, e.g. state (5) for the next time step.

The traversal from state (1) and (2) to (3), (4) and (5) can be seen as a process of active state wave-front expansion in a time step. The challenge for data parallel operations is that the expansion from (1) to (3) to (4) to (5) requires multiple level traversal: one non-epsilon level and two epsilon levels. The traversal incurs significant instruction stream divergence and uncoalesced memory accesses if each thread expands through an arbitrary number of levels. Instead, a data parallel traversal limits the expansion to one level at a time and the recognition network is augmented such that the expansion process can be achieved with a fixed number of single level expansions. Figure 5.21 illustrates the necessary recognition network augmentations for Two-Level and One-Level setups.

Each step of expansion incurs some overhead, so to reduce the fixed cost of expansion we want fewer number of steps in the traversal. However, depending on recognition network topology, augmenting the recognition network may cause significant increase in the number of arcs in the recognition network, thus increasing the variable cost of the traversal. We demonstrate this trade-off with a case study in the Results section.

Experimental Platform and Baseline Setup

We use the NVIDIA GTX280 GPU with a Intel Core2 Q9550 based host platform. GTX280 has 30 cores with 8-way vector arithmetic units running at 1.296GHz. The processor architecture allows a theoretical maximum of 3 floating point operations (FLOP) per cycle, resulting in a maximum of 933 GFLOP of peak performance per second. The sequential results were measured on an Intel Core i7 920 based platform with 6GB of DDR memory. The Core i7-based system is 30% faster than the Core2-based system because of its improved memory sub-system, providing a more conservative speedup comparison. The sequential implementation was compiled with icc 10.1.015 using all automatic vectorization options. Kernels in the compute-intensive phase were hand optimized with SSE intrinsics [46]. Figure 5.22 shows that the sequential performance achieved was 3.23 seconds per one second of speech, with Phase 1 and 2 taking 2.70 and 0.53 seconds respectively. The parallel implementation was compiled with icc 10.1.015 and nvcc 2.2 using Compute Capability v1.3.

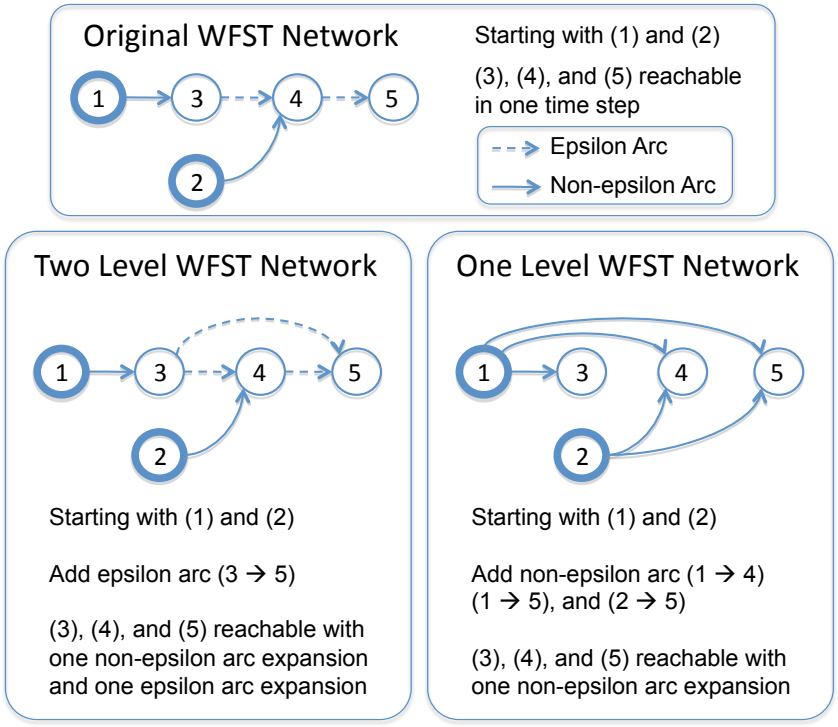


Figure 5.21: Network modification techniques for a data parallel inference engine.

Speech Models and Test Sets

The speech models were taken from the SRI CALO realtime meeting recognition system [146]. The front-end uses 13d PLP features with 1st, 2nd, and 3rd order differences, VTL-normalized and projected to 39d using HLDA. The acoustic model was trained on conversational telephone and meeting speech corpora, using the discriminative MPE criterion. The LM was trained on meeting transcripts, conversational telephone speech and web and broadcast data [142]. The acoustic model includes 52K triphone states which are clustered into 2,613 mixtures of 128 Gaussian components. The recognition network is an $H \circ C \circ L \circ G$ model compiled using WFST techniques [46].

The test set consisted of excerpts from NIST conference meetings taken from the "individual head-mounted microphone" condition of the 2007 NIST Rich Transcription evaluation. The segmented audio files total 3.1 hours in length and comprise 35 speakers. The meeting recognition task is very challenging due to the spontaneous nature of the speech[3]. The ambiguities in the sentences require larger number of active states to keep track of alternative interpretations, which leads to slower recognition speed.

Our recognizer uses an adaptive heuristic to control the number of active states by adjusting the pruning threshold at run time. This allows all traversal data to fit within a

[3]A single-pass time-synchronous Viterbi decoder from SRI using lexical tree search achieves 37.9% WER on this test set

Table 5.3: Accuracy, word error rate (WER), for various beam sizes and decoding speed in real-time factor (RTF)

Avg. # of Active States		32398	19306	9763	3390
WER		51.1	50.9	51.4	54.0
RTF	Sequential CPU	4.36	3.17	2.29	1.20
	Parallel GPU	0.37	0.29	0.20	0.14
Speedup		11.7	11.0	11.3	9.0

pre-allocated memory space. Table 5.3 shows the decoding accuracy, i.e., word error rate (WER) with varying thresholds and the corresponding decoding speed on various platforms. The recognition speed is represented by the real-time factor (RTF), which is computed as the total decoding time divided by input speech duration.

As shown in Table 5.3, the GPU implementation can achieve order of magnitude more speedup over the sequential implementation [46] for the same number of active states. More importantly, one can trade-off speedup with accuracy. For example, one can achieve 54.0% WER traversing an average of 3390 states per time step with a sequential implementation, or one can achieve a 50.9% WER traversing an average of 19306 states per time step while still getting a 4.1× speedup, improving from an RTF of 1.20 to 0.29.

Compute-intensive Phase

The parallel implementation of the compute-intensive phase achieves close to peak performance on GTX280. As shown in Table 5.4, we found the GMM computation memory-bandwidth-limited and our implementation achieves 85% of peak memory bandwidth. The logarithmic mixture reduction is compute-limited and our implementation achieves 98% of achievable peak compute performance given the instruction mix.

Table 5.4: Efficiency of the Computation Intensive Phase

(GFLOP/s)	Step 1	Step 2
Theoretical Peak	933	933
	Mem BW Limited	Inst Mix Limited
Practical Peak	227	373
Measured	194	367
Utilization	85%	98%

Communication-intensive Phase

Parallelizing this phase on a quadcore CPU achieves a 2.85× performance gain [46] and incurs intermediate result transfer overhead. We achieved a 3.84× performance gain with an equivalent configuration on the GPU and avoided intermediate result transfer overhead. Despite the better speedup on the GPU, this phase became more dominant as shown in Fig 5.22.

Table 5.5: Performance with Different Recognition Network Augmentation (Run times Normalized to one Second of Speech)

Active States	Original WFST Network				Two-Level WFST Network				One-Level WFST Network			
(% of state space)	0.1%	0.3%	0.5%	1.0%	0.1%	0.3%	0.5%	1.0%	0.1%	0.3%	0.5%	1.0%
Total States	4,114,507				4,114,672 (+0.003%)				4,116,732 (+0.05%)			
Total Arcs	9,585,250				9,778,790 (+2.0%)				12,670,194 (+32.2%)			
Arcs Traversed*	27,119	64,489	112,173	171,068	27,342	65,218	114,043	173,910	44,033	103,215	174,845	253,339
Arcs increase (%)	-	-	-	-	+0.8%	+1.1%	+1.7%	+1.7%	+62%	+60%	+56%	+48%
Phase 1 (ms:%)	77:41%	112:43%	146:43%	177:41%	77:48%	112:50%	146:48%	178:45%	73:55%	110:55%	147:51%	177:48%
Phase 2 (ms:%)	97:52%	127:49%	171:50%	230:53%	74:46%	99:44%	138:45%	191:48%	52:39%	81:40%	125:43%	175:47%
Seq. Ovrhd (ms:%)	13:7%	20:8%	24:7%	28:6%	11:7%	14:6%	20:7%	25:6%	8: 6%	10:5%	16:5%	21:6%
Total (ms)	187	258	341	436	161	225	304	393	134	202	289	373
Faster than real time	5.3×	3.9×	2.9×	2.3×	6.2×	4.4×	3.3×	2.5×	7.5×	5.0×	3.5×	2.7×

** Average number of arcs traversed per time step*

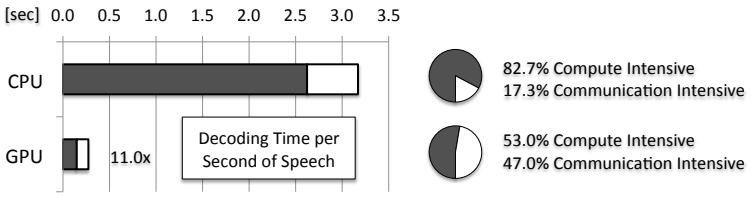


Figure 5.22: Parallel Speedup of the Inference Engine.

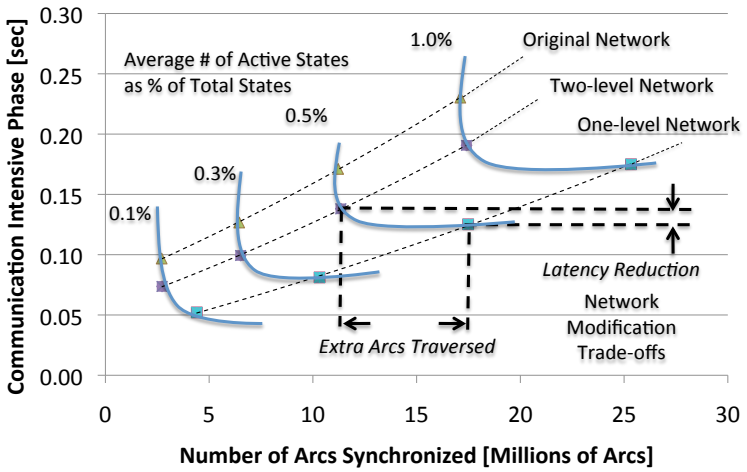


Figure 5.23: Communication Intensive Phase Run Time in the Inference Engine (normalized to one second of speech).

Table 5.5 demonstrates the trade-offs of recognition network augmentation for efficient data parallel traversal in our inference engine. The augmentation for Two-Level setup resulted in a 2.0% increase in arc count and the augmentation for One-Level setup resulted a 32.2% increase. The dynamic number of arcs evaluated increased marginally for the Two-Level setup. However for the One-Level solution it increased significantly by 48-62%, as states with more arcs were visited more frequently.

Fig 5.23 shows the run time for various pruning thresholds. The network modifications are described earlier in this section. Without network modifications, there is significant performance penalty, as multiple levels of epsilon arcs must be traversed with expensive global synchronization steps between levels. With minimal modifications to the network, we see a 17-24% speedup for this phase. An additional 8-29% speedup can be achieved by eliminating epsilon arcs completely, saving the fixed cost of one level of global synchronization routines, but this comes at the cost of traversing 48-62% more arcs.

Conclusion

A fully data parallel speech inference engine is presented with both observation probability computation and graph traversal implemented on an NVIDIA GTX280 GPU. The results show that being able to leverage application domain expertise to make modifications to the recognition network is essential for efficient implementation of data parallel WFST-based LVCSR algorithm on GPU. The implementation achieved up to 11.7× speedup compared to highly optimized sequential implementation with 5-8% sequential overhead without sacrificing accuracy. This software architecture enables performance improvement potentials on future platforms with more parallelism.

5.1.5 Optimizations on Various Hardware Platforms

This section demonstrates the implications of implementing the highly challenging graph traversal operations on two highly parallel architectures: an Intel Core i7 multicore processor and an NVIDIA GTX280 manycore processor. Multicore processors are considered as processors that devote significant transistor resources to complex features for accelerating single thread performance, whereas manycore processors are considered as processors that use their transistor resources to maximize total instruction throughput at the expense of single thread performance. It is shown that the best algorithm on one architecture may perform poorly on another due to varying efficiencies of key parallel operations, and that the efficiency of the key parallel operations is more indicative of the performance of the application implementation.

Given the challenging and dynamic nature of the underlying graph-traversal routines in LVCSR, implementing it on parallel platforms presents two architectural challenges: efficient SIMD utilization and efficient core level synchronization. These challenges are key factors in making the algorithms scalable to increasing number of cores and SIMD lanes. To find a solution to these challenges, two aspects of the algorithmic level design space were explored: the graph traversal technique and the arc transition evaluation granularity. Figure 5.24 illustrates the design space under consideration.

Traversal Techniques: Aggregate or Propagate

The two graph traversal techniques are traversal by *propagation* and traversal by *aggregation*. During the graph traversal process, each arc has a source state and a destination state. Traversal by *propagation* organizes the traversal process at the source state. It evaluates the outgoing arcs of the active states and *propagates* the result to the destination states. As multiple arcs may be writing their result to the same destination state, this technique requires write conflict resolution support in the underlying platform. The programmer declares certain memory operations as atomic, and the implementation platform resolves the potential write conflicts.

Traversal by *aggregation* organizes the traversal process around the destination state. The destination states update their own information by performing a reduction on the evaluation results of their incoming arcs. The programmer explicitly manages the potential

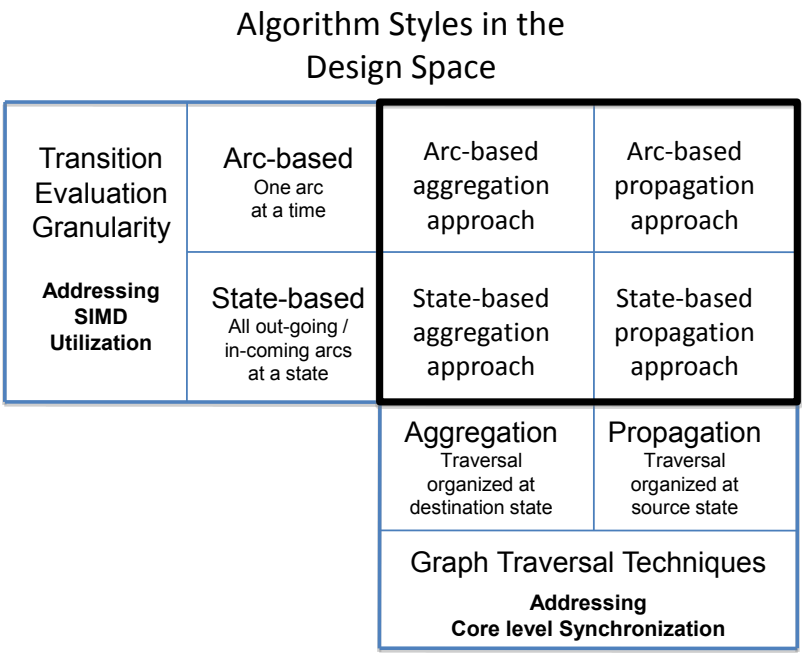


Figure 5.24: The algorithmic level design space for graph traversal scalability analysis for the inference engine.

write conflicts by using additional algorithmic steps such that no write-conflict-resolution support is required in the underlying platform.

The choice of the traversal technique has direct implications on the cost of core level synchronization. Efficient synchronization between cores reduces the management overhead of a parallel algorithm and allows the same problem to gain additional speedups as we scale to more cores. Figure 5.25 outlines the trade-offs in the total cost of synchronization between the aggregation technique and the propagation technique. The qualitative graph shows increasing synchronization cost with increasing number of concurrent states or arcs evaluated.

The Y-intercept of line (a) in Figure 5.25 illustrates the fixed cost for the aggregation technique, which is higher than that of the propagation technique, as it requires a larger data structure and a more complex set of software routines to manage potential write conflicts. The relative gradient of the aggregation and propagation techniques depends on the efficiency of the platform in resolving potential write conflicts. If efficient hardware-supported atomic operations are used, the variable cost for each additional access would be small, and the propagation technique should scale as line (b) in Figure 5.25. If there is no hardware support for atomic operations, and sophisticated semaphores and more expensive software-based locking routines are used, the propagation technique would scale as line (c). In addition, if the graph structure creates a scenario where many arcs are contenting to write to a small set of next states, serialization bottleneck may appear and the propagation technique could scale as line (d).

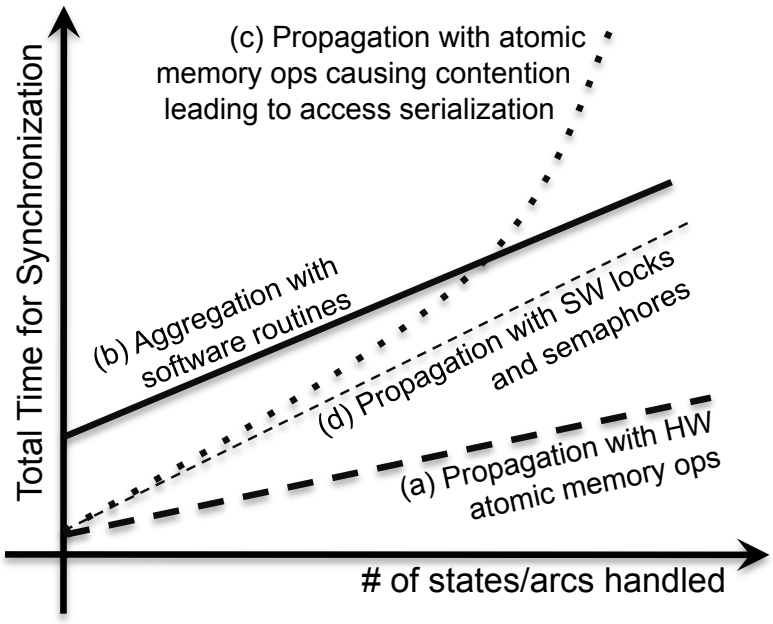


Figure 5.25: Scalability of the traversal process in terms of total synchronization time.

In order to minimize the synchronization cost for a given problem size, we need to choose the approach corresponding to the lowest-lying line in Figure 5.25. For small number of active states or arcs, we should choose the propagation technique. For larger number of arcs, however, the choice is highly dependent on the application graph structure and the write-conflict-resolution support in the underlying implementation platform.

Evaluation Granularity: Arc-Based or State-Based

We also explore two recognition network evaluation granularity: *state-based* evaluation and *arc-based* evaluation. In a parallel implementation we must define units of work (or tasks) that can be done concurrently. *State-based* evaluation defines a unit of work as the evaluation of all outgoing or incoming arcs associated with a *state*, with the majority of states having one or two outgoing or incoming arcs. *Arc-based* evaluation defines a unit of work as the evaluation of a single *arc*. The fine granularity of tasks allows the workload to scale to increasingly parallel implementation platforms. Each fine-grained task, however, has little instruction-level parallelism and can not fully utilizing a growing number of SIMD lanes in a core. We must efficiently map tasks to SIMD lanes to gain higher SIMD utilization, such that the algorithm can scale to even wider SIMD units in future processors.

SIMD operations improve performance by executing the same operation on a set of data elements packed into a contiguous vector. Thus, SIMD efficiency is highly dependent on the ability of all lanes to synchronously execute useful instructions. When all lanes are fully utilized for an operation, we call the operation "coalesced". When operations do not coalesce, the SIMD unit becomes under-utilized.

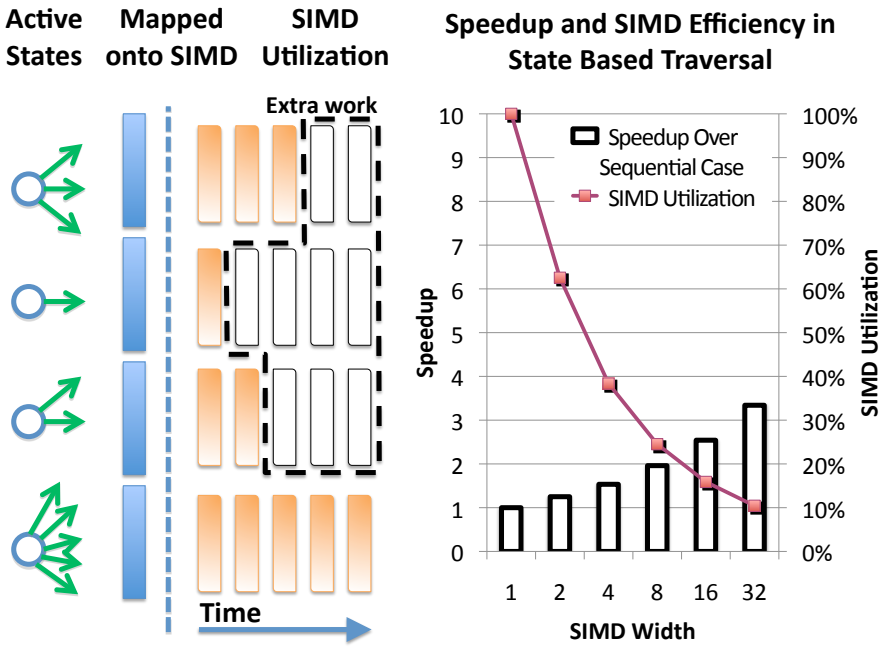


Figure 5.26: SIMD unit utilization in the active-state-based traversal.

For the state-based approach, we see on the left of Figure 5.26 that the control flow diverges as some lanes are idle, while others are doing useful work. In our recognition network, the number of outgoing arcs of the active states ranges from 1 to 897. The bar chart on the right of Figure 5.26 shows that the state-based evaluation granularity incurs significant penalties with increasing SIMD width. A 32-wide SIMD achieves only 10% utilization and gets only a 3.3× speedup over a sequential version.

We can eliminate this kind of control flow divergence by using the arc-based approach, as each arc evaluation presents a constant amount of work. However, such fully coalesced control flow requires extra storage overhead. In order for each arc evaluation to be an independent task, the arc must have a reference to its source state. We must store this information for every arc we evaluate. For a small SIMD width, this overhead may eliminate any gains from coalesced control flow.

Evaluation of the Inference Engine

We explore the traversal techniques and evaluation granulates on two hardware platforms: the Intel Core i7 920 multicore processor with 6GB memory and the NVIDIA GTX280 manycore processor with a Core2 Quad based host system with 8GB host memory and a GTX280 graphics card with 1GB of device memory. The Core i7 was programmed using the task queue abstraction [100], and the GTX280 was programmed using CUDA [122]. The results are shown in Table 5.7.

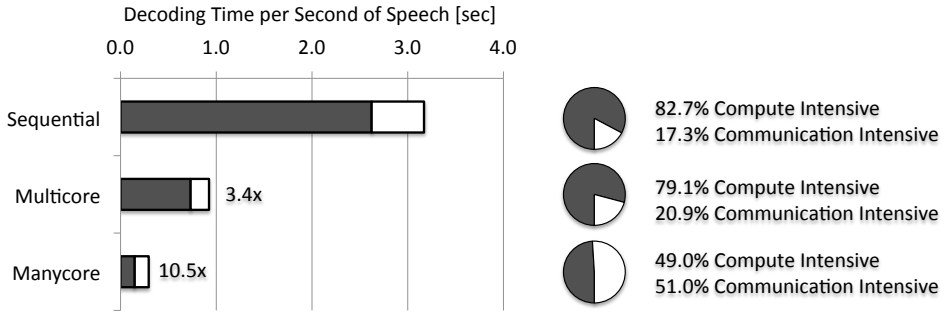


Figure 5.27: Ratio of computation intensive phase of the algorithm vs communication intensive phase of the algorithm.

Figure 5.27 highlights the growing importance of optimizing the communication intensive phases on parallel platforms. Over 80% of the time in the sequential implementation is spent in the compute intensive phases of the application. While the compute intensive phase achieved a 4-20× speedup in our highly parallel implementations, the communication intensive phases incurred significant overhead in managing the parallelism. Even with a respectable 3× speedup, the communication intensive phases became proportionally more dominant, taking 49% of total runtime in the manycore implementation. This motivates the need to examine in detail the parallelization issues in the communication intensive phases of our inference engine. We mainly analyze its synchronization efficiency and effectiveness of using SIMD parallelism.

Table 5.6: Accuracy, word error rate (WER), for various beam sizes and corresponding decoding speed in real-time factor (RTF).

Avg. # of Active States		32820	20000	10139	3518
WER		41.6	41.8	42.2	44.5
RTF	Sequential	4.36	3.17	2.29	1.20
	Multicore	1.23	0.93	0.70	**0.39**
	Manycore	**0.40**	0.30	0.23	0.18

The speech models were taken from the SRI CALO realtime meeting recognition system [146], trained using the data and methodology developed for the SRI-ICSI NIST RT-07 evaluation [142]. The acoustic model includes 128-component Gaussians. The pronunciation model contains 59K words and 80K pronunciations. The recognition network contains 4.1M states and 9.8M arcs and is composed using WFST techniques.

The test set consisted of 44 minutes of segmented audio from 10 speakers in NIST conference meetings. The recognition task is very challenging due to the spontaneous nature of the speech. The ambiguities in the sentences require a larger number of active states to keep track of alternative interpretations, which leads to slower recognition speed.

As shown in Table 5.6, the multicore and manycore implementations can achieve significant speedup for the same number of active states. More importantly, for the same real time

Table 5.7: Recognition performance normalized for one second of speech for different algorithm styles with average of 20,000 active states each iteration. Speedup reported over SIMD-optimized sequential version. Results explained are in bold.

	Core i7	Core i7			GTX280			
Seconds (%)	Sequential Prop. by states	Prop. by states	Prop. by arcs	Aggr. by states	Prop. by states	Prop. by arcs	Aggr. by states	Aggr. by arcs
Phase 1	2.623 (83%)	0.732 (79%)	0.737 (73%)	0.754 (29%)	0.148 (19%)	0.148 (49%)	0.147 (12%)	0.148 (16%)
Phase 2	0.474 (15%)	0.157 (17%)	0.242 (24%)	1.356 (52%)	0.512 (66%)	0.103 (34%)	0.770 (64%)	0.469 (51%)
Phase 3	0.073 (2%)	0.035 (4%)	0.026 (3%)	0.482 (19%)	0.108 (15%)	0.043 (14%)	0.272 (23%)	0.281 (31%)
Sequential Overhead	-	0.001	0.001	0.001	0.008(1.0%)	**0.008(2.5%)**	0.014(1.2%)	0.014(1.6%)
Total	3.171	0.925	1.007	2.593	0.776	0.301	1.203	0.912
Speedup	1	**3.43**	3.15	1.22	4.08	**10.53**	2.64	3.48

factor (RTF), parallel implementations provide a higher recognition accuracy. For example, for an RTF of 0.4, accuracy improves from 44.5% to 41.6% WER going from a multicore implementation to manycore implementation.

Our implementations are structured to be scalable. As shown in Table 5.7 the sequential overhead in our implementations was measured to be less than 2.5% even for the fastest implementation. Also seen in Table 5.7, the fastest algorithm style differed for each platform. Synchronization using the aggregation technique has an overwhelming overhead, despite using highly optimized software routines. The propagation technique, in comparison, had strictly better results. However, our first propagation implementation on GTX280 using global atomic operations had severe atomic-memory-access conflicts and performed worse than Core i7. The issue was resolved by using local atomics operations [4].

The GTX280 has a logical SIMD width of 32. The implementation on GTX280 benefited significantly from evaluation by arc, re-gaining the lost efficiencies seen in Figure 5.26, making propagation by arc the fastest algorithm style. However, SIMD has not been applied to the implementation on Core i7, since the overhead of coalescing control flow exceeds the benefit comes from 4-wide SIMD. Thus, the fastest algorithmic style on Core i7 was propagation by state.

5.1.6 Productivity Concerns

The ASR *pattern-oriented application framework* addresses many of the productivity concerns of ASR domain experts. It provides extensive background information in two aspects of the ASR application, the concurrency opportunities and the suitability of exploiting them on a highly parallel manycore microprocessor. It also reduces the number of lines of code that are required to be written to deploy an ASR-based application, which reduces the number of potential bugs in the development process.

The extensive background information on concurrency opportunities in the ASR application is presented in the *application context* component of the *pattern-oriented application framework*. As demonstrated in Section 4.4.1, the *application context* exposes the application concurrency in ASR using the parallel programming patterns described in Section 3.2.2. Clarifying the concurrency opportunities with parallel programming patterns brings forth a set of background knowledge in the solution techniques and their trade-offs in implementing an ASR based application. This reduces the time needed to acquire the necessary information to make design choices. For this reason, the *application context* improves the productivity of the users of the ASR application framework.

The *software architecture* of the ASR application framework is described in Section 4.4.2. This section clarifies the structure of a parallel implementation of an ASR inference engine that can effectively take advantage of the concurrency opportunities, and specifies the trade-offs and solution techniques on the GPU. This improves the productivity of the user of the application framework by providing assistance with navigating the software architecture of the ASR framework.

The *pattern-oriented application framework* for parallel programming incorporates all of the techniques that are described in sections 5.1.1, 5.1.3, 5.1.4, and 5.1.5 in a *reference*

implementation that is immediately available for ASR experts to use. For ASR experts, the speech inference process uses a standard HMM forward-backward implementation for interpreting speech (as seen in Section 4.4.2). Adaptations to different usage scenarios involve the tuning of the acoustic features extraction techniques, as well as varying the methods with which acoustic models and language models are trained.

During the process of developing multiple ASR-based applications [44, 96], we observed that the most frequently adapted module is the observation probability computation. The observation probability computation estimates the instantaneous match between the input and the acoustic model at a point in time. For different usages scenarios, the variations in the acoustic model affect the acoustic model file format and the observation probability computation routines. While the number of features in the feature vector extracted from the input may vary, the basic mechanism of one feature vector per time step is preserved. Based on this observation, a *pattern-oriented application framework* with an *extension point* for adapting the observation probability computation was constructed to allow ASR domain experts to customize the observation probability computation without having to implement the HMM forward-backward pass in the inference process (as described in Section 4.4.2).

The *extension point* component of the *pattern-oriented application framework* greatly improves the ASR domain expert's productivity in two ways. Firstly, the *extension point* focuses the interactions between the plug-ins and the framework through clearly defined, narrow application programming interfaces (API). This provides a clean abstraction of the application framework for the developers of the plug-ins to target. The clearly defined, narrow APIs reduce the potential for bugs caused by mis-understanding of the interface and reduce the potential for problems to appear during the integration process. Secondly, the *extension point* reduces the number of lines ASR domain experts have to implement to produce an efficient ASR-based application. This reduction is significant: to arrive at a working ASR-based application, using the *pattern-oriented application framework* reduces the effort from developing a ten-thousand-line application for the entire speech recognition engine to a few hundred lines of plug-in customization.

To further support the productive construction of ASR-based applications, the *pattern-oriented application framework* also provides other extension points, as listed below:

1. File input format: Data format for the different components of the Recognition Network, including the acoustic model, pronunciation model, and language model

2. Pruning Strategy: Fixed pruning threshold or adaptive pruning threshold

3. Observation Probability Computation: The instantaneous match between the input and the acoustic model at a time step

4. Result output format: Data output format to target either the next step in processing or the accuracy checking script

The description of the extension point has been illustrated in more detail in Section 4.4.4. Chapter 6 will present their utilization in the deployment of different application usage scenarios.

5.2 Risk Analytics Application Framework

With the proliferation of algorithmic trading, derivative usage and highly leveraged hedge funds, there is increasing need to monitor financial risk in order to measure the severity of potential portfolios losses in real time. As risk management is now obliged to keep pace with the market, overnight runs of risk estimates no longer suffice. It is acknowledged that the ability to compute intra-day risk metrics is a highly desirable capability in a risk analytics engine [60].

There are many types of financial risks that a financial institution is exposed to, including:

- Market Risk: The market value of trading assets may fall
- Credit Risk: Debtors (including market counterparties, e.g. brokers, investment banks, and other securities dealers that serve as the contracting party when completing financial securities transactions) may not be able to meet their obligations, causing a loss to the bank
- Operational risk: Loss of money due to a failure in the banks infrastructure or internal controls[4].
- Liquidity risk: Banks may incur costs in raising cash to meet obligations that become due
- Business risk: A business may become less profitable because of changes in the market for the bank's services

This work examines the challenges facing the construction of application frameworks estimating the first two types of risks: market risk and credit risk (related to counterparties, or counterparty risk). While the work presented here analyzes the configuration of the risk analytics engine that are most sensitive in terms of execution efficiency, the integration of such analysis into an application framework is an on-going task.

5.2.1 Market Risk Estimation

Financial institutions seek quantitative risk infrastructure that is able to provide 'on demand' reporting of global financial market risk using the Value-at-Risk (VaR) estimates. VaR refers to the maximum expected loss that will not be exceeded under normal market conditions over a predetermined period at a given confidence level [91] (page xxii). It emerged as a distinct concept in the 1980s and was likely triggered by the stock crash of 1987, when academically trained quantitative analysts were in high enough positions

[4]Risks arises from failures in a bank's infrastructure could also have adverse effects on the market as a whole. Recent regulations banning "naked" market access is one move by the U.S. Securities and Exchange Commission (SEC) to protect the market from the operational risks the market is exposed to as a result of traders having "unfiltered" or "naked" access to exchanges or ATS [9].

to worry about firm-wide survival [91]. The metric became institutionalized in 1997, in response to the U.S. Securities and Exchange Commission ruling that public corporations must disclose quantitative information about their derivatives activity, when major banks started including VaR information in the notes to their financial statements. VaR was selected as the preferred approach for measuring market risk in Basel II [5], an international standard initially published in 2004 and revised in 2006, which banking regulators can use when creating regulations about how much capital banks need to put aside to guard against the types of financial and operational risks that such institutions may face.

While the use of VaR to measure market risk is standardized, the methodology for estimating it is not. One common approach for estimating VaR uses the Monte Carlo method (Section 5.2.3).

Major banks model the joint probability distribution of hundreds, or even thousands, of market risk factors, and consider hundreds of thousands of future market scenarios [160]. Market risk factors are financial exchange quoted instruments, which could potentially devalue the portfolio. Common examples of market risk categories include interest rates, stocks, commodities, credit and foreign exchange. Hundreds of thousands of future market scenarios must be computed to reduce the statistical error in the estimated VaR to below 0.1%.

A faster implementation can improve the responsiveness of risk management systems, enable risk analysts to perform more comprehensive VaR estimates on an *Ad hoc* basis (e.g. pre-deal limit checking), and can give financial institutions a competitive edge in algorithmic trading. It will also allow institutions to effectively step up to the more pervasive systematic stress testing standards being imposed by market risk regulators.

The ability to closely monitor and control an institution's market risk exposure is critical to the performance of trading units, which rely on complex risk analysis in order to structure and execute trades. Yet, despite tightening legal requirements on market risk estimation, the industry is still far from adopting a standardized and comprehensive framework for market risk analysis. In a recent survey conducted by the Global Association of Risk Professionals in conjunction with SYBASE [6], only approximately 5% of the firms currently perform complex risk analysis on their portfolios and global positions in real time while about 25% are able to perform real time analysis on individual trades. Over a third of these firms indicated that the current risk infrastructure was not fast enough for the needs of the business and that the gap between IT and business domain expertise is one of the major sources of dislocation in the deployment of risk management systems. This calls for a more comprehensive approach to the design and deployment of complex risk analytics with a stronger emphasis on using financial domain knowledge and expertise to enhance the software optimization process.

5.2.2 Counterparty Exposure Estimation

VaR is not the only risk metric being used in the banking industry. Large banks are also using techniques such as potential future exposure (PFE) as a metric to quantify the counterparty risk posed by fluctuations in market prices in the future during the lifetime

of the transactions in its portfolio. Counterparties are brokers, other investment banks and other securities dealers that serve as the contracting party when completing financial securities transactions such as over-the-counter (OTC) derivative contracts.

In an OTC contract, two parties agree on how a particular trade or agreement is to be settled in the future through signing a bilateral contract. It is usually from an investment bank to its clients directly, without going through an exchange where a well-capitalized financial institution could protect one side of a transaction when the other side defaults. In an OTC contract, both sides are exposed to the risk of their counterparty defaulting on the contract. According to the June 2010 statistics from the Bank for International Settlements, the total outstanding notional amount of OTC derivative contracts is $583 trillion [8], making OTC a significant source of counter party risk in the financial market.

PFE is a long-term credit risk metric that covers contracts with lifetimes that can be as long as decades. As defined by Dean here:

> PFE: the maximum credit exposures over a specified period of time calculated at some level of confidence. This maximum is not to be confused with the maximum credit exposure possible. Instead, the maximum credit exposure indicated by the PFE analysis is an upper bound on a confidence interval for future credit exposure.

In a PFE calculation, the exposure is computed with respect to each counterparty contract individually in a Monte Carlo based estimation process. The approach here is significantly more complex as compared to the approach described in Section 5.2.1, where risk is computed for the entire portfolio at the same time. For large financial institutions, a portfolio could have as many as one million trades. Each traded asset requires the simulation of 30-300 time steps over as many as 5000 market scenarios. Computing their exposure over longer time horizons is a computationally expensive process. Such computations are usually done over-night on computing centers, requiring hours of computation on scores of servers. As the number of global positions held by the bank increases, there is increasing need to improve computational efficiency and reduce the sprawling of computing centers.

5.2.3 The Monte Carlo Method

The Monte Carlo method is an approach where the solution to a problem is estimated by statistically sampling its parameter space with thousands to millions of experiments using different parameter settings. The analysis method was made practical by the advent of computers and was pioneered by Stan Ulam, John Von Neuman, and Enrico Fermi, the physicists first used the approach for neutron diffusion calculations in the field of high energy physics [15]. By statistically sampling a problem's parameter space and simulating the outcome of an experiment, we gain valuable insights into complex problems that may be impossible or impractical to solve analytically or iteratively using partial differential equation based approaches. The ease and intuitiveness of setting up the experiments makes the Monte Carlo method a popular approach [72].

The Monte Carlo method has the following properties that make it desirable for implementation on a high performance parallel computing accelerator such as the GPU:

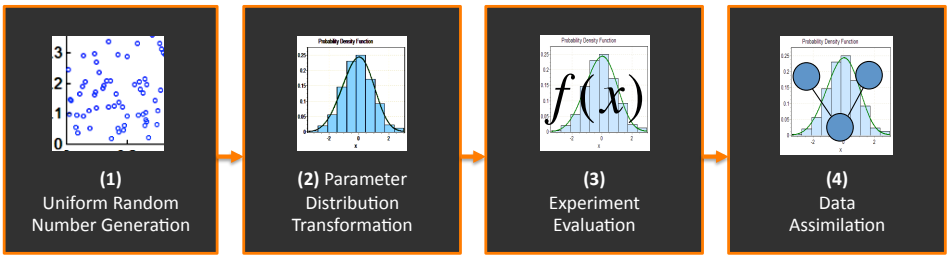


Figure 5.28: The solution structure for Monte Carlo based analysis

1. Experiments are independent and parallelizable: the approach assumes that experiments are independent and identically distributed (i.i.d.), such that the set of experiments provides a statistically valid sampling of the parameter space. This independence between experiments provides significant parallelization opportunities for GPU-based implementations.

2. Execution is computationally expensive: according to the law of large numbers, the statistical error (standard error) in the solution is proportional to the inverse square-root of the experimental size, i.e. to achieve 10x more precision in the result, one needs to run 100x more experiments. The GPU-based implementation can provide the necessary speedup to allow many problems to become computationally feasible.

3. Input specifications and results are concise: the Monte Carlo method takes a small set of experimental parameter inputs, generates thousands to millions of experiments, executes them, and assimilates the results as a single solution. There is a large amount of computation consumed with little input/output data transferred. This is ideal for GPU-based implementations, as input/output data has to be transferred between the CPU and the GPUs and all of the computation is performed on the GPU.

Figure 5.28 shows four key steps in a Monte Carlo based analysis. These steps can be optimized from several different perspectives, as demonstrated in Section 5.2.4 and Section 5.2.6. Step 1 generates uniform random numbers; Step 2 converts the uniform random numbers into statistical distributions that model the parameters in the scenarios to be simulated; step 3 evaluates the experiments; and step 4 assimilates the results. A typical Monte Carlo method based implementation executes one step at a time, with intermediate results written out to memory.

5.2.4 Efficiency Concerns in Market Risk Estimation

Seven key implementation techniques were identified in the construction of Monte Carlo based applications by studying the implementation process from three different perspectives, as discussed in [61]. The three perspectives for achieving efficient execution are the *task-centric*, *numerical-centric*, and *data-centric* perspectives.

The *task-centric* perspective focuses on the tasks being executed (Section 5.2.4). It involves applying two key optimization techniques. The first technique involves the minimization of the amount of necessary computation using task transformation and algorithm reformulation. The second technique involves the maximization of computational efficiency by identifying opportunities to leverage existing high-performance library components.

These techniques are applied to MC-VaR as a reformulation of the loss aggregation module to avoid matrix-matrix computations, and an analysis of the relative performance of various linear algebra routines to take advantage of more efficient types of cuBLAS (CUDA BLAS) library components.

The *numerical-centric* perspective focuses on the numerical properties of modules and their influence on the application performance[5] (Section 5.2.4). It involves three key optimization techniques. The first technique involves the selection of random sequences, which provide desirable convergence properties. The second technique involves the selection of random sequences that enable parallel generation of sequences. The third technique involves the selection of distribution conversion modules which preserve a uniformity property[6] of the random sequences.

These techniques are applied in MC-VaR as the selection of Sobol' quasi-random sequences for its uniformity property in accelerating convergence, the usage of a Sobol' generator with skip-ahead capability to enable parallel generation of random sequences, and the selection of the Box-Muller algorithm for its ability to preserve property A, a measure of uniformity, under distribution transformation proposed by Sobol'.

The *data-centric* perspective focuses on the data access properties of the algorithms and their influence on the application performance (Section 5.2.4). It involves two key optimization techniques. The first technique involves the elimination of redundant data transfers between small kernels. The second technique involves optimizations of the algorithm with respect to the memory hierarchy.

These techniques are applied in MC-VaR, first as the merging of random number generation and distribution conversion to eliminate redundant storage and transfer of intermediate results and the identification of key data alignment needs between parameter generation and risk estimation, second as the application of flexible data blocking strategies that maximally leverage the shared memory resources (or local scratch-pad memory).

The seven optimization techniques discussed above are applied to a VaR estimation application based on *delta-gamma* (Δ-Γ) approximation [61]. The Monte Carlo method is used to simulate the correlated market risk factor log returns (value changes) over a fixed time horizon. More precisely, each experiment k, where $k = 1..M$, is setup by first generating uncorrelated Gaussian Random Variables, X_{ik}, where $i = 1..N$. These are the uncorrelated Gaussian random perturbations away from the current market scenario, which is defined by the current value of the risk factors R_i. The input to a Monte Carlo based VaR estimation includes:

[5]Implementation techniques for efficient generation of Gaussian random numbers in CUDA are well studied, see for example [83].

[6]A uniformity property is a measure of how well sub-sequences of the random sequence cover the target distribution.

Statistical Parameters: the estimated means μ_i and Cholesky factor matrix $\hat{Q}$ of the covariance matrix $\Sigma = \hat{Q}^T\hat{Q}$ for the Gaussian distributed market risk factor log returns $d\ ln(R_i)$, where $i = 1..N$, $j = 1..N$, are obtained from historical time series;

Delta term: Δ_i, where $i = 1..N$, is the sensitivity of the portfolio to a change in the log of the market risk factor R_i;

Gamma term: λ_i and U_{ij}, where $i = 1..N,\ j = 1..N$, are the eigenvalues and orthogonal matrix of column eigenvectors of the semi positive definite matrix $\sum_{ij} \hat{Q}^T_{mi}\Gamma_{ij}\hat{Q}_{jn}$, Γ_{ij} is the sensitivity of Δ_i to a change in the log of market risk factor R_j. Γ is generally sparse and sometimes even diagonal, because the Δs from simple models of instrument prices typically only depend on one or a small number of risk factors.

Percentile for VaR evaluation: is typically either 1% or 5%; and

Time horizon: the duration of the simulation is typically at least a day and is measured in units of years.

The execution of each experiment k outputs the change of value of the portfolio dP_k using the (rotated) *delta-gamma* (Δ-Γ) approximation

$$dP_k = \sum_i \underbrace{\Delta_i Y_{ik}}_{delta} + \underbrace{\frac{1}{2}\lambda_i X^2_{ik}}_{gamma}, \tag{5.1}$$

where $Y_{ik} = \sum_j \mu_i + Q_{ij}X_{jk}$ are the correlated Gaussian random variables obtained from multiplying the (rotated) Cholesky matrix factor $Q = \hat{Q}U$ with the i.i.d. standard Gaussian random variables X_{ik}. Expressing the approximation in rotated random variables simplifies the expression in the gamma term [7].

By far the simplest model assumption is that portfolio losses are described by a joint normal distribution. But normal loss distributions do not exhibit the fatter tails of the loss distribution implied from historical data, where extreme events occur more frequently than the normal distribution describes [72]. Consequently normal loss distributions tend to severely underestimate potential losses. We relax this mathematically convenient assumption and present a VaR simulation approach, which generalizes to more realistic distribution functions. To this end, we also demonstrate the generation of student-t random variables using *Bailey's method* in the *data-centric perspective* section.

Task-centric Perspective

Statistical sampling of the parameter space with the Monte Carlo method involves running thousands to millions of independent experiments. With respect to the structure of an implementation of the Monte Carlo method, the experiment execution step often has the

[7] Readers more familiar with the approximation in original variables should refer to [72] for further details. The problem reformulation described in the next few pages relies on this rotated form of the approximation in which the rotated Cholesky matrix factor is now a full matrix and no longer upper triangular.

most amount of parallelism. For this reason, the evaluation of the delta term of the loss function (5.1) can be efficiently implemented as a dense matrix computation using existing well-parallelized cuBLAS library routines. This is illustrated in two steps, with Figures 5.29 showing the first step and Figure 5.30 showing the second step for the delta component of the loss function evaluation with N risk factors and $M >> N$ experiments.

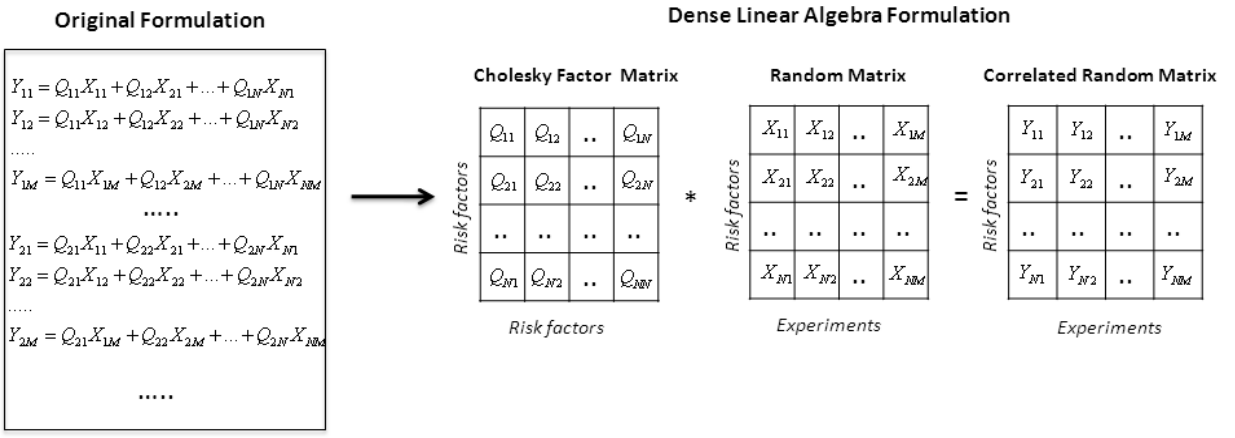


Figure 5.29: The correlation of random variables is re-factored as a dense matrix-matrix multiplication in order to use the existing well-parallelized cuBLAS library.

Step 1 forms the correlated random $N \times M$ matrix Y using matrix-matrix multiplication of the $N \times N$ (rotated) Cholesky factor matrix and the uncorrelated Gaussian random $N \times M$ matrix in $2N \times N \times M$ FLOPs. Step 2 computes the M vector of the delta component of the portfolio losses dp^{Δ} as a matrix-vector multiplication of Y and the N vector Δ in a further $2N \times M$ FLOPs.

Problem reformulation A key optimization reformulates the computation of the delta component of the portfolio loss function. The reformulated computation is illustrated in two steps. Firstly, the product of the (rotated) Cholesky matrix factor Q and the Δ vector is stored in a vector $q := \Delta^T Q$ using a call to the BLAS level 2 matrix-vector multiply function `Sgemv`. Figure 5.31 shows the pre-computation of q. Precomputation of q enables the bottleneck matrix-matrix computation to be replaced with a BLAS matrix-vector operation, reducing computation by $O(N)$, where N is the number of risk factors. Figure 5.32 shows the reformulated loss function evaluation.

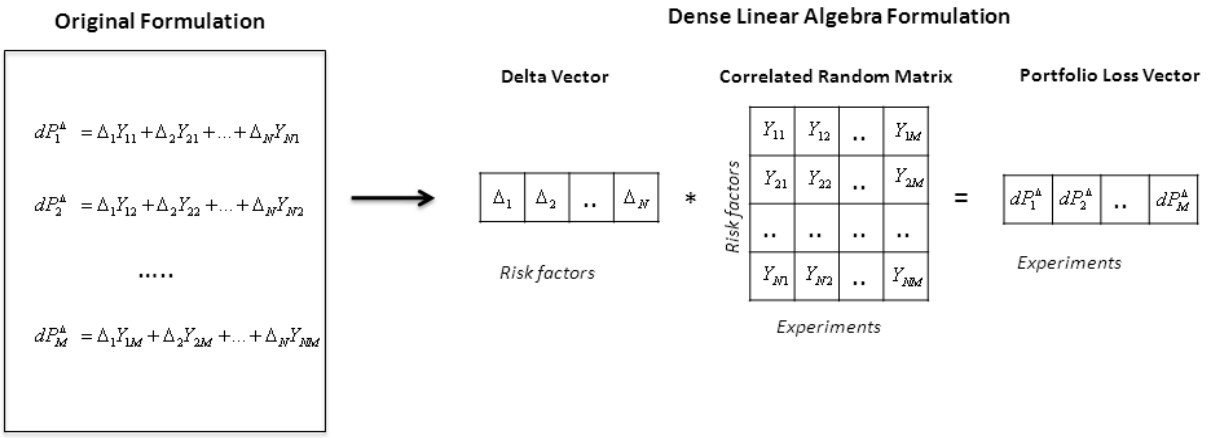


Figure 5.30: Loss function evaluation is also re-factored as a dense matrix-vector multiplication in order to use the existing well-parallelized cuBLAS library.

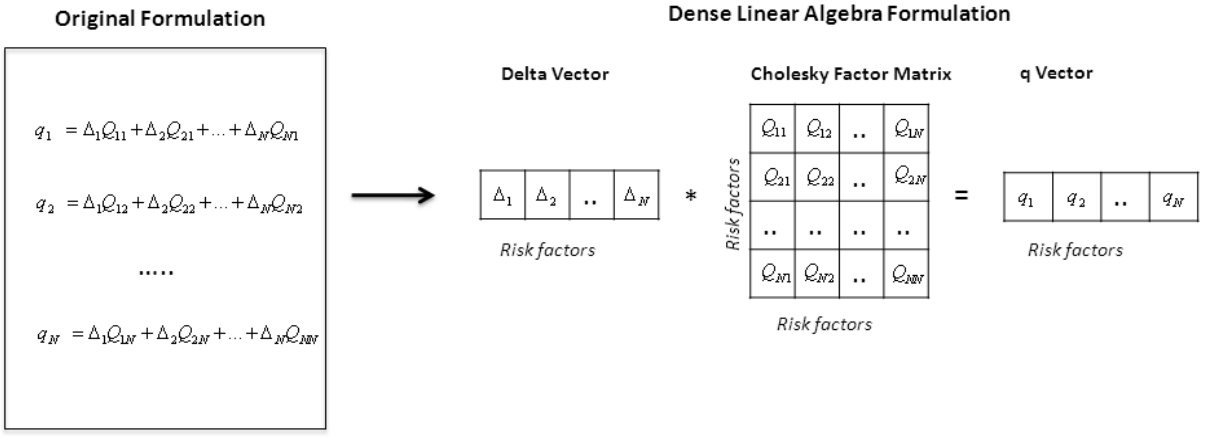


Figure 5.31: The precomputation of q is factored as a dense matrix-vector multiplication.

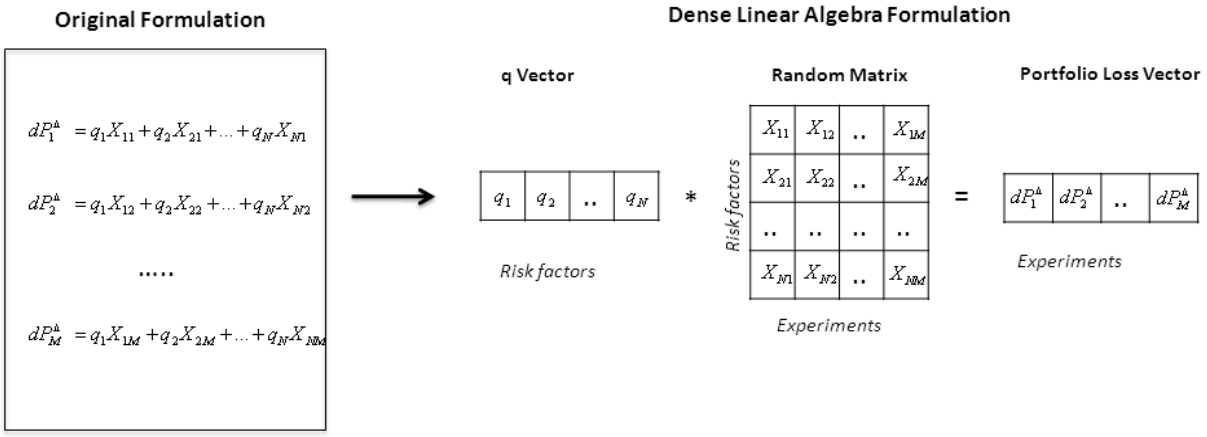


Figure 5.32: The reformulated loss function evaluation is also factored as a dense matrix-vector multiplication.

The Δ-Γ approximation of the portfolio loss is thus obtained using the optimized form

$$dP_k = dP_k^{\Delta} + dP_k^{\Gamma} = \sum_i q_i X_{ik} + \frac{1}{2}\lambda_i X_{ik}^2. \tag{5.2}$$

By reformulating the computation, the deterministic component of the delta term, q, computes in $2N \times N$ FLOPs and the delta component of the loss function computes in just $2N \times M$ FLOPs (excluding terms independent of M). This reformulation achieves a factor of $N + 1$ reduction in computation resulting from replacing the BLAS single precision general matrix-matrix multiplication kernel `Sgemm` with the matrix-vector multiplication kernel `Sgemv`.

The quadratic gamma term is evaluated by first scaling each row of X_{ik} by λ_i using N `Saxpy` evaluations totaling $N \times M$ FLOPs. This step is followed by M BLAS single precision vector-vector products (`Sgemv` evaluations), totaling $2N \times M$ FLOPs. The evaluation of the gamma term totals $3N \times M$ FLOPs. Table 1 summarizes the FLOP count for Δ and Δ-Γ loss function approximations with and without optimization.

	Standard	Reformulated	Speedup
Δ	$2N \times (N+1) \times M$	$2N \times M$	$N+1$
Δ-Γ	$N \times (2N+5) \times M$	$5N \times M$	$\frac{(2N+5)}{5}$

Table 5.8: A comparison of the number of FLOPs (excluding terms independent of M) required to compute the standard and reformulated Δ and Δ-Γ loss function approximations.

Performance results Table 5.9 compares the performance of MC-VaR on an Intel Core i7 processor and an NVIDIA GeForce GTX 480. Readers should refer to the later paragraphs on *experiment setup* for further system configuration details. The estimate assumes a normal distribution for the joint risk factor returns. $N_b = 23$ blocks of size M_b were generated using a portfolio of $N = 4000$ risk factors. With a maximum block size of $M_b = 32768$ (for the available memory), this number of blocks ensures that approximately 7.5×10^5 (753664) scenarios are generated to achieve 0.1% accuracy in the standard error of the loss distribution. Table 5.9 further shows the comparative effect of the optimized Monte Carlo algorithm on the time taken to evaluate Δ-Γ VaR on the GPU and CPU. The numbers in parentheses are the loss estimation portions of the execution times without the QRNG and distribution conversion steps.

Without problem reformulation, we see only a 7.76× speedup going from the baseline CPU to the baseline GPU implementation. By reformulating the problem, we see an additional 108× speedup for the loss estimation from baseline GPU implementation to reformulated GPU implementation. Reformulation enabled a 61.9× speedup for the Δ-Γ VaR estimation problem.

Timing (s)	Standard	Reformulated	Speedup
CPU	457 (384)	73.9 (1.17)	6.18× (328×)
GPU	58.9 (58.5)	0.951 (0.540)	61.9× (108×)
Speedup	7.76×(6.56×)	77.7×(31.5×)	**481**×(711×)

Table 5.9: The comparative timings(s) of the standard and reformulated Monte Carlo algorithm for evaluating Δ-Γ VaR on a NVIDIA GeForce GTX480 and an Intel core i7 CPU using the Box-Muller method with 7.5×10^5 simulations. The parenthesized values represent the times and speedup factors in just the loss function evaluation step.

Overall, the reformulated algorithm performs 77.7× faster on the GPU than the reformulated algorithm on the CPU.

Numerical-centric Perspective

A critical feature of the Monte Carlo method is the type of random number generator (RNG) used to generate the random experiments [83] in the first step of the solution structure illustrated in Figure 5.28. These generators must not only be extremely fast but also generate high dimensional sequences, which lead to fast convergence in the VaR estimate. Quasi-random number generators (QRNG), also referred to as low discrepancy sequence generators, are used extensively in financial applications[8] [71] and are the preferred choice for generating high dimensional sequences of uniformly distributed variables with respect to a measure of uniformity [151]. Many of these generators use a 'skip-ahead' method [140] to efficiently generate sequences on parallel architectures. As VaR estimation typically requires Gaussian random numbers, a distribution conversion function must also be used to transform the uniform random sequence to a Gaussian random sequence. This distribution stage should preserve the uniformity properties of the initial random sequence.

The Sobol' QRNG has recently been shown by Joe and Kuo [88] to satisfy Sobol's measure of uniformity (property A) in upto $N = 16900$ dimensions. Broadly put, this means a VaR estimate can be performed using up to $N = 16900$ market risk factors, provided the distribution conversion stage preserves this uniformity property. The best choice of distribution conversion function isn't always obvious and the decision should be approached on a case-by-case basis.

Sobol' Sequences The Sobol' generator produces a sequence of M N-dimensional uniformly distributed points of the form

$$X_{i,k} = 2^{-32} m_{i,k} \tag{5.3}$$

where i is the dimension (risk factor) index, k is the experiment index, $m_{i,k}$ is a 32-bit unsigned integer and $X_{i,k}$ is in the interval $[0, 1)$. The CUDA implementation generates the initial value of $m_{i,k}$ for each parallel thread based on Bratley and Foxs' 659 algorithm

[8]Monte Carlo methods which use QRNGs are referred to as 'Quasi-Monte Carlo' methods.

$$m_{i,k} = g_0 v_{i,0} \wedge g_1 v_{i,1} \wedge \ldots g_{31} v_{i,31} \tag{5.4}$$

where g_j is the j^{th} significant bit of the gray code `g = k & (k-1)` and $v_{i,j}$ is the j^{th} element of an unsigned integer direction vector for dimension (risk factor) i. If the number of experiments is $M = 2^n$, then only the first n vectors are used for $k < M$. 2^{32} is far more experiments than necessary so typically $n < 32$. An important feature of the CUDA implementation is that contiguous blocks of uniform distributed points are then generated in p-way parallel using the recursion relation

$$m_{i,k+1} = m_{i,k} \wedge v_{i,c_g}, \tag{5.5}$$

to 'skip-ahead' from element $X_{i,k}$ in the sequence to element $X_{i,k+p}$ in $\mathcal{O}(\log\ p)$ operations[9].

Another important feature of the CUDA implementation is that it is able to generate multiple blocks of random numbers, should $N \times M$ single precision variables saturate the device memory. This is achieved by starting each block from the last point in the previous block.

Distribution conversion The CUDA SDK 3.1 demonstrates three Gaussian distribution conversion functions, the Box-Muller method [33], the Moro interpolation method [115] and the inverse complementary error function (inverse erfc). We just consider the first two approaches here. The former takes a pair of uncorrelated uniform random numbers, from different dimensions of the Sobol' sequence, in the product of closed unit intervals $(0, 1) \times (0, 1)$ and uses the polar coordinate transformation to generate a pair of uncorrelated standard Gaussian variables. The Moro interpolation method, takes a single uniform random number in the open unit interval $[0, 1]$ and draws a standard Gaussian variables using a polynomial interpolation of the inverse cumulative Gaussian distribution function.

The convergence rate in the standard error of the portfolio loss distribution can be used as a metric for choosing the best distribution method. It is also advisable to compare the delta-VaR estimate with the analytic estimate. In general, the Δ-Γ VaR cannot be estimated analytically, of course, but having an understanding of how the standard error compares with the delta VaR estimate error can provide some intuition into what tolerance to choose for the standard error given a target VaR estimate error.

The results of the comparison between the two distribution functions are presented in Figure 5.33. For 4000 risk factors, approximately 1.5×10^6 scenarios are required to estimate the error in the simulated one-day portfolio delta VaR to be within 0.1% by using Moro's interpolation method. In contrast, by using the Box-Muller method, only half the number of scenarios (7.5×10^5 scenarios) is required when using to reach the same error bound. The study of the comparative effect of using single versus double precision arithmetic on the convergence rate is beyond the scope of this chapter.

[9] c_g is the index of the least-significant zero bit in g.

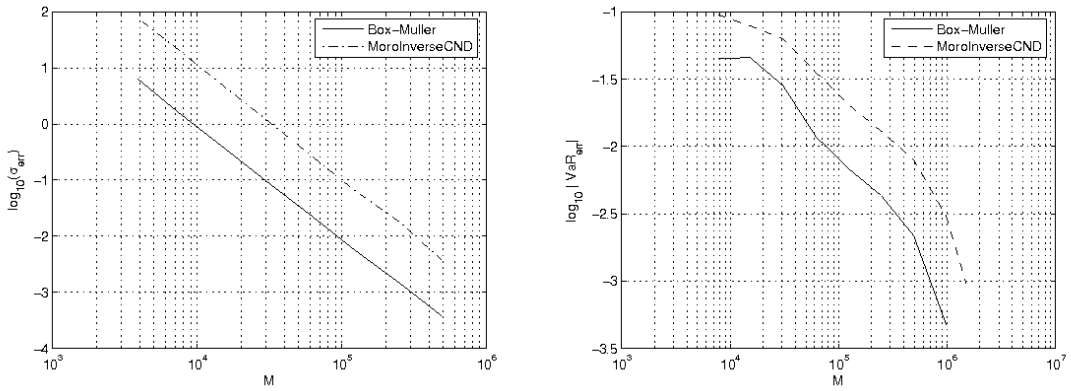


Figure 5.33: (Left) A comparison of the standard error (%) in the portfolio loss distribution using Moro's interpolation method and the Box-Muller method applied to Sobol' sequences. (Right) The corresponding error (%) in the simulated 1 day portfolio delta VaR (c=95%) monotonically converges to the analytic delta VaR (9.87%) with the number of scenarios. In single-precision arithmetic, approximately 1.5×10^6 or 7.5×10^5 scenarios is sufficient to estimate the delta-VaR to within 0.1% when using Moro's interpolation method or the Box-Muller method respectively.

Performance results The CUDA implementation of the Box-Muller method transforms $4000 \times 7.5 \times 10^5$ uniform quasi-random numbers in $0.282s$ and accounts for 29.6% of the total time of the reformulated algorithm.

The primary criteria for choosing the Box-Muller method is that the standard error converges twice as fast in single precision as when using Moro's interpolation method applied to the same sequence of uniform quasi-random numbers.

Data-centric Perspective

In Monte Carlo based risk estimation, thousands to millions of experiments are generated and executed. The computation takes place according to the four steps shown in Figure 5.28. There is a significant amount of intermediate results that must be managed between the steps. Figure 5.34a shows a basic implementation, where the steps are executed one at a time. The amount of intermediate data can be 100s of MBytes. Storing them out to off-chip memory after each step and bringing them back in for the next step can be inefficient, especially when steps such as Step 1 and Step 2 require very little computation.

The need to maintain large intermediate result working sets is not required by the application. It is usually put in place to achieve functional modularity in large projects. To optimize the implementation for execution on GPUs, one must re-evaluate the software architecture trade-offs and work towards minimizing the number of data movements, which can dominate the execution time of an implementation.

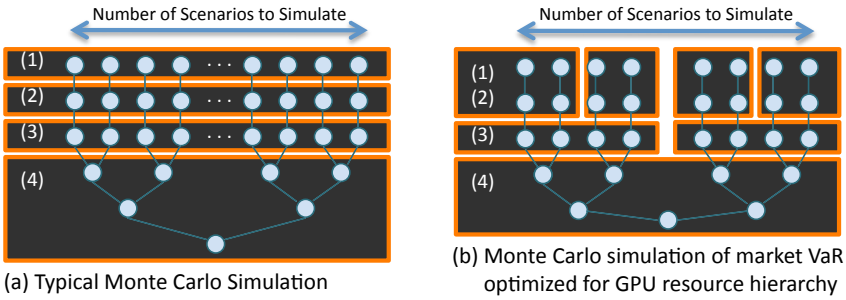


Figure 5.34: The solution organization of Value-at-Risk on the GPU.

Timing (s)	Standard (Separate)	Optimized (Merged)	Speedup
Box-Muller (step1)	0.128	0.156	2.63x
(step2)	0.282		
Bailey (step1)	0.128	0.441	1.16x
(step2)	0.384		

Table 5.10: The benefits of merging Step 1 (uniform random number generation) and Step 2 (distribution conversion)

Kernel Merging In the case of the VaR implementation, merging the uniform random sequence generation and the distribution conversion steps provided significant performance boost. Figure 5.34b illustrates the case where the distribution conversion step executes in place as soon as the uniform random values are generated. By converting the uniform random values while they are still in the GPU's register files on-chip, we save the execution time associated with a set of round trip memory transfers as compared to writing out the results to device memory and reading them back.

Performance results Table 5.10 illustrates the performance impact on a NVIDIA GeForce GTX 480 from applying kernel merging to two similar methods - the Box-Muller and Bailey's methods[10]. A 2.63× speedup was achieved through merging steps 1 and 2 for the Box-Muller method. This optimization, however, can be sensitive to the amount of context each thread must maintain.

The optimization is found to be effective for the Box-Muller method, but not as effective for Bailey's method, even though their functional form is very similar. The Bailey's method contains two extra parameters for producing distributions with higher weights for less likely

[10]Bailey's method [21] also makes use of a polar coordinate transformation of uniform random variables. Unlike the Box-Muller method, however, it produces student-t distributed random variables which give more conservative VaR estimates [90]. Performance benchmarks using the Bailey's method are based on our own implementation as is it not available in the CUDA SDK.

events (or "fatter tails" for the distribution). These extra parameters require more registers in the computation. Specifically, the merged Bailey's method resulted in a modest 1.16× speed up over two separate steps. This indicates that register spilling is causing responsible for the reduction in performance improvements on the GPU architectures.

Data Blocking In Step 3, as explained in Section 5.2.4, we are leveraging the cuBLAS library for experiment execution. For VaR estimation, 7.5×10^5 experiments are used. With thousands of risk factors being considered in each experiment, the experiments must be executed in batches to allow all operands and results of the cuBLAS routines to reside in a GPU's global memory. Sharing the 1GB GPU global memory with the Monte Carlo based VaR estimation engine, one can fit 2^{14} experiments within one batch. The batches of experiments are executed and the outcome of each experiment is saved for later use. Figure 5.34b illustrate the segmented Steps 1-3 that is used to block the batch processes for optimized data reuse[11].

Results After Addressing Efficiency Concerns

Our baseline GPU NVIDIA GeForce GTX480 graphics processing unit (GPU) implementation of MC-VaR is a straightforward port from the CPU implementation and has a 7.76x speed advantage over an eight-core Intel i7 Q920 central processing unit (CPU) based implementation. By reformulating the VaR estimate to reduce the amount of computation, we achieved a 61.9x speedup. Use of a Box-Muller algorithm to convert the distribution gives an additional 2x speedup over interpolating the inverse of the Gaussian distribution function. We merged data-parallel computational kernels to remove redundant load store operations leading to an additional 2.63x speedup. Overall, we have achieved a speedup of 169x against the baseline GPU implementation, reducing the time of a VaR estimation with a standard error of 0.1% from minutes to less than one second.

Experiment Setup The benchmark results for the CPU implementation are measured on an Intel i7 Q920 eight-core CPU, with 8MB L2 cache and 2.67GHz clock frequency. The estimation engine is compiled with Intel ICC version 11.1 (072). We use the multi-threaded BLAS kernels implemented in the Intel Math Kernel Library (MKL) version 10.2.5 (update 35).

The benchmark results for the GPU implementation are measured on an NVIDIA GeForce GTX 480 GPU with 1.5GB of global memory. The GTX480 has 15 multiprocessors at 1.4GHz clock frequency, each with dual issue 16-way vector arithmetic units. The CUDA programs are compiled with NVCC release 3.1 and use BLAS routines available in the cuBLAS library version 3.1.

We have chosen algorithmically equivalent baseline CPU and GPU implementations, which provide negligible differences in the intermediate results and VaR estimates. In other words, the baseline implementations only provide a transparent reference point from which

[11]Step 4 does not need to be batched, because its data set comfortably fits in the GPU's global memory.

to trace back any differences in output between the CPU and GPU after subsequent code modifications. They are not fully optimized for performance.

The baseline CPU implementation first generates Sobol' sequences using our own OpenMP parallel implementation of the Sobol' QRNG adapted from a publicly available C++ code written by Joe and Kuo [88]. This implementation conveniently provided pre-computed direction vectors to dimensions beyond our requirements. Intel's MKL provides an optimized implementation of the Sobol' QRNG (with merged distribution transformations) based on Bratley and Fox's [34] algorithm 659. This implementation, however, is only pre-configured to generate quasi-random numbers in dimensions of up to 40, although it does allow for precomputed direction vectors in higher dimensions (e.g. from [88]) to be externally referenced.

Sobol' sequences are transformed into normal random variables using a C implementation of Moro's Inverse Cumulative Distribution Function (ICDF) provided, for comparative reasons, in the NVIDIA CUDA SDK 3.1. The loss function is then evaluated with calls to the Streaming SIMD Extensions (SSE) enabled MKL cBLAS kernels `cblasSgemm`, `cblasSgemv` and `cblasSaxpy`, and the result is stored in a loss vector.

We observed that the baseline CPU implementation spends approximately half the time generating the random matrices (QRNG + distribution conversion), and half the time evaluating the loss function. Finally, the moments of the loss distribution are estimated by sorting the loss vector of length M using `qsort` from `cstdlib` in $\mathcal{O}(MlogM)$ operations on average. This is a negligible step compared to the BLAS computations which are at least $\mathcal{O}(MN)$.

The baseline GPU implementation first generates uniform quasi-random numbers with the "embarrassingly parallel" Sobol' QRNG provided in the NVIDIA SDK 3.1. The sequence is transformed using our own optimized version of Moro's ICDF which makes extensive use of computation blocking and shared constants in the on-chip shared memory. The loss function is then evaluated using `cublasSgemm`, `cublasSgemv` and `cublasSaxpy` kernels. At the final step, the loss vector is copied from device to CPU memory and sorted using `qsort` from `cstdlib`.

Overall Performance Results The overall speedup from a three-stage optimization of the CUDA implementation of the Δ-Γ VaR model is presented in Table 5.11. The columns in the table represent the three steps of execution in the VaR estimation and the table content specifies the absolute and proportional timings of the steps.

The baseline GPU implementation is able to exploit the absence of cross thread communication in the Sobol' QRNG and Moro's ICDF by leveraging the faster native transcendental functions to significant effect. There is a 22.2× speedup for the Sobol' quasi-random number generation and a 25.8× speedup for Moro's ICDF. With a 7.97× speedup, the loss function evaluation step also benefits from being mapped to the GPU, although it becomes the performance bottleneck in the baseline GPU implementation.

We optimize the VaR computation on the GPU in three ways: using problem reformulation, module selection, and kernel merging. We briefly state the effect of these three optimizations here and explain them in detail in the next section:

Timing (s)	QRNG	Distribution Conversion	Loss Evaluation
Baseline GPU	0.257 (0.22%)	0.564 (0.48%)	117 (99.3%)
Problem Formulation (GPU)	- (13.5%)	- (29.7%)	1.08 (56.8%)
Module Selection (GPU)	0.129 (13.6%)	0.282 (29.6%)	0.540 (56.8%)
Kernel Merging (GPU)	0.156 (22.4%)		- (77.6%)
Speedup	5.27x		217x
Total Speedup	169x		

Table 5.11: The columns of the table illustrates the different steps in a Monte Carlo simulation. The top half of the table shows the run time in seconds after each respective optimizations. The percentages in parentheses on each row add up to 100%. They indicate the proportions of the algorithm steps. The bottom half of the table illustrates the GPU speedup for the Δ-Γ approximation with 4000 risk factors, simulated to achieve a standard error in the normal loss distribution within 0.1%.

- *Problem reformulation:* By reformulating the algorithm using task-centric techniques as described in the *Numerical-centric Perspective* section, we are able to obtain a 108× speedup in the loss function evaluation - the bottleneck in the baseline implementation. This amounts to an overall speedup in the VaR estimate of 61.9×.

- *Module selection:* Using the numerical-centric techniques to choose the Box-Muller method over Moro's ICDF, as described in the *Numerical-centric Perspective* section, we selected the module that gives the faster numerical convergence rate in the standard error. We were able to reach the same VaR estimation accuracy with nearly half the number of simulations. This provides another 2× speedup.

- *Kernel merging:* Using the data-centric techniques, we merge the Sobol' and distribution conversion steps, as described in the *Data-centric Perspective* section, to remove a pair of redundant load and store operations from the computation of each distribution conversion. This reduces the QRNG generation and distribution conversion execution time by 2.63×.

After these optimizations, we have enabled a 169× faster implementation compared to a GPU-based baseline solution, and 1311× faster implementation compared to a CPU-based baseline solution. This illustrates that using a GPU-based implementation may provide some speedup over a CPU-based implementation, but relying on the platform advantage alone would overlook significant acceleration opportunities.

5.2.5 Productivity Concerns in Market Risk Estimation

A Monte Carlo based VaR estimation is highly modular. Each of the four steps in the Monte Carlo method that was described in Section 5.2.3 can be replaced with functional variants to form new Monte Carlo based applications. While an application framework for Monte Carlo based VaR estimation on the GPU remains a topic of on-going research, the efficient implementation described in Section 5.2.4 can serve as a *reference implementation*. With this reference implementation, a framework user can immediately experience the performance achievable for VaR estimation on the GPU.

In terms of the other components of a VaR application framework, the *application context* is outlined in Section 5.2.1. The concurrency in the VaR estimation application can be described as an instance of the Monte Carlo Methods pattern (see Appendix A). Clarifying this association brings forth a set of background knowledge in the solution techniques and their trade-offs, which reduces the time needed to acquire the necessary information to make design choices. For this reason, the *application context* improves the productivity of the users of the VaR application framework.

The *software architecture* of the VaR application framework is described in Section 5.2.3. By clarifying the structure of a parallel implementation of a Monte Carlo based estimation engine, a VaR application expert can effectively take advantage of the concurrency opportunities in the VaR estimation problem, and learn about the suitability of the solution techniques to exploit the concurrency opportunities on the GPU. This improves the productivity of the VaR application framework users by providing assistance with the navigation of the software architecture of a VaR application framework.

The *extension points* are described here in order to illustrate what an application developer can expect from a Monte Carlo based VaR Application Framework.

1. The *Random number generation (RNG) extension point*: while we have found that the Sobol' quasi-random number generator is well-suited for single time-step short time-horizon VaR computations, larger scale risk analytics approaches often use a Mersenne twister [109] based pseudo RNG for its long periods. An extension point for RNG allows different generation approaches to be applied.

   There are two requirements that a RNG implementation must satisfy to become a plug-in for this extension point: 1) It must support skip-ahead to allow a thread of execution to be able to fast forward to a particular element in the random sequence; 2) It must be able to save and reload its internal states, i.e. the RNG step must be allowed to be swapped out of parallel contexts in a highly parallel implementation and restore the execution where it left off at a later time. For example, to satisfy the first requirement, an RNG routine interface should contain an index to which the implementation will skip to for the next element to be generated; to satisfy the second requirement, a RNG routine interface should contain a set of internal states to be loaded for the next elements to be generated.

2. *Parameter distribution transformation extension point*: This is a crucial extension point that is expected to be frequently customized in deployment. The appropriateness of parameter distribution is conditional on the type of financial instruments

for which the risk is being analyzed. In the reference implementation presented in Section 5.2.4, we have interchangeably used the Moro interpolation method and the Box-Muller method, which produce Gaussian distribution functions, as well as the Bailey's method, which produces the student t-distribution with more emphasis on the occurrence of unlikely events (sometimes referred to as "fatter tails").

There are some optimization opportunities in the merging of the plug-ins specified at the *RNG extension point* and the *parameter distribution transformation extension point*, as demonstrated in the *data-centric perspective* shown in Section 5.2.4. The approach to exploit this opportunity is a topic of on-going research in programming frameworks for parallel computing.

3. *Experiment evaluation extension point*: The experiments in the reference implementation are evaluated using highly optimized dense matrix operations from the cuBLAS library, a BLAS library for CUDA. For estimating VaR on different portfolios of financial instruments, there are other variants besides the *delta-gamma* (Δ-Γ) approximation. For simpler cases, one could use the first-order *delta* (Δ) approximation. For more complex cases, one could use the third-order approximation. The decomposition of the evaluation into matrix routines may also change as new faster dense matrix libraries emerge. The *experiment evaluation extension point* allows new plug-ins to be developed without re-implementing the full VaR estimation infrastructure.

4. *Data assimilation extension point*: This extension point handles the analysis of the experiment results. Different confidence intervals could be used for VaR estimates, and different data output formats could be used given the needs of the overall application.

A VaR pattern-oriented application framework with the four extension points as described above highlights the key steps in a Monte Carlo based VaR estimation. With these four extension points, an application developer can adapt the framework to a variety of the VaR estimation usage scenarios by specifying only the plug-ins that *differ* from a standard implementation. This significantly reduces the number of lines a VaR application expert have to implement to produce an efficient VaR estimation-based application. Specifically, using the *pattern-oriented application framework* reduces the effort from developing a ten-thousand-line application to implementing a few hundred lines of plug-in customization.

While the implementation of this application framework is on-going research, we see that its implementation can help the VaR application framework users productively understand the application concurrency opportunities and the software architecture of reference implementation. It can greatly reduce the number of lines of code required to implement a variant of the VaR application, which will also reduce the number of potential for bugs in the implementation process, resulting in improvements in productivity.

5.2.6 Efficiency Concerns in Counterparty Exposure Estimation

The challenges in the acceleration of Potential Future Exposure (PFE) calculations on the GPU were explored[12]. Compared to the VaR implementation that was explored in Section 5.2.4, estimating PFE for a portfolio is significantly more computationally intensive. The computation can be seen as calculating Present Values (PV) of future gains/losses along three dimensions for a portfolio:

1. Over different trades: there can be as many as 5×10^5 to one million trade-able assets in a portfolio, such as Bonds, Options, Swaps, Swaptions, etc.

2. Over different paths: each trade is simulated with as many as 5000 future market scenarios or paths

3. Over different time steps: for each trade on each path, it is simulated for 30-300 time steps into the future

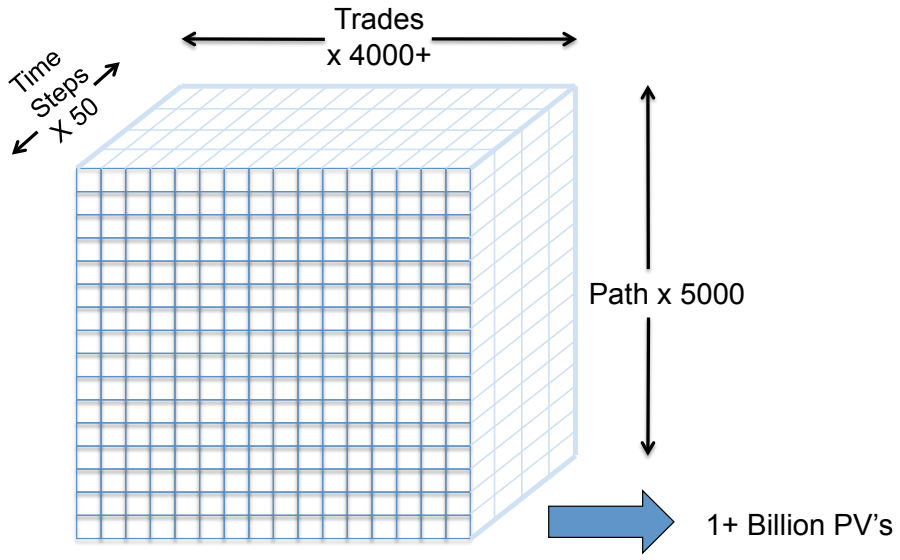


Figure 5.35: The cube (three dimensional matrix) of Present Values (PVs), or Monte Carlo simulation results, in a Potential Future Exposure (PFE) application.

For any given time step of the market scenario, the value each of the trade is computed. To normalize the value of the trades to a present day value, the Present Value is computed by depreciating future value with the interest rate curves specified in the market scenario. Figure 5.35 represents these computations as a three-dimensional matrix (also known as the PV Cube). Even for a nominally sized PFE calculation with only 4000 trades, a billion

[12]These experiments were jointly conducted with an industry partner based on a production code base that is currently in use. The industry partner is a large global financial information company

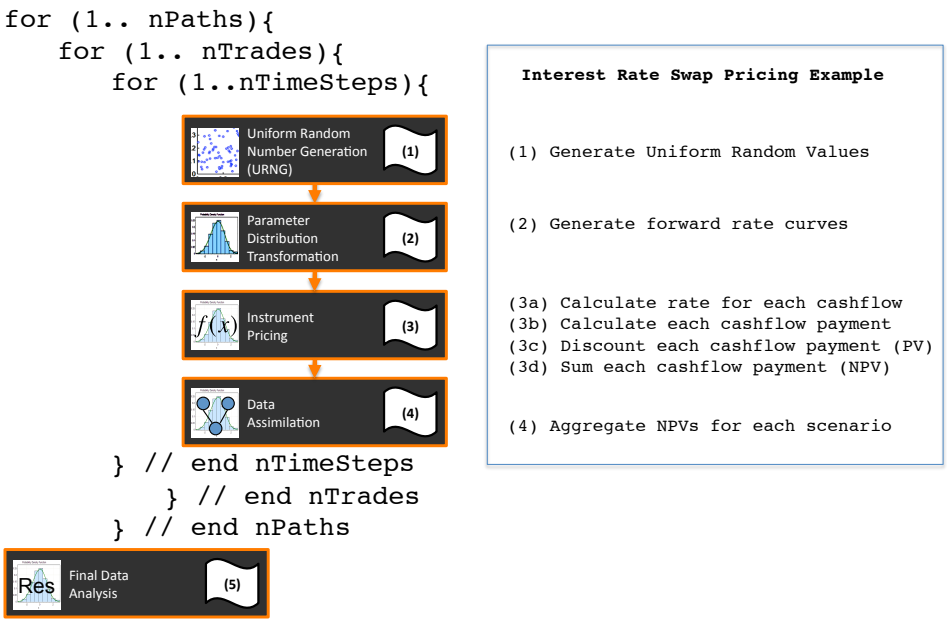


Figure 5.36: The task-centric perspective of a potential future exposure (PFE) estimation.

PV computations are required. Computing the PVs in this matrix consumes over 90% of the total execution time of a PFE application. The rest of the execution time is spent on loading market and trade data, as well as performing calibrations.

Task-centric Perspective

From the task-centric perspective, the 3D matrix of PVs is computed in a set of nested *for* loops. Among the *for* loops, all trades can be processed independently, and all paths can be processed independently. However, at the time step level for certain trade types such as options, it is more efficient to be computing the time steps in chronological order, as early termination of the computation may be possible depending on the market scenario.

Figure 5.36 illustrates the organization of the tasks. On the left is the triple nested loop for computing the PVs in the 3D matrix of a computation. On the right is an example of an interest rate swap (IRS) trade, where the computation for the specific trade type is listed and labeled with lines (1-4).

More specifically, for each path in each trade, a typical implementation of the Monte Carlo Engine creates a task that goes through each time step. The task first generates uniform random values (line 1), and then creates a scenario going forward into the future (line 2). In the case of estimating PFE for an IRS, an interest rate scenario (i.e. forward rate curves) is constructed according to the perturbations that the random values generated. Once the path scenario is generated, the trade's behavior is simulated (lines 3a-3b). In the case of the IRS, this involves calculation of the cash flow at each time step, as well as the associated cash pay out. The value computed at each time step is normalized to the same

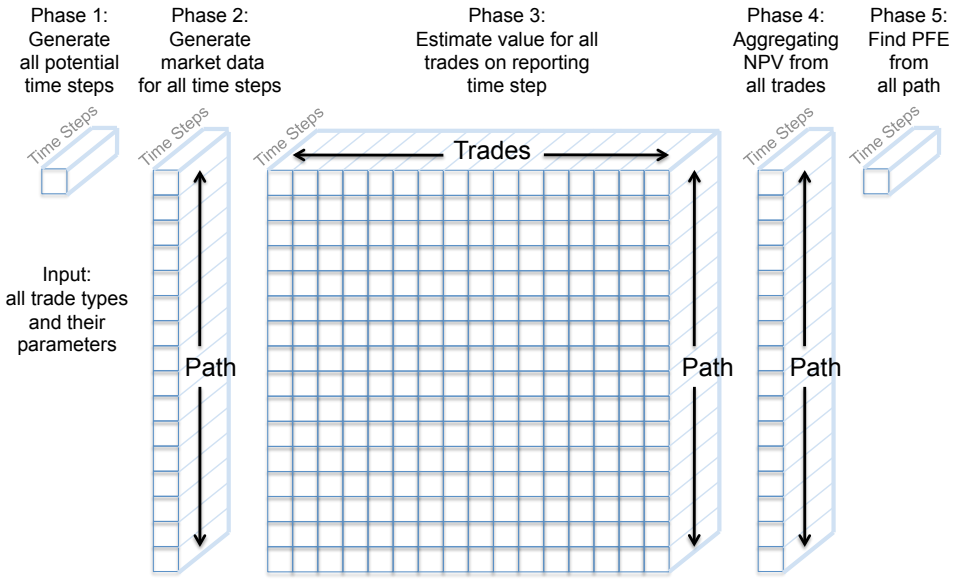


Figure 5.37: The data-centric perspective of potential future exposure (PFE).

point in time (line 3c). In this case, they are discounted to the present day, providing the present value (PV) (line 3d). The PV is then accumulated into the value of the trade given its behavior in all of the time steps, thus providing the Net Present Value (NPV) for the trade (line 4).

The parallelization techniques are explored in the result section. There are two caveats to keep in mind:

1. In computing the PFE of a portfolio of trades, all trades may have slightly different parameterization, meaning that their behavior is likely to differ across the time steps.

2. Path generation must be coordinated among the trades, as the same scenario should be applied to all trades in order to arrive at the behavior for a portfolio for that scenario.

Data-centric Perspective

Taking the task-centric perspective into account, one can also analyze the PFE workload from a data-centric perspective. Figure 5.37 illustrates the amount of results generated across different phases of execution (from left to right). It highlights that the majority of the work in the Monte Carlo-based PFE estimation is spent in Phase 3 working on pricing the trade-able assets.

In Phase 1, a sequence of time stamps is generated based on trade types and parameters. These time stamps represent important reporting dates for the portfolio and its trades. Based on the time stamps of importance, the market data for the scenarios can be generated in Phase 2. Note that not all scenarios have to be generated at the same time. In fact, the process illustrated in Figure 5.36 works with one path at a time, and it works through all

of the trades for that scenario before starting work on the next one. In Phase 3, the NPVs are computed for each trade in each path. In Phase 4, the NPVs are accumulated across all of the trades in order to summarize the value of the portfolio in each path over time. The final step computes the PFE based on the NPVs over the thousands of simulated paths.

Results

The benchmark results for the CPU implementation are measured on an Intel Core2 Q9300 quad-core CPU, with 6MB L2 cache and 2.50GHz clock frequency. Microsoft Visual Studio was used to compile the code. The benchmark results for the GPU implementation are measured on an NVIDIA GeForce GTX 260 GPU with 896MB of global memory. The GTX260 has 12 multiprocessors running at 1.24GHz clock frequency, each with 8-way vector arithmetic units. The CUDA programs are compiled with NVCC release 2.2 [4].

The production code for the industrial scale application contains more than 100k lines of C++ source code. The infrastructure has been developed over many years and was optimized for maintainability. A proof of concept (POC) Monte Carlo Engine of 5k lines of C++/CUDA was constructed based on the production code base in order to allow fast exploration of the software architecture design space to occur.

Using the POC Monte Carlo Engine, for a test set with 1000 interest rate swaps, 3000 scenarios, and 50 time steps, a speed up of 750x was achieved. Figure 5.38 illustrates the different approaches that were experimented with here.

On the left in Figure 5.38, the software architecture that was used in the serial CPU implementation can be seen. Only the financial instrument (trade) pricing step was mapped to the GPU. In this case, the instrument pricing routine was parallelized over 50 time steps. This approach maximally preserves the Monte Carlo engine setup, but resulted in a slow down in the execution time, as there was not enough concurrency or computation to speed up and compensate for the significant overhead in transferring the operands and results.

In the middle of Figure 5.38, the application is parallelized over the number of paths. At each time step, the paths (or market scenarios) are generated for all trades, and the PVs are computed one trade at a time across all paths. There is a 3000-way concurrency over the paths, and the market scenario generated for each path only has to be transferred to the GPU once. We are able to observe a 25x speed up over the original C++ baseline performance.

On the right in Figure 5.38, the application is parallelized over the number of paths and the number of trades. At each time step, the paths (or market scenarios) are generated for all of the trades. The PV computations are blocked to maximally leverage the shared memory resources local to each streaming multiprocessors (SM) in the GPU. Specifically, each thread block works on a trade, and each thread within the thread block works on a scenario. Such mapping of workload to the GPU enables a 750x speed up over the original C++ baseline performance.

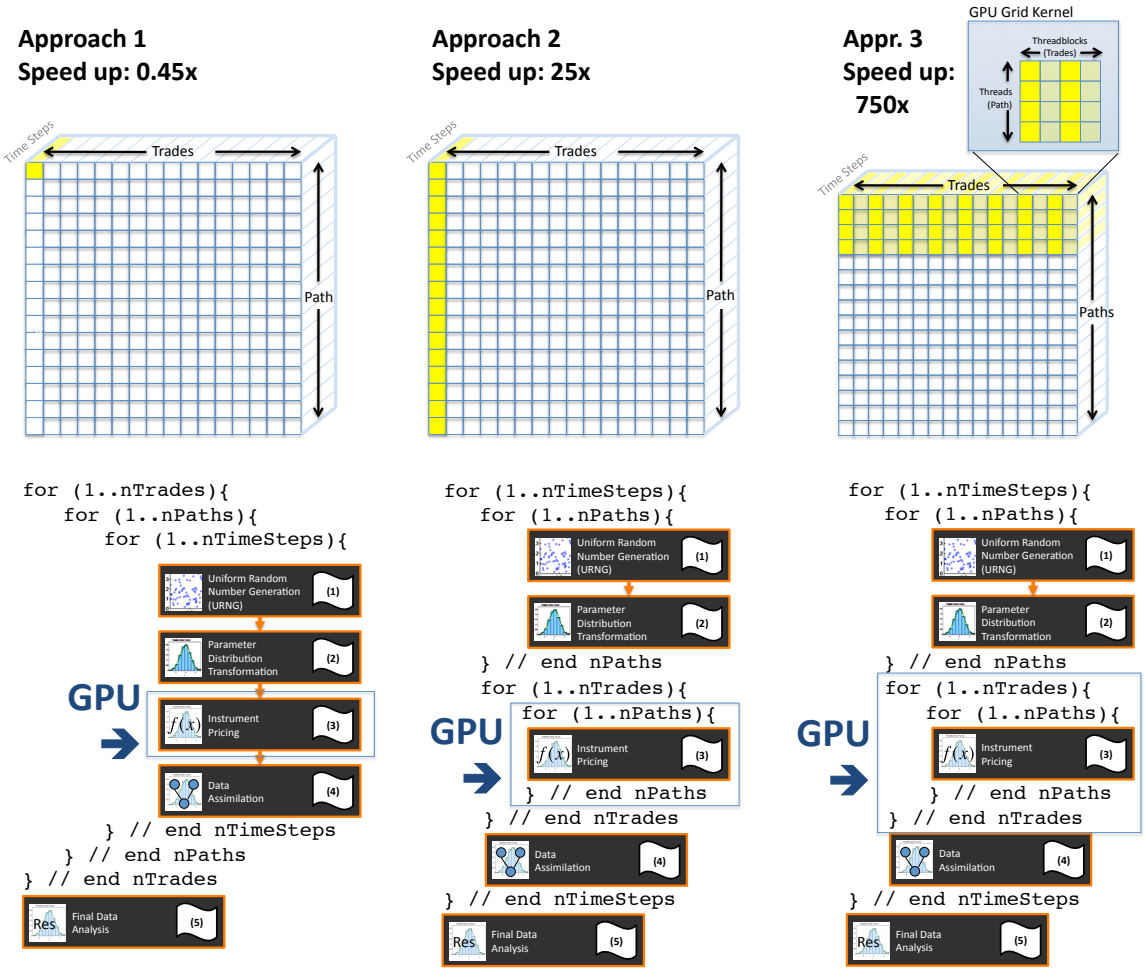


Figure 5.38: Three approaches to GPU-based PFE implementation.

5.2.7 Productivity Concerns in Counterparty Exposure Estimation

While a Monte Carlo based PFE pattern-oriented application framework using the GPU remains a topic of on-going research, the implementation demonstrated in Section 5.2.6 can serve as a *reference implementation*. With this reference implementation, one can immediately experience the performance achievable for the PFE application on the GPU.

The code base for counterparty exposure estimation contains domain knowledge from three domains of expertise in: 1) market models for scenario generation, 2) trade models and business logic for simulating the trades under different market conditions, and 3) calibration routines to characterize model parameters. These three domains knowledge are encapsulated in a code base of 100-thousand lines of code. The productivity concerns for a 100-thousand-line code base is focused on maintainability, as it is difficult for any one software developer to be aware of all of the parts of the code base at once. Maintainability here means that

the software architecture must be designed such that people with different domain expertise can each have a module to implement functions in their domain without jeopardizing the correctness of the overall application. If this is achieved, one can consider the software infrastructure maintainable.

In this spirit, the production code is already well architected with *extension points* available for each commonly modified entity, such as time step management, forward rate construction, and a variety of internal and external libraries for instrument (or trade) pricing. The interface for pricing instruments has been kept at the finest grain of one time step, one path, and one trade. While this is a very flexible interface, as was observed in the result in Approach 1 (left panel in Figure 5.38), it cannot be efficiently parallelized on a manycore platform.

The center panel in Figure 5.38 illustrates Approach 2, and the right panel in Figure 5.38 illustrates Approach 3. The implication for Approach 2 is that the pricing library interface must be modified to allow the specification of a range of paths to be concurrently priced. Similarly, the implication for Approach 3 (right panel in Figure 5.38) is that the pricing library interface must be modified to allow a set of paths as well as a list of trades to be priced concurrently. While there is on-going research investigating how much the actual pricing routines must be modified to enable the alternative interface, approaches 2 and 3 shows that the production Monte Carlo Engine must be modified to allow more flexible packed execution of the pricing engine calls to take place.

In terms of the components of a PFE pattern-oriented application framework, the *application context* is outlined in Section 5.2.2. The concurrency in the VaR estimation application can be described as an instance of the Monte Carlo Methods pattern (see Appendix A). Clarifying this association brings forth a set of background knowledge in the solution techniques and their trade-offs. This reduces the time needed to acquire the necessary information to make design choices. For this reason, the *application context* improves the productivity of the users of the PFE application framework.

The *software architecture* of the PFE application framework is described in Section 5.2.3. By clarifying the structure of a parallel implementation of a Monte Carlo based estimation engine, a PFE application expert can effectively take advantage of the concurrency opportunities in the PFE estimation problem, and learn about the suitability of the solution techniques to exploit the concurrency opportunities on the GPU. This improves the productivity of the PFE application framework users by providing assistance with the navigation of the software architecture of a PFE application framework.

While the implementation of this application framework is on-going research, we see that its implementation can help the application framework users productively understand the application concurrency opportunities and the application framework architecture. The challenge is in deploying such a large infrastructure in a real world scenario, with a significant change to an extension point interface. We focus on these challenges for PFE in Section 6.2.2.

5.3 Summary

This chapter demonstrates that a significant amount of optimizations can be incorporated into an application framework, many of which require an intimate understanding of the underlying computing platform capabilities, as well as the knowledge of application implementation alternatives. Such optimizations cannot be achieved by application domain experts using just a powerful compiler or parallel programming experts using just a sample sequential implementation. Both areas of expertise are required.

In the case of ASR implementation, the optimizations that have been explored include:

- Four parallel programming techniques for the acceleration of the irregular data access patterns in the inference process (Section 5.1.2)
- The selection of recognition network representation, where there is an intricate trade-off between the simpler, more regular, but highly redundant Linear Lexical Model (LLM) representation and the more advanced, highly irregular, and more terse Weight Finite State Transducer (WFST) representation (Section 5.1.3)
- The optimization of recognition network structure, where the network structure is optimized to assist with more efficient data-parallel graph traversal on manycore micro processors (Section 5.1.4)
- Implementation platform-specific optimizations, where we found that different graph traversal techniques are optimal on different platforms, and that platform-specific algorithm tuning and adaptation is required (Section 5.1.5)

In the case of VaR performance optimizations, attempts were made utilizing three perspectives (Section 5.2.4), including:

- A task-centric perspective, where the application was architected to leverage existing efficient library routines, and the algorithm for computing VaR was transformed and reformulated to reduce the total number of operations
- A numerical-centric perspective, where the random numbers generator and statistical distribution conversion algorithms were selected based on their numerical properties in order to achieve fast numerical error convergence
- A data-centric perspective, where the implementation is optimized based on its data access patterns

In the case of PFE performance optimization, the efforts focused on developing a software architecture where the capabilities of a manycore processor can be leveraged to deliver application performance (Section 5.2.6).

This chapter also highlighted the productivity concerns in developing extension points to allow the application framework to be flexibly reused in a variety of application usage scenarios. The challenge of deploying an application framework for the industry is the topic of the next chapter.

Chapter 6

The Deployment of a Pattern-Oriented Application Framework

Chapter 4 introduced an exemplar of a pattern-oriented application framework for a domain expert to effectively utilize manycore micro processors, and Chapter 5 offered a detailed explanation of the construction process of the pattern-oriented application framework, where the amount of application-level optimization that goes into an application framework is often beyond what an application domain expert can handle using a parallelizing compiler. This chapter illustrates how pattern-oriented frameworks can be customized for various usage scenarios and deployed in the field.

The ASR application framework involves approximately ten thousand lines of code and can be deployed as a stand-alone module. Section 6.1 demonstrates two deployments of the ASR application framework, with one of them extending the ASR framework to incorporate video information to enable lip-reading in noisy recognition environments.

While the application frameworks for the risk analytics applications examined in this thesis are still on-going research, based on our optimized reference implementations of the VaR and PFE applications, we have a grounded estimate of their expected scale of complexity. When completed, the VaR application framework involves approximately ten thousand lines of code, and can be deployed as a stand-alone module. The PFE application framework, when fully implemented, will involve a code base of more than one hundred thousand lines of code, with addition plug-ins implemented in external libraries. This poses significant challenges in the deployment process, which is explored in Section 6.2.

6.1 Automatic Speech Recognition Application Framework Deployment

In this section, two case studies are presented that illustrate the flexibility of an application framework in regard to adapting it to different usage scenarios. The case studies

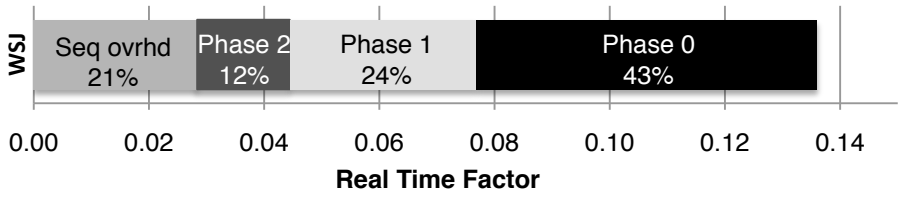


Figure 6.1: The recognition speed shown as real time factor (RTF) demonstrated with the Wall Street Journal corpus at 8.0% WER

show two examples in extending the ASR application framework to an advanced audio-only speech recognition application and an audio-visual recognition application that enables lip-reading in high noise recognition environments. The adaptation to the latter scenario also demonstrates how the ASR application framework has enabled a programmer with prior experience only in Matlab and Java to effectively utilize a GPU to produce an implementation that achieves a 20x speedup in recognition throughput as compared to a sequential CPU implementation.

6.1.1 Application Framework Reference Implementation

The reference implementation is provided with a small 5000-word model based on the Wall Street Journal corpus trained on broadcast news. The acoustic model was trained by HTK [154] with the speaker independent training data in the Wall Street Journal 1 corpus. The frontend uses 39 dimensional features that have 13 dimensional MFCC, delta and acceleration coefficients. The trained acoustic model consists of 3,000 16-mixture Gaussians. The WFST network is an $H \circ C \circ L \circ G$ model compiled and optimized offline with the dmake tool described in [11]. There are 3.9 million states and 11.4 million arcs in the WFST graph representing the language model. The test set consists of 330 sentences totaling 2,404 seconds from the Wall Street Journal test and evaluation set. The serial reference implementation using the LLM representation has a word error rate (WER[1]) of 8.0% and runs with a 0.64 real time factor.

For the implementation platform, we used the NVIDIA GTX480 (Fermi) GPU with a Intel Core i7 920 based host platform. GTX480 has 15 cores, each with dual issue 16-way SIMD arithmetic units running at 1.45GHz. Its processor architecture allows a theoretical maximum of two single-precision floating-point operations (SP FLOP) per cycle, resulting in a maximum of 1.39 TeraFLOP of peak performance per second. For compilation, we used Visual Studio 2008 with nvcc 3.1.

To achieve the same 8.0% WER, the framework's reference implementation achieved 0.136 real time factor, or 4.7x speed up. The pruning threshold was set at 20,000 states, and the resulting run traversed an average of 20,062 states and 141,071 arcs per time step. From Figure 6.1, we observe that the execution time is dominated by Phase 0 for data

[1]The Word Error Rate is computed by summing the substitution, insertion and deletion errors of a recognition result after it is matched against a golden reference using the longest common sub-sequence matching algorithm.

gathering. Phase 0 is necessary to align operands for Phase 2, the graph traversal phase. For the sequential overhead, 65% is used for transferring the backtrack log from the GPU to the CPU and 35% is for backtrack, file input and output on the CPU.

6.1.2 Deployment for Usage in Meeting Transcription

The SRI speech models for meetings [146] are produced for the accurate automatic transcription of multi-party meetings. It aims to construct an interactive agent that provides online and offline assistance to meeting participants.

We used the speech models from the SRI CALO realtime meeting recognition system [146]. It is produced with an advanced frontend that uses 13 dimensional perceptual linear prediction (PLP) features with first, second, and third order differences, is vocal-track-length-normalized and is projected to 39 dimensions using heteroscedastic linear discriminant analysis (HLDA). The acoustic model is trained on conversational telephone and meeting speech corpora, using the discriminative minimum-phone-error (MPE) criterion. The language model is trained on meeting transcripts, conversational telephone speech, and web and broadcast data [142]. The acoustic model includes 52K triphone states which are clustered into 2,613 mixtures of 128 Gaussian components. This type of acoustic model uses tied Gaussian Mixtures [142] and requires a more complex Phase 1 module for observation probability computation.

To adapt the ASR application framework to the SRI Meeting Model, a new Observation Probability Computation plug-in was developed to handle the tied Gaussian Mixture Model with a two-step computation. The first step computes the match between the input data and tied Gaussian mixtures related to active states. The second step applies mixture weights specific to each triphone state that is associated with active states. This extension also involved a new component to read in SRI's model file format. A new Result Output plug-in was developed to produce files for SRI's accuracy scoring script. The Pruning Strategy plug-in was kept the same.

The test set consisted of excerpts from NIST conference meetings, taken from the "individual head-mounted microphone" condition of the 2007 NIST Rich Transcription evaluation. The segmented audio files total 44 minutes in length and comprise 10 speakers. For the experiment, we assumed that the feature extraction is performed offline so that the inference engine can directly access the feature files. The meeting recognition task is very challenging due to the spontaneous nature of speech. The ambiguities in the sentences require larger number of active states to keep track of alternative interpretations, which leads to slower recognition speed.

Table 6.1 shows at various pruning thresholds the accuracy in WER, speed in real time factor and speedup on the GTX480 GPU (compared to an optimized sequential version implemented on the CPU). By leveraging the optimizations in the ASR framework, we were able to achieve up to 14.2x speedup GPU compared to a sequential version run on the CPU.

For data analytics, where one searches for the keywords in a recorded meeting conversation, the range of WER presented here is tolerable and still yield useful results.

Table 6.1: Accuracy, word error rate (WER), for various beam sizes and corresponding decoding speed in real-time factor (RTF)

# of Active States	30k	20k	10k	3.5k
WER	41.6%	41.8%	42.2%	44.5%
Sequential RTF	4.36	3.17	2.29	1.20
Manycore RTF	0.37	0.22	0.18	0.13
Speedup	11.6x	14.2x	13.0x	9.10x

6.1.3 Deployment in Audio-Visual Speech Recognition Usage Scenario

Robustness of speech recognition can be significantly improved by multi-stream recognition and especially by audio-visual speech recognition (AVSR). Under -10dB of babble noise, AVSR WER can be almost halved, from 10.3% WER to 5.4% WER [95]. The improvement in robustness is of interest for example for human-machine interaction in noisy reverberant environments, and for transcription of or search in multimedia data. The most robust implementations of audiovisual speech recognition often utilize coupled hidden Markov models (coupled HMMs) [118], which allow for both input streams to be asynchronous to a certain degree. In contrast to conventional speech recognition, the coupled HMM approach increases the search space significantly as the coupling of audio and video states requires the state machines to be composed to allow for asynchronies between sound and image in the pronunciation of a word in an input video sequence. This increase in the search space makes many current implementations of coupled HMM systems not real-time capable. Thus, for real-time constrained applications, such as online transcription of VoIP communication or responsive multi-modal human-machine interaction, using parallel computing capability is vital to achieve real-time performance.

Model Architecture of Coupled HMMs

Multistream and audiovisual speech recognition both use a number of streams of audio and/or video features in order to significantly increase robustness and performance ([127, 119]). Coupled HMM, with their tolerance for stream asynchronicities, can provide a flexible integration of these streams and have shown optimum performance in a direct comparison of alternative model structures in [118].

In coupled HMMs, both feature vector sequences are retained as separate streams. As generative models, coupled HMMs describe the probability of both feature streams jointly as a function of a set of two discrete, hidden state variables, which evolve analogously to the single state variable of a conventional HMM.

Thus, coupled HMMs have a two-dimensional state $\mathbf{q}$ which, for audiovisual recognition, is composed of an audio and a video state, q_a and q_v, respectively. Figure 6.2 shows this composition.

Each possible sequence of states through the model represents one possible alignment with the sequence of observation vectors. To evaluate the likelihood of such an alignment,

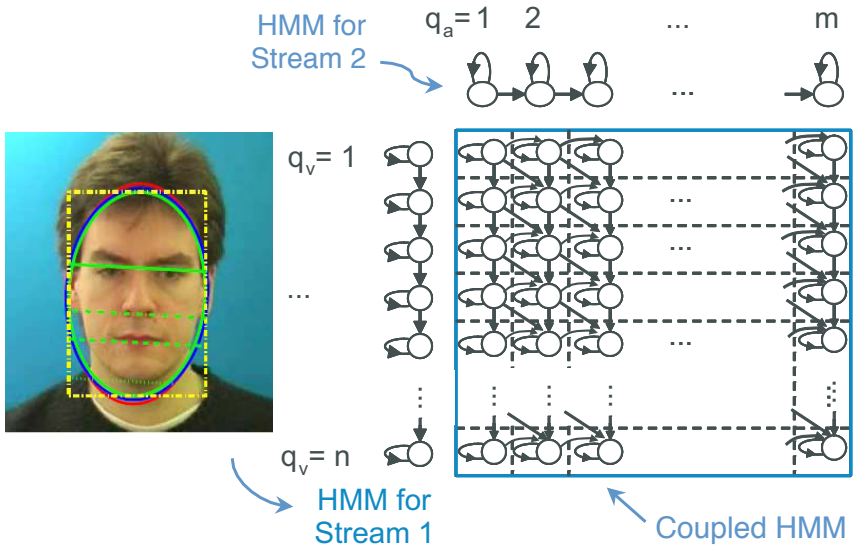


Figure 6.2: A coupled HMM consists of a matrix of interconnected states, which each correspond to the pairing of one audio- and one video-HMM-state, q_a and q_v, respectively.

each state pairing is connected by a transition probability, and each state is associated with an observation probability distribution.

The transition probability and the observation probability can both be composed from the two marginal HMMs. Then, the coupled transition probability becomes

$$\begin{aligned} \mathrm{p}\Big(q_a(t+1) \quad &= j_a, q_v(t+1) = j_v | q_a(t) = i_a, q_v(t) = i_v\Big) \\ &= a_a(i_a, j_a) \cdot a_v(i_v, j_v) \end{aligned} \tag{6.1}$$

where $a_a(i_a, j_a)$ and $a_v(i_v, j_v)$ correspond to the transition probabilities of the two marginal HMMs, the audio-only and the video-only single-stream HMMs, respectively. This step of the computation, the so-called *propagation step*, is memory intensive due to the need for transition probability lookup in a large and irregularly structured network.

For the observation probability, both marginal HMMs could be composed similarly, to form a joint output probability by

$$\mathrm{p}(\mathbf{o}|\mathbf{i}) = b_a(o_a|i_a) \cdot b_v(o_v|i_v), \tag{6.2}$$

where $b_a(o_a|i_a)$ and $b_v(o_v|i_v)$ denote the output probability distributions for both single streams.

Such a formulation, however, does not take into account the different reliabilities of the two feature streams. Therefore, Eq. (6.2) is commonly modified by an additional stream weight γ as follows

$$\mathrm{p}(\mathbf{o}|\mathbf{i}) = b_a(o_a|i_a)^{\gamma} \cdot b_v(o_v|i_v)^{1-\gamma}. \tag{6.3}$$

Finally, computation of the marginal HMM state probabilities can be implemented in the same way as for a standard single HMM system, e.g. as an M-component Gaussian mixture

model (GMM)

$$b(o|i) = \sum_{m=1}^{M} \gamma_m \mathcal{N}\left(o \mid \mu_{i,m}, \Sigma_{i,m}\right). \tag{6.4}$$

$\mathcal{N}\left(o|\mu, \Sigma\right)$ stands for a multivariate Gaussian distribution evaluated for the vector-valued observation o with mean μ and covariance matrix Σ. The covariance matrix may be either a full or a diagonal covariance matrix, where the latter implies that feature vector components are either independent or that their dependencies may be neglected.

The steps given by (6.3) and (6.4) will be referred to as the *likelihood combination* and the *likelihood computation* steps, respectively. These steps, especially the *computation*, have a much greater compute-to-memory-access ratio than the propagation step, due to the computational effort involved in GMM evaluation.

How was the framework used?

A coupled HMM can be compiled into a WFST that conforms to the required format in two steps:

- First, the two-dimensional state index needs to be converted into one linear index for each of the involved word models. These linearized word models can then be stored in an applicable format, in this case, the OpenFST [12] input format.
- Second, the word models need to be composed into the sentence level WFST. This compilation, and a subsequent minimization, were carried out using OpenFST, which resulted in an overall network size of 3167 states and 12751 arcs for the GRID grammar [49].

Once the WFST network is available, the only relevant change with respect to a "regular" HMM is the observation probability computation according to Eq. (6.3). Therefore, the significant extension point for enabling coupled HMM-based audiovisual and multistream ASR was the observation probability computation step, which had to be adapted for coupled HMMs. For this purpose, CUDA kernels were implemented for Equations (6.3), the *likelihood combination*, and (6.4), the *likelihood computation* step.

The likelihood computation was optimized especially for the use with full covariance matrices, which can often result in substantial performance improvements; the relevant optimizations are shown in some detail in [96]. For the likelihood combination step, a simple kernel was designed that is parallelized over all those coupled states that are in the active set at the given time frame.

Performance results

The WFST decoder was used for multi-stream speech recognition in the following experiment. The two combined marginal HMMs were a 39-dimensional full-covariance Mel-frequency Cepstrum model and a 31-dimensional diagonal covariance RASTA-PLP model. The accuracy remained precisely the same for the C++ reference implementation and the

GPU version of decoder, reaching 99.3% for the best-performing, speaker-independent model on clean data.

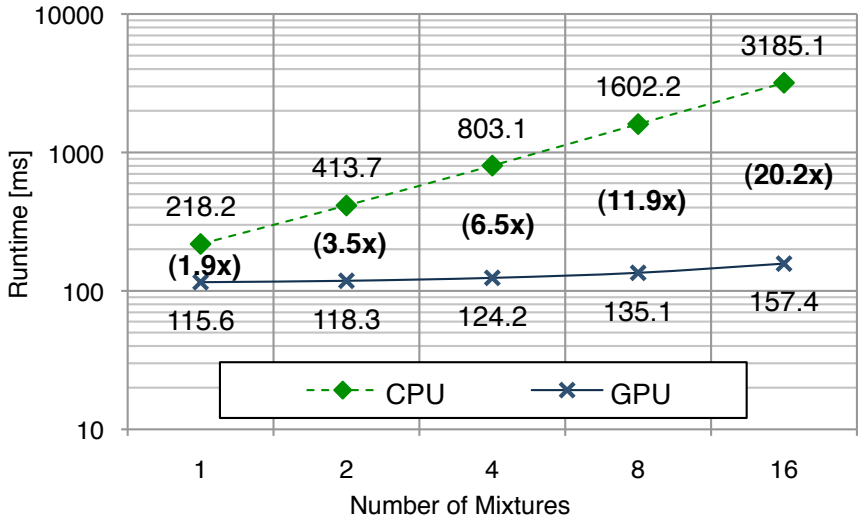


Figure 6.3: Runtime in ms per file of 3s length for $M = 1, 2, \ldots, 16$ mixture components used in Eq. (6.4). The speedup factor S is given in parentheses.

Figure 6.3 shows the runtime per file, averaged over 47 utterances, where the decoder was running sequentially on a Intel Core i7 920 CPU or on a NVIDIA GTX480 GPU.

As can be seen, the speedup grows almost linearly with model complexity, reaching 20.2x speed up with models using 16 mixture components. For the GPU version, system overhead for calling the accelerator dominates the overall runtime for models with less than 4 mixture components. The likelihood computation starts dominating the runtime for more complex models with 8 and 16 mixture components.

6.1.4 Deployments to Industrial Usage Scenarios

Commercial usage of ASR is already appearing in industries such as customer service call centers for data analytics where ASR is used to search recorded content, track service quality, and provide early detection of service issues. In such usage scenarios, speech transcription is one of the many processing steps in the exaction of actionable information from speech-based data sources.

For a code module of ten thousand lines of code, ASR can be conveniently deployed as a complete software module with flexible interfaces that allow for the use of customization for specific usage scenario without jeopardizing the performance the ASR module.

6.2 Risk Analytics Application Framework Deployment

In this section, we discuss the deployment considerations of two risk analytics applications: the Value-at-Risk (VaR) estimation application and the Potential Future Exposure

(PFE) estimation application. As previously mentioned, the construction of the risk analytics application frameworks is a topic of on-going research. We estimate the expected complexity of the code base using our optimized reference implementations as grounding, and discuss the anticipated challenges for the deployment of these application frameworks once they are available.

We estimate that the VaR application framework can be built with ten thousand lines of C++, and the PFE aplication framework will require one hundred thousand lines of C++. The deployment consideration is closely associated with the size of the framework being deployed. Using the widely used basic COCOMO model [30], a framework at the size of a VaR application framework takes an estimated 2.8 engineer-years to develop. A framework such as a PFE application framework could take ten times as much efforts to develop. While in VaR, the market risk that a portfolio is exposed to is estimated for the portfolio, in PFE, different business logic is required for estimating the risk exposures between counterparties for each contract. For example, the business logic for interest rate contracts, credit default swaps, foreign exchange contracts, commodity contracts, and equity contracts can vary significantly. The variety in the business logic requires a much large code base to support. As the anticipated size of the two frameworks differs by an order of magnitude, we expect them to require different deployment strategies. The next two sections explore the specific strategies that can be used.

6.2.1 Value-at-Risk Application Framework Deployment

The VaR application framework produces risk metrics for risk managers in banks. The application can be deployed in a variety of usage scenarios. In general, the inputs of a VaR application are financial data feeds and the outputs are the VaR estimation results entered into a database. The results can be used by analysts at quant desks for strategy validation or for risk assessment and stress testing.

To deploy the VaR application framework in these scenarios, an infrastructure must be available to calibrate the parameters used in the VaR estimation, where the parameters describe the sensitivity of a portfolio's value to various risk factors over time. The infrastructure for calibration and the infrastructure for risk estimation are linked by the parameter set. In practice, under the revised Basel II market risk framework, calibration is required at least once a month, where as the VaR metric can be computed on-demand or updated regularly. From this perspective, a VaR application framework can be deployed as a software module in a larger risk assessment infrastructure, with customizable functions in plug-ins without jeopardizing the execution efficiency of the VaR application framework. Potential plug-ins include modules for various finanical data feed formats, modules utilizing different risk models and confidence intervals, and modules for outputing specific file formats.

6.2.2 Potential Future Exposure Application Framework Deployment

PFE is a metric to quantify the counterparty risk of a financial portfolio posed by fluctuations in market prices in the future during the lifetime of the transactions, where the counterparties are brokers, investment banks, and other securities dealers that serve as the contracting party when completing financial securities transactions. Counterparty risk must be quantified on a transaction-by-transaction basis, where extensive business logic and modeling abstractions must be incorporated into the risk estimates for each type of transactions. A PFE engine requires a code base that can be ten times larger than that of a VaR engine. There is usually also a large set of legacy software routines in the form of a transaction-pricing library that is present as part of the infrastructure. Each legacy pricing routine typically estimates the value of one transaction under one particular market condition at one specific point in time.

In Section 5.2.6, we showed that to efficiently utilize the GPU, the interface between the Monte Carlo engine and the pricing library must be modified. The Monte Carlo engine sets up the market scenarios for the PFE estimation and the pricing library computes the Net Present Value (NPV) of each instrument under the market scenarios. The PFE *pattern-oriented application framework* for the GPU redefines the extension-point interface for the pricing library. The framework extension-point interface allows sets of transactions to be priced on sets of scenarios at the same time (Section 5.2.7). This approach uses a more suitable granularity of computations for highly parallel manycore computing platforms, and was demonstrated to achieve orders of magnitude performance improvements on the GPU. With the new extension point interface, the infrastructure of a PFE framework can efficiently execute the pricing workload over the tens of cores on one manycore microprocessor within a computing cluster node. The infrastructure can also continue to distribute the pricing workload to an entire cluster of computing nodes.

The redefinition of extension-points, however, requires the re-tooling of existing pricing libraries to allow batch pricing of instruments. Such efforts can take years of engineering effort. Investing in such a large engineering effort is a non-trivial decision and companies rarely have the resources to concurrently support the existing infrastructure and develop a new GPU-based infrastructure. The resource-intensive nature of industrial deployment of large new application frameworks is a significant challenge for the adoption of application frameworks, as the "*inversion of control*" property of application framework demands the application framework to completely take over the control of the existing execution infrastructure [65].

One solution to resolve this challenge is to enable the incremental deployment of the application framework, such that the GPU-accelerated pricing engines can be developed over time, and the benefits derived from the GPU-accelerated pricing engines can be leveraged before the infrastructure is fully replaced.

An incremental deployment of the application framework requires two capabilities to be in-place:

1. The Monte Carlo engine should be capable of initiating groups of transactions on

groups of scenarios.

2. The scenario generation routine should be able to replicate the generation of the same scenario for the simulation of different groups of transactions.

With these two properties, one can continue to utilize the legacy Monte Carlo engine in the PFE application to work on most types of transaction while allowing the GPU to accelerate the specific ones that can be accelerated. In a deployment, the legacy Monte Carlo engine can skip working on the GPU-accelerated transaction types, allowing the skipped transactions to be executed later by the GPU-accelerated pricing library. In this way, there can be a seamless deployment of the GPU-accelerated solution as the pricing library is slowly migrated toward being completely GPU-accelerated over time.

6.3 Summary

In this chapter, we demonstrated the deployment of the application framework in end-user usage scenarios. The automatic speech recognition (ASR) application framework was successfully deployed in the meeting transcription scenario with the acoustic models from SRI, and it achieved a maximum of a 14.2x speed up on the GPU when compared to an optimized sequential version implemented on the CPU. The ASR application framework was also used to deploy an audio-visual speech recognition task that enables lip-reading. By using the application framework, a Matlab/Java programmer was able to achieve a 20.2x speed up on the GPU as compared to a sequential version running on the CPU.

The industrial deployment of the applications enabled by the application frameworks has also been discussed. There are some known challenges in deploying frameworks that involve large software infrastructures of more than one hundred thousand lines of code, as legacy and accelerated software infrastructures may have to co-exist as the infrastructure is being migrated. We proposed an incremental deployment of the application framework, where the application can be partitioned such that the legacy system can skip some computation intentionally and have them completed by the manycore-accelerated implementation separately.

The deployment examples for the ASR application framework as well as the deployment discussions of the risk analysis application frameworks provide efficient and productive implementation paths from application to highly parallel software implementations on the manycore microprocessors. They demonstrate that pattern-oriented application frameworks can effectively close the implementation gap (as described in Section 2.4) for domain experts looking to utilize manycore microprocessors.

Chapter 7

An Ecosystem for Pattern-Oriented Application Frameworks

We believe that *pattern-oriented application frameworks* are tools that can be widely deployed in industry. Their adoption can close the parallel software application *implementation gap* for application domain experts (Section 2.4), and allow application domain experts to productively develop and deploy software applications for the new generations of highly parallel manycore microprocessors. This chapter analyzes the ecosystem in which *pattern-oriented application frameworks* can find initial adoption and evolve to meet industry needs.

The concept of an *ecosystem for technology and businesses* originated in the early 1990s, when James F. Moore described a new ecology of competition [113]. In his book [114], he elaborated on the strategic planning concept of a business *ecosystem*, and defined the concept as an economic community supported by a foundation of interacting organizations and individuals – the organisms of the business world. Moore writes:

> In the new world...the new paradigm is about market creation. It is about envisioning and helping to shape networks of contributions and processes in order to weave rich new economic tapestries.

The driving force behind the ecosystem for *pattern-oriented application framework* is the introduction of manycore microprocessors as a general-purpose computing platform. Today, the highly parallel manycore microprocessor has entered the center stage in the high performance computing (HPC) community. The NVIDIA Tesla M2050 manycore GPU helped put the Tianhe-1a supercomputing center at the number one position in the top-500 list of the World's fastest supercomputers in November 2010 [7]. The GPU is being recognized as an economical approach to obtain world-class computing capabilities for many compute-intensive workloads.

The capabilities of these highly parallel GPUs, as we have see in this thesis, are not restricted to supercomputing applications in supercomputing facilities. Companies can build internal compute facility with GPU-based servers from top-tier hardware vendors such as Dell and HP, or they can rent computing time on GPU-based servers from cloud computing

vendors such as Amazon EC2, PierOne, and Penguin Computing. For prototyping an application, a development system costs just a few hundred dollars. There is little hardware acquisition barrier for companies to explore the benefits of a GPU-based parallel computing platform.

The bottleneck limiting the increasing utilization of manycore microprocessor is the parallel software application *implementation gap*. To close the *implementation gap* with *pattern-oriented application frameworks*, we discuss four important questions regarding the adoption and evolution of the *pattern-oriented application frameworks*:

1. Who are the *lead users* of *pattern-oriented application frameworks*?
2. Who are developing *pattern-oriented application frameworks*?
3. What building blocks are used for developing *pattern-oriented application frameworks*?
4. What are the components of a thriving ecosystem for *pattern-oriented application frameworks*?

The answers to these questions sketch out the potential challenges of developing an ecosystem for *pattern-oriented application frameworks* and lays down directions for future research.

7.1 Lead Users of Pattern-Oriented Application Framework

Lead users, as defined by Hippel [82], are users who: 1) face needs that will be general in a marketplace, but face them months or years before the bulk of that marketplace encounters them, and 2) are positioned to benefit significantly by obtaining a solution to those needs.

While many compute-intensive workloads used in industry are going to experience significant acceleration on GPU platforms, one group of companies that can see immediate benefits to their operations are the software-as-a-service (SaaS) companies with their own private computing infrastructure who are running compute-intensive workloads for their business.

The SaaS sector generates $8 billion a year, and is growing at 22% a year[1]. There exists a tight connection between the efficiency of the services SaaS companies provide and the profit they realize, where an order of magnitude speed-up in their workload can result in significant business value. From our interactions with SaaS companies, we found that those providing compute-intensive hosted solutions represent an important group of *lead users*. These companies face the needs to increase services efficiency and are positioned to benefit from the reduced operations cost and improved profit margin by using more efficient software applications.

[1]http://www.crmlandmark.com/saasmarket.htm, quoting a May 7, 2009 Gartner report titled Market Trends: Software as a Service, Worldwide, 2009-2013

However, the deployment of efficient software applications on manycore microprocessors is plagued with the implementation gap of highly parallel software applications (Section 2.4). To realize the full performance potential of the highly parallel manycore microprocessors, a development team requires both application domain expertise, as well as parallel computing platform expertise. Many SaaS companies do not have the resources to obtain both areas of expertise to develop the application they need.

With the concept of *pattern-oriented application frameworks* for domain experts, an application framework can be developed as a short-term project, with the exploration of application algorithmic design space done by small teams of application domain experts and parallel programming experts. Once the application framework is set up, it can be maintained with just the application domain experts on staff.

While this lays out a group of lead users who can significantly benefit economically from *pattern-oriented application frameworks*, it naturally raises the question: Who are developing these industrial-strength *pattern-oriented application frameworks*?

7.2 Developers of the Pattern-Oriented Application Framework

We have shown in this thesis that the *pattern-oriented application framework* is an implementation assistant tool that allows domain experts to productively generate efficient parallel implementations for various application usage scenarios. We describe three groups of people who are motivated to developing *pattern-oriented application frameworks*: teams within companies, academic researchers, and consultants.

Teams within Companies Software engineering teams within companies can create efficient *pattern-oriented application frameworks* to allow domain experts in different teams to collaborate in a structured manner. For example, in the development of a PFE application, one team can be working on accelerating the infrastructure of the PFE Monte Carlo engine, while others can be developing plug-ins such as pricing library routines.

As both application domain expertise and parallel programming expertise are required to create an application framework, developing an application framework in-house for a company consists of a multi-year commitments of maintaining a team of multiple experts. Such effort can easily cost millions of dollars, and is only affordable by institutions such as government labs working on super computing centers [77, 75], or in highly lucrative industries such as hedge funds and trading firms[2]. There exist significant financial barriers for small business entities to leverage the benefits of manycore microprocessors.

Companies that can afford to develop application frameworks keep their implementations proprietary and consider them key competitive advantages of the company. The *extension points*, or the narrow interfaces that are used for extending the capabilities of the framework can be developed by the third-party. In other words, *pattern-oriented application framework*

[2]Although there are few sources to cite, infrastructure teams in hedge funds are known to build elaborate proprietary infrastructures similar to application frameworks for the quants and traders to use.

can find initial adoption within companies and evolve over time as more plug-ins either in-house or through third-party vendors.

Academic Researchers Academic researchers create efficient *pattern-oriented application frameworks*, such as our automatic speech recognition engine, as tools to assist researchers and practitioners implement applications in a field of study.

For the development of application frameworks in academia, multiple research groups are required to work closely together for an extended period of time. The development of the ASR application framework presented in this thesis is made possible by the collaboration between parallel programming researchers and speech recognition researchers in separate research groups over a period of more than two years. Depending on the structure of the academic institution, this may be a significant commitment.

Application frameworks can find initial adoption among researchers and practitioners in a field, and then rely on the open source community to evolve over time. Application frameworks that can thrive in this environment must be in a domain that appeals to a broad community of developers and users to survive and evolve over time.

Consultants Consultants create *pattern-oriented application frameworks* as tools to help efficiently implement a variety of end-user applications that their clients care about. A *pattern-oriented application framework* can be optimized and deployed for multiple clients with targeting different usage scenarios, lowering the cost for clients to develop highly parallel software applications in-house.

A consulting firm with an interdisciplinary team can create application frameworks, improve the frameworks over time, and deploy the framework for their clients. The development cost of the application frameworks can be amortized over multiple clients, making it economically feasible to invest in the extensive optimizations necessary in constructing efficient application frameworks.

Consulting firms fill an important gap in the economics of developing application frameworks. However, the process of sharing application framework development infrastructure also raises questions regarding the intellectual property rights associated with the application framework.

A *pattern-oriented application framework* is a piece of intellectual property (IP) that can provide business advantage of lower cost of application development and deployment. One can group *pattern-oriented application frameworks* into three categories[3]: *core IP*, *important IP*, and *other IP*.

Core IP is essential for a company's competitiveness in the market place and crucial for the viability of the company. Therefore, they are often closely guarded. An example is the trading infrastructure used in a high-frequency trading hedge fund, where a small one

[3]This categorization is used in the development of Nike's open innovation initiative "GreenXchange", where environmentally-friendly technologies are brought into a forum such that they can be licensed by third parties [123].

millisecond advantage in trading latency can mean 100 million dollars of revenue a year[4]. Application frameworks that fit into this category are developed exclusively in-house.

Important IP for a company is strategic for the company's success, but may be in adjacent industries that are not part of the company's core competency. An example is the automatic speech recognition technology for a call center data analytics company, where it is important for the company's business, but not part of the company's core business. Application frameworks that fit into this category can be developed in collaboration with consultants.

Other IP for a company is the technology necessary in the operation of the company. An example is the development of a billing system for an online retailer. Application frameworks that fit into this category are usually development by third-parties and purchased as a product or service.

A *pattern-oriented application framework* developed by a consultant can find initial adoption with *lead users* in companies requiring the framework as *important IP*. As the framework evolves, it can be adapted to usage scenarios for a variety of fields, and be packaged into a product or a service for other companies requiring it as *other IP*.

Given the groups of people in companies, academic institutions and consultancies who are motivated to build application frameworks to leverage the capabilities of manycore microprocessors, what are the building blocks they need to construct a *pattern-oriented application framework*?

7.3 Building Blocks for Developing Pattern-Oriented Application Frameworks

Our *pattern-oriented application framework* for parallel programming involves four main components: application context, software architecture, reference implementation, and extension points.

To describe the application context and present the software architecture, a system of parallel programming patterns is required as the common language. In this thesis, we use Our Pattern Language (OPL) [94] as a foundation.

The parallel programming patterns, as well as the pattern language [5], forms a source knowledge with which software architectural design choices can be contemplated and design alternatives with sever bottlenecks can be eliminated. An example of this process is demonstrated in Section 5.1.1, where concurrency opportunities were evaluated with the parallel programming patterns.

In constructing the reference implementation, a plethora of building blocks can be used. The capabilities of the manycore microprocessors are exposed through vendor-provided software development kits (SDK) and application development languages such as CUDA and

[4]Philip A Davis quoting a Goldman Sachs estimate in Wall Street's big bet on IT, Intel Circuit News, June 7, 2007.

[5]A pattern language is set of inter-related design patterns that are organized to provide guidance to a software developer for the process of solving a design problem.

OpenCL. Basic libraries are also provided by the microprocessor vendors. In the CUDA-based ecosystems, for example, standard libraries for many domains such as CUBLAS[6] [51], CUFFT[7] [51], and CUDPP[8] [52] have been developed by the hardware manufactures. For more specialized libraries, the ecosystem depend on third-party entities to produce libraries specific for particular domains. For example, EM Photonics [9] provides the CULA (GPU-accelerated linear algebra library), which contains specialized routines such as variations of eigensolvers; Cluster Technologies[10] provides functions to generate random numbers and interest rate paths in a financial application library.

To customize a *pattern-oriented application framework*, the application domain expert can implement plug-ins that target the extension points provided by the application framework. There are a number of productivity tools that help application domain experts implement the plug-ins. We mention three examples here.

Thrust [145] is a standard template library (STL) interface for CUDA. It provides a more productive layer C++ based abstraction for programmers. GMAC [70] is a CUDA runtime to unifying GPU and CPU memory space. It provides a shared memory abstraction to simplify the coordination of data between CPU and GPU memory spaces. Copperhead [39] is a data parallel subset of Python. It leverages the wealth of infrastructure that exists around Python and provides some automatic optimization capabilities by performing introspection on the module developed in Copperhead.

These tools aim to provide abstractions to hide implementation details from the application domain experts, and can serve well to provide additional productivity to the application framework customization process. However, they are not designed to provide guidance in exploring application level design space, and cannot replace the combined expertise of application domain experts and parallel programming experts in developing application frameworks.

7.4 Components of a Thriving Ecosystem

Creating a thriving ecosystem for using *pattern-oriented application frameworks* to close the parallel programming *implementation gap* will be impossible without a community of participants. When constructing one application framework can involve years of effort from an interdisciplinary team of experts, creating a thriving ecosystem cannot depend on any one business, academic, or government entity alone. We see four components that if allowed to evolve over time, will help promote a thriving ecosystem for constructing and using *pattern-oriented application frameworks*.

[6]CUBLAS: CUDA dense linear algebra routines

[7]CUFFT: CUDA Fast Fourier Transforms

[8]CUDPP: CUDA Performance Primitives – common data-parallel programming routines such as parallel scan, parallel sort

[9]EM Photonics is a privately held organization headquartered in Newark, Delaware that uses hardware based platforms, such as FPGAs and GPUs, to accelerate applications for industries, such as scientific computing and image processing.

[10]Cluster Technologies Ltd is a Hong Kong based professional service company that specializes in the provisioning of advanced computing technology solutions (http://www.clustertech.com/)

A Community for Parallel Programming Design Pattern

Recall that a *design pattern* is a generalizable solution to a class of recurring problem. There has been a wealth of activity in the design pattern community on documenting softwaredesign patterns online. One of the early efforts was the Portland Pattern Repository, a repository for computer programming design patterns. The web site was accompanied by a companion website, WikiWikiWeb, which was the first wiki (a collection of reader-modifiable Web pages), to help make exchange of ideas between programmers easier[11].

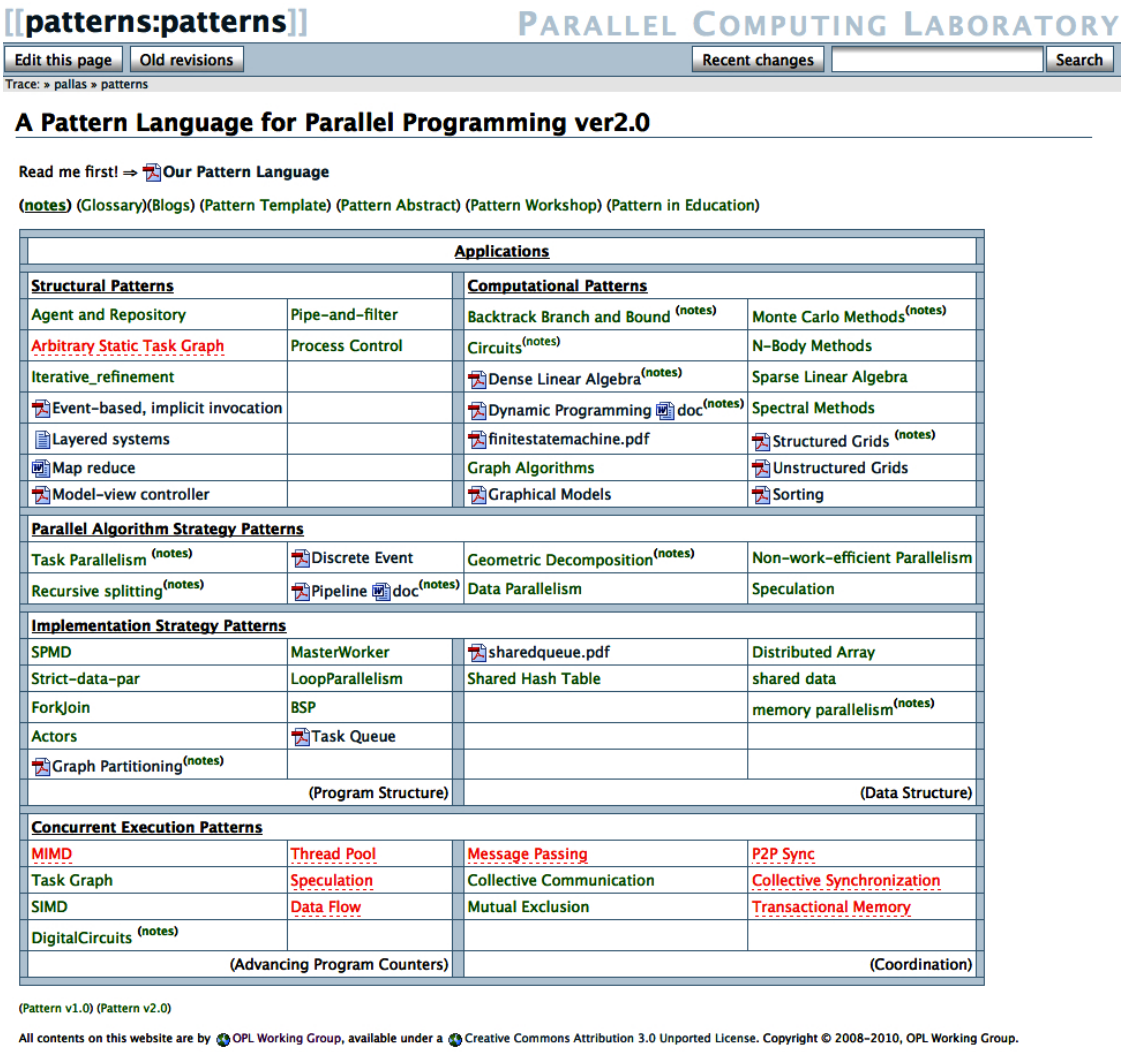
Wiki is an ideal way to document patterns. As a good pattern write-up requires multiple areas of expertise, and new solutions appear as the underlying implementation technology evolves over time. A pattern documented on a wiki can be easily updated to include new solutions.

Each pattern is an extensive document that can have an elaborate structure. For example, each pattern in OPL [94] has a *name* that alludes to the problem or the solution of the pattern that can invoke familiarity for the reader, a narrative that illustrates the *problem* that it solves, the *context* in which it is relevant, the *forces* that act on the solution, and the *solution* to the problem. It also includes pedagogical *examples*, *known uses*, as well as a list of *related patterns*. As more application frameworks are developed, new *solutions* acting on the solution can come to light. For example, through our work in financial risk analytics accelerating Value-at-Risk estimates (Section 5.2.4), we were able to introduce a new numerical-centric perspective in the *solution* section discussing to how the numerical properties of random number generator can be used to accelerate convergence of the result in the Monte Carlo Methods Computational Pattern (Appendix A). A wiki-based pattern repository can allow such insights to be mined and shared as soon as they are validated in practice.

The wiki for a pattern language for parallel programming should be more structured than the Portland Pattern Repository, where the site has sprawled to more than 34,000 pages as of 2010. We have developed a website (see Figure 7.1) with a top-level structure of five areas of patterns, where researchers from both industry and academia are collaborating on refining pattern for parallel programming. The researchers come from a variety of academic institutions, including University of California, Berkeley, University of Illinois, Urbana-Champaign, University of Victoria Canada, as well as from industry, including Intel, Microsoft, National Instrument, and NEC. We have made significant progress in defining the patterns for the benefit of the application development community and the parallel programming community.

In terms of the ecosystem for *pattern-oriented application framework*, a set of up-to-date patterns and the pattern language definitions is crucial to provide a consistent set of vocabulary and background for the discussion of software architectures. The OPL wiki [2] aims to be a community-supported, up-to-date living website of documents to fulfill this purpose.

[11] WikiWikiWeb was created by Ward Cunningham in 1994 and released in 1995



[[patterns:patterns]] PARALLEL COMPUTING LABORATORY

Edit this page | Old revisions | Recent changes | Search

Trace: » pallas » patterns

A Pattern Language for Parallel Programming ver2.0

Read me first! ⇒ Our Pattern Language

(notes) (Glossary)(Blogs) (Pattern Template) (Pattern Abstract) (Pattern Workshop) (Pattern in Education)

Applications			
Structural Patterns		**Computational Patterns**	
Agent and Repository	Pipe-and-filter	Backtrack Branch and Bound (notes)	Monte Carlo Methods (notes)
Arbitrary Static Task Graph	Process Control	Circuits (notes)	N-Body Methods
Iterative_refinement		Dense Linear Algebra (notes)	Sparse Linear Algebra
Event-based, implicit invocation		Dynamic Programming doc (notes)	Spectral Methods
Layered systems		finitestatemachine.pdf	Structured Grids (notes)
Map reduce		Graph Algorithms	Unstructured Grids
Model-view controller		Graphical Models	Sorting
Parallel Algorithm Strategy Patterns			
Task Parallelism (notes)	Discrete Event	Geometric Decomposition (notes)	Non-work-efficient Parallelism
Recursive splitting (notes)	Pipeline doc (notes)	Data Parallelism	Speculation
Implementation Strategy Patterns			
SPMD	MasterWorker	sharedqueue.pdf	Distributed Array
Strict-data-par	LoopParallelism	Shared Hash Table	shared data
ForkJoin	BSP		memory parallelism (notes)
Actors	Task Queue		
Graph Partitioning (notes)			
(Program Structure)		(Data Structure)	
Concurrent Execution Patterns			
MIMD	Thread Pool	Message Passing	P2P Sync
Task Graph	Speculation	Collective Communication	Collective Synchronization
SIMD	Data Flow	Mutual Exclusion	Transactional Memory
DigitalCircuits (notes)			
(Advancing Program Counters)		(Coordination)	

(Pattern v1.0) (Pattern v2.0)

Figure 7.1: A screen shot of the Our Pattern Language (OPL) website

A Forum for Software Architecture Discussions

As parallel computing platforms become ubiquitous, we would like to expose and exploit parallelism in an expanding variety of application to take advantage of the capabilities of these platforms. As shown in Section 5.1.1, an application often contains a hierarchy of concurrency opportunities. An analysis and discussion of these opportunities can bring forth bottleneck in potential implementations before significant effort is invested.

We envision an online forum, where discussions about software architecture decisions can take place. For application framework developers, the process of concisely describing the hierarchy of concurrency opportunities using the pattern language to others in a forum can be a helpful exercise. With feedback from others about the software architecture of applications, cross-pollination of ideas can take place, where a hierarchical composition of patterns in one application domain may influence the design decisions in another domain.

A software architecture forum provides a space in the eco-system for *pattern-oriented application framework* to evolve over time, allowing efficient composition of patterns to be shared across application domains.

A Symbiotic Relationship between Libraries and Frameworks

As described in Section 7.3, there are a plethora of on-going efforts in developing accelerated modules for application framework developers to use. A good resource for CUDA based development is the CUDA Community Showcase website [1]. As of November 2010, there are 1228 accelerated applications/libraries posted, with the work searchable by types and domains.

Understanding the landscape of existing accelerated module is crucial for building *pattern-oriented application frameworks*. By using standard modules, such as '*sort*, *scan* or *matrix multiply*, one is defining an extension point in the application framework where the plug-in is a standard model that is accelerated and maintained by third-parties. As the underlying parallel platform architecture evolves over time, the application framework can stay efficient by choosing the most efficient implementation of the standard module.

In terms of the ecosystem for *pattern-oriented application framework*, the application framework and standard modules have a symbiotic relationship. The more application frameworks use a standard module as a building block, the more important a standard module becomes, and more resources will be spent on optimizing its performance. The better the performance of a standard module, the more likely application frameworks will choose to use it. To allow the ecosystem to thrive, *pattern-oriented application framework* developers should choose to use existing standard building blocks where possible.

A Scalable Plug-in Development and Management System

One of the goals of the *pattern-oriented application framework* is to provide a software environment where an efficient reference implementation can be productively customized through the extension points with plug-in modules without jeopardizing execution efficiency. Depending on the interface of the extension point, the plug-ins could be in a standard off-the-shelf module, such as *sort*; a domain-specific module, such as file-parsing module for a domain-specific file format; or an application-specific module, such as a heuristic for a particular algorithm step. Although the extension points are specified with particular interface definitions in-mind, as the application framework and the plug-ins evolve over time, compatibility issues inevitably arise. As a *pattern-oriented application framework* evolves, the management of a large set of plug-ins for each *pattern-oriented application framework* becomes an important challenge.

An enabling technology to manage this problem for the *pattern-oriented application framework* ecosystem is *continuous integration*. Continuous integration is a software development practice where members of a software development team frequently integrate their changes to a central repository. When changes are checked in, automatic build and test routines are triggered to exercise the code with a test suite, and the results are automatically examined for errors. In an environment where an update in the application framework must be tested with a portfolio of plug-ins, continuous integration is crucial for the stability of

the code base. One of the tools that assists with the practice of continuous integration is Hudson [93]. The build and release management is implemented with a web interface, and the build and test processes can be distributed to remote servers, allowing a large number of plug-ins to be tested concurrently.

With tools like Hudson, we envision that flexible *pattern-oriented application frameworks* can be developed to accommodate a variety of plug-ins that will allow application domain experts to productively create efficiently end-user applications for highly parallel computing platforms.

7.5 Summary

In this chapter we recognize that in order to develop and deploy *pattern-oriented application frameworks* in industry, an ecosystem must exist to support the application framework developers and their users. We answer four important questions on the adoption and evolution of *pattern-oriented application framework* regarding: the *lead users*, the developers, the building blocks, and the ecosystem component of the *pattern-oriented application framework*.

We suggest that potential *lead users* of *pattern-oriented application frameworks* for parallel computing are SaaS companies who can obtain immediate economic benefits in increased compute-efficiency of applications on manycore microprocessors (Section 7.1).

The *developers* of *pattern-oriented application frameworks* can be teams within companies, academic researchers, or consultants (Section 7.2). Building a team within a company to develop *pattern-oriented application framework* can be an expensive undertaking: a team of parallel programming and application domain experts working for a year could mean millions of dollars in investment. The significant resource requirement puts parallel application development out-of-reach of many small companies. Building an application framework in academia requires a multi-year collaboration between multiple research groups. For the application framework to evolve, it must be in a domain that appeals to a broad community of developers and users. Building an application framework in a consulting company can be more practical, as there can be a stable group of experts working together to create and deploy solutions that can be improved upon over time. The high-cost of development can be amortized by deploying the framework for multiple usage scenarios for multiple clients.

The *building blocks* of *pattern-oriented application frameworks* include parallel programming patterns, libraries of accelerated modules, and programming tools (Section 7.3). Parallel programming patterns provide the vocabulary and background for developers and users in the *application context* and *software architecture* components of the *pattern-oriented application frameworks*. The libraries of accelerated modules assist in building the *reference implementation*. And the programming tools assist users to productively customize the *pattern-oriented application frameworks* and build plug-ins to target the *extension points*.

The *components of a thriving ecosystem* for *pattern-oriented application frameworks* include: 1) a community for parallel programming design patterns, 2) a forum for software architecture discussions, 3) a symbiotic relationship between libraries and frameworks, and 4) a scalable plug-in development and management system (Section 7.4). The community for parallel programming allows the patterns and pattern language definitions to evolve

over time and stay up-to-date with new solution techniques mined from application framework implementations. The forum for software architecture discussion provides feedback to application framework developers in the process of exposing and exploiting application concurrency. The symbiotic relationship between libraries and frameworks allows frameworks to be more portable when they utilize standard libraries modules, and allows libraries to gain value and importance as more frameworks depend on them. The scalable plug-in development and management system based on *continuous integration* allows the complexity of frameworks, extension points, and plug-ins to be practically managed, enabling the *pattern-oriented application framework* to evolve over time.

By clarifying the lead users, developers, building blocks, and the components of the ecosystem for *pattern-oriented application frameworks*, we observe that many ecosystem components are already in place. At the same time, we recognize that some ecosystem components that are still missing. Specifically, there are lead users and developers ready to adopt the *pattern-oriented application framework* concepts, and many building blocks such as patterns, libraries, and programming tools are already available to assist application framework developers. On the other hand, continued effort is required to evolve the parallel programming patterns and the pattern language, discussion forums for software architecture design still need to be set up, and a sample implementation of a *pattern-oriented application framework* with plug-in management capabilities still needs to be demonstrated.

As the components of the ecosystem fall into place, with the wide adoption and deployment of *pattern-oriented application framework*, we believe the *implementation gap* of parallel application development can be effectively closed, and application domain experts will be able to productively develop efficient applications on manycore microprocessors.

Chapter 8

Key Lessons

In this chapter, we highlight the key lessons learned in the process of developing and utilizing pattern-oriented application frameworks. We first provide our observations as a context behind each lesson and then provide the lessons learned.

8.1 Industry Landscape

We start with the examination of the broad industry landscape in both hardware and software solutions from the perspective of the application domain expert.

With respect to the hardware landscape, the observation is that the "Power Wall", "Memory Wall" and "ILP Wall" have forced microprocessor architectures to go parallel. A new breed of manycore microprocessors has emerged to use many simpler and more power-efficient processor cores in parallel in order to achieve high compute throughput within practical power budgets. This has caused a disruption in the application development process, where significantly more parallelism must be exposed and exploited in the development of performance-sensitive end-user applications on manycore microprocessors.

The lesson learned is that data layout, data placement, and synchronization processes are increasingly important factors to consider when extracting performance from highly parallel manycore microprocessors. Failure to carefully consider these factors could mean an order of magnitude of loss in application performance. Yet these factors are not usually part of an application domain expert's daily concerns. Application domain experts need an effective tool to productively utilize highly parallel manycore microprocessors to accelerate their applications.

With respect to the software landscape, the observation is that the expanding variety of applications and the applications' increasing complexity motivate the need for application domain expertise in the development of parallel applications that can take advantage of parallel manycore microprocessors.

The lesson learned is that both application domain expertise and parallel programming expertise are required as having either one or the other area of expertise alone is not sufficient if one wants to develop efficient parallel software applications. This problematic situation is the parallel application *Implementation Gap*.

8.2 Implementation Gap

To address the *Implementation Gap*, we examined available tools that can help application domain experts effectively utilize manycore microprocessors. We assert that a tool to effectively assist application domain experts to more productively implement efficient applications on highly parallel microprocessors must provide application execution *efficiency*, application development *productivity*, and application domain expert effort *portability*.

Software architecture narrative tools

The observation is that *software architecture narrative* tools help application domain experts grasp the background knowledge necessary to understand the opportunities and challenges in developing a parallel application. These tools include idioms, patterns and pattern languages.

The lesson learned is that while assistance in understanding the background contributes to higher productivity in software development for an application domain expert, using it alone is not enough to close the implementation gap. An application domain expert still needs significant expertise in the parallel hardware platform to implement the applications.

Software implementation support tools

The observation is that *software implementation support* tools assist application domain experts with the implementation of the applications. These tools include libraries, skeletons, and application frameworks. While libraries have concise APIs and are great abstractions for complex tasks, they often implement fixed functions and are relatively inflexible to adapting to new application requirements. Skeletons provide the outline of an algorithm in a programming environment, and users can incorporate custom functions within the skeleton as "flesh" code. Skeletons are more flexible than libraries, and target applications with a single dominant algorithm well. They provide little help, however, with the composition of multiple levels of algorithms in complex applications. The *application framework*, which is a type of software framework, involves complex compositions of algorithms to implement an application. However, there are often concerns about how easy it is to learn and use an application framework.

The lesson learned is that a software environment can be built to impose an additional constraint on the concept of application framework such that customizations of the framework may only be in harmony with the underlying software architecture. The constrained customization capability allows application domain experts to flexibly leverage the efficient underlying infrastructure involving a composition of algorithms while implementing new functions for particular usage scenarios by constructing custom plug-ins for specific framework extension points. In addition, by emphasizing the use of parallel programming patterns to illustrate our pattern-oriented application frameworks, we provide a rich resource to concisely describe complex software architectures, making it convenient to learn and use our *pattern-oriented application frameworks*.

Software implementation support tools

The observation is that *software implementation support* tools must be constructed in some software implementation infrastructures. The common infrastructures, such as processes, threads, MPI, OpenMP, CUDA and OpenCL, have been discussed. Concepts from processes and threads form the basis of parallel processing paradigms. MPI and OpenMP build on top of processes and threads with SPMD, loop level, as well as task level parallelism abstractions to assist developer in utilizing cluster and multicore platforms.

The lesson learned is that in order to target the level of parallelism within a manycore microprocessor, CUDA and OpenCL provide the necessary abstraction to effectively exploit fine-grained parallelism.

8.3 Pattern-Oriented Application Framework

We proposed pattern-oriented application frameworks for parallel programming as an approach to help application domain experts to more effectively utilize highly parallel manycore microprocessors and productively implement efficient applications in a portable way.

The first observation is that application domain experts seek assistance to better understand the concurrency inherent in an application to assess the application's acceleration potential.

The lesson learned is that by incorporating an elaboration of the characteristics of an application and exposing the application's parallelization opportunities in the *application context* component of a *pattern-oriented application framework*, we can can provide the application domain experts with a better understanding of the application concurrency to assess the application's acceleration potential.

The second observation is that application domain experts seek assistance to learn about techniques for exploiting the application concurrency, as well as assistance to understand the trade-offs between alternative implementation techniques and the common bottlenecks to avoid.

The lesson learned is that by illustrating the organization of a software application as a hierarchical composition of structural and computational programming patterns, the *Software Architecture* component in a *pattern-oriented application framework* can provide the application domain experts with a better understanding of the trade-offs involved in exploiting application concurrency based on parallel programming patterns. The component can also provide application domain experts with guidance to using the reference implementation component.

The third observation is that application domain experts seek reference designs to demonstrate how the techniques for exploiting application concurrency can be integrated together in order to achieve efficient execution.

The lesson learned is that by providing a fully functional, efficiently constructed sample parallel design of the application, the *Reference Implementation* component of the *pattern-oriented application framework* can allow application domain experts to experience the performance achievable on a hardware platform and examine the specific application concur-

rency that is exploited in an efficient software implementation.

The fourth observation is that application domain experts seek assistance in flexible customization of the reference design to implement a full class of applications so as to target different usage scenarios.

The lesson learned is that by creating interfaces for creating families of functions that extend the capability of the application framework, the *extension point* component of the *pattern-oriented application framework* provides an environment with which an application domain expert can quickly develop new capabilities for an application targeting different usage scenarios.

The four components of the *pattern-oriented application framework* work together to allow application domain experts more productively develop efficient applications on many-core microprocessors.

8.4 The Construction of Pattern-Oriented Application Frameworks

The observation is that many engineering-years of optimizations can be incorporated into a pattern-oriented application framework. Many of these optimizations require an intimate understanding of the underlying computing platform capabilities, as well as knowledge of the application implementation alternatives.

Specifically, for the development of an efficient automatic speech recognition (ASR) reference implementation, application domain expertise was applied in sections 5.1.3 and 5.1.4 to explore multiple recognition network representations and to transform input recognition network structure. Parallel programming expertise was applied in sections 5.1.1, 5.1.2, and 5.1.5 to expose and exploit application concurrency and to experiment with multiple HW implementation platforms.

For the development of our Value-at-Risk (VaR) reference implementation, application domain expertise was applied to reformulate the algorithm and select algorithms with desirable numerical properties to achieve in faster simulation convergence. Parallel programming expertise was applied to organize the software architecture around efficient library components, eliminate redundant data transfers between small kernels, and introduce data-blocking strategies to take advantage of the processor memory hierarchy.

For the development of our Potential Future Exposure (PFE) reference implementation, application domain expertise was applied to the definition of alternative application programming interfaces (API) and develop pricing library components to simulate the behavior of specific types of financial instrument; Parallel programming expertise was applied to refactor the Monte Carlo simulation engine for execution speed on manycore multiprocessors and enable the engine's incremental deployment into the company's infrastructure.

The lesson learned is that building a team that has multiple areas of expertise the development of every end-user application can be cost-prohibitive for many applications domains. At the same time, failure to enable the deployment of efficient end-user applications on highly parallel microprocessors will severely stunt the growth of the entire semiconduc-

tor industry. Pattern-oriented application frameworks can amortize the cost of developing an efficient infrastructure across multiple end-user usage scenarios. It is an approach that can effectively bridge the *Implementation Gap* to productively construct efficient end-user applications on highly parallel microprocessors.

8.5 Deployment of Pattern-Oriented Application Framework

We demonstrated the deployment of the application framework in end-user usage scenarios. The automatic speech recognition (ASR) application framework was successfully deployed in the meeting transcription scenario with the acoustic models from SRI, as well as in an audio-visual speech recognition task that enables lip-reading. We also discussed the challenges of deploying large-scale application frameworks of over 100,000 lines of code with accesses to third party libraries, and proposed solutions for incremental deployment.

The observation is that in the deployment of the ASR application framework in an meeting transcription scenario, we achieved a maximum of a 14.2x speed up on the GPU when compared to an optimized sequential version implemented on the CPU. In the deployment of the ASR application framework in an audio-visual speech recognition application that enables lip-reading, a programmer with only prior experience in Matlab/Java was able to leverage the framework and achieve a 20.2x speed up on the GPU as compared to a sequential version running on the CPU.

The lesson learned is that by utilizing the large number of performance optimizations in a *pattern-oriented application framework*, application domain experts can effectively customize the frameworks to produce efficient end-user applications on highly parallel manycore microprocessors.

The observation is that depending on the scope of the deployment of an application framework, it may require years of engineering efforts to setup the associated libraries and performance sensitive components or framework plug-ins to fully replace legacy systems.

The lesson learned is that an incremental deployment strategy can be used where the application can be partitioned such that the legacy system can skip some computation intentionally and have the skipped portions completed by the manycore-accelerated implementation separately.

8.6 The Ecosystem for Pattern-Oriented Application Framework

Finally, we discussed the ecosystem of development and deployment of application frameworks and how application frameworks can evolve and leverage each other over time.

The observation is that Moore's concepts of ecosystem for technology and businesses motivates an analysis of the lead users, developers, building blocks, and ecosystem components required to enable wide adoption and deployment of *pattern-oriented application*

frameworks.

The lesson learned is that by clarifying the lead users, developers, building blocks, and the ecosystem of the pattern-oriented application framework, we learned that many ecosystem components, such as a community for *parallel programming design patterns*, are already in place. We also recognize that other ecosystem components, such as a forum for software architecture discussions, are still missing. As the components of the ecosystem fall into place, with the wide adoption and deployment of pattern-oriented application framework, we believe the *implementation gap* of parallel application development can be effectively closed, and application domain experts will be able to productively develop efficient applications on manycore microprocessors.

8.7 Summary

The landscape in microprocessor design is shifting to embrace parallelism. This shift is causing a disruption in the software development process. While the highly parallel manycore microprocessors can provide an order of magnitude more compute capability per chip, without careful considerations in data layout, data placement, and synchronization processes during the application development process, orders of magnitude of application performance could be lost. At the same time, the types of applications that take advantage of the manycore microprocessors are expanding. With the growing complexity in these applications, a parallel software application *implementation gap* appears between the application and the implementation platform. Both application domain expertise and parallel programming expertise are required to close this *implementation gap*, and enable the development of efficient parallel applications. The requirement of having both areas of expertise inflates the cost of adopting manycore microprocessors and slows the momentum of the semiconductor industry.

To meet these challenges, we developed the *pattern-oriented application framework* as a tool to allow application domain experts to productively utilize highly parallel manycore microprocessors to accelerate their applications. A *pattern-oriented application framework* encapsulates a large number of performance optimizations and involves four components: 1) the *application context* component outlines application characteristics using parallel programming patterns, 2) the *software architecture* component describes the application concurrency exploited in the framework as a hierarchical composition of patterns, 3) the *reference implementation* component contains a fully functional, efficiently constructed sample design allowing the acceleration to be experienced first hand, and 4) *extension point* component involves a set of interfaces and sample plug-ins for application domain experts to flexible customize the application framework to target specific usage scenarios.

We demonstrated efficient reference implementations of applications in the fields of machine learning and computational finance, specifically studying automatic speech recognition (ASR), financial market value-at-risk estimation, and financial potential future exposure estimation. We presented a *pattern-oriented application framework* for ASR, and illustrated four optimization efforts that went into the acceleration of the application including: 1) detailed efficiency optimizations resolving the challenges of implementing data-parallel speech

recognition, 2) optimization of the type of algorithm used in speech recognition in the context of manycore execution platform, 3) optimization of the input data format for more efficient data-parallel operations, and 4) optimization the software architecture with respect to hardware implementation platform parameters. Using the *pattern-oriented application framework* we have developed for ASR, we conducted two separate deployment case studies, one of which flexibly extends the ASR framework to incorporate video information to enable lip-reading in noisy recognition environments. The ASR application framework enabled a programmer with only prior experience in Matlab/Java to productively customize the framework with only a few hundred lines of code in selected plug-ins to produce an efficient end-user application. Running on a manycore GPU, the application achieved a 20x speedup in recognition throughput as compared to a sequential CPU-based implementation.

We believe that the *pattern-oriented application framework* is a tool that has the potential to be widely deployed in industry. Its adoption can close the parallel software application *implementation gap* for application domain experts to effectively utilize the highly parallel manycore microprocessors.

Bibliography

[1] CUDA community showcase. http://www.nvidia.com/object/cuda_showcase_html.html.

[2] Our Pattern Language website. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

[3] Portland Pattern Repository: WikiWikiWeb. http://c2.com/cgi-bin/wiki?PortlandPatternRepository.

[4] NVIDIA CUDA programming guide, April 2009. Version 2.2.

[5] Revisions to the Basel II Market Risk Framework. http://www.bis.org/publ/bcbs148.html, January 2009. Basel Committee on Banking Supervision, Bank for International Settlements.

[6] Risk management systems in the aftermath of the financial crisis: Flaws, fixes and future plans. http://www.sybase.com/files/White_Papers/RiskReport_wp.pdf, 2010. A GARP report prepared in association with SYBASE.

[7] Top500 Supercomputer List, November 2010. http://www.top500.org, November 2010.

[8] Semiannual OTC derivatives statistics at end-June 2010. http://www.bis.org/statistics/derstats.htm, June, 2010. Bank for International Settlements.

[9] SEC adopts new rule preventing unfiltered market access. http://sec.gov/news/press/2010/2010-210.htm, November 3, 2010. U.S. Securities and Exchange Commission.

[10] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[11] C. Allauzen, M. Mohri, M. Riley, and B. Roark. A generalized construction of integrated speech recognition transducers. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 761–764, 2004.

[12] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings CIAA 2007*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 2007.

[13] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. GpuCV: A GPU-Accelerated Framework for image processing and computer vision. *Advances in Visual Computing, Lecture Notes in Computer Science*, 5359:430–439, 2008.

[14] M. Anderson, B. Catanzaro, J. Chong, E. Gonina, K. Keutzer, C. Y. Lai, M. Moskewicz, M. Murphy, B. Y. Su, and N. Sundaram. PALLAS: Mapping Applications onto Manycore. *Multiprocessor System-On-Chip: Current Trends and the Future*, November 2010.

[15] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[16] I.A. Antonov and V.M. Saleev. An economic method of computing LP-sequences. In *U.S.S.R Comput. Maths. Math. Phys.*, page 252256, 1979.

[17] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[18] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[19] M. Bachle and P. Kirchberg. Ruby on Rails. *IEEE Software*, 24:105–108, 2007.

[20] S. A. Baeurle. Multiscale modeling of polymer materials using field-theoretic methodologies: a survey about recent developments. pages 363–426.

[21] R. W. Bailey. Polar generation of random variates with the t-distribution. *Math. Comput.*, 62:779–781, April 1994.

[22] J. K. Baker. The DRAGON system – An overview. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-23(1):24–29, February 1975.

[23] D. S. Batory. Construction of file management systems from software components. Technical report, CS-TR-88-36, University of Texas at Austin, Austin, TX, USA, 1988.

[24] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of Finite State Markov Chains. *Ann. Math. Statist.*, 37(6):1554–1563, 1966.

[25] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. In *Workshop on Specification and Design, held in connection with Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Orlando, Florida, October 4-8, 1987. http://c2.com/doc/oopsla87.html.

[26] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[27] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical timing analysis: From basic principles to state of the art. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(4):589 –607, april 2008.

[28] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. In *Journal of Political Economy*, pages 637–654, 1973.

[29] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[30] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II.* Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[31] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. pages 243–252, Los Alamitos, CA, USA, 6-9 Aug 1996. IEEE Computer Society.

[32] H. Bourlard, Y. Kamp, H. Ney, and C. J. Wellekens. Speaker-dependent connected speech recognition via dynamic programming and statistical methods. In M. R. Schroeder, editor, *Speech and Speaker Recognition*, volume 12 of *Bibliotheca Phonetica*, pages 115–148. Karger, Basel, 1985.

[33] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29:610–611, 1958.

[34] P. Bratley and B. L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14:88–100, March 1988.

[35] J. Brouillat, C. Bouville, B. Loos, C. D. Hansen, and K. Bouatouch. A Bayesian Monte Carlo approach to global illumination. *Comput. Graph. Forum*, 28(8):2315–2329, 2009.

[36] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[37] R. H. Campbell and N. Islam. A technique for documenting the framework of an object-oriented system. *Computing Systems*, 6(4):363–389, 1993.

[38] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau. GPU accelerated acoustic likelihood computations. In *Proceedings of the 9th Annual Conference of the International Speech Communication Association (Interspeech)*, pages 964–967, Brisbane, Australia, September 22-26, 2008.

[39] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. *16th ACM SIGPLAN Annual Symposium on Principles and Practices of Parallel Programming*, February 2011.

[40] B. Catanzaro, N. Sundaram, and K. Keutzer. A MapReduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, April, 2008.

[41] J. Chong, G. Friedland, A. Janin, N. Morgan, and C. Oei. Opportunities and challenges of parallelizing speech recognition. In *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, 2010.

[42] J. Chong, E. Gonina, D. Kolossa, S. Zeiler, and K. Keutzer. A speech recognition application framework for highly parallel implementations on the GPU. In *Submitted to Principles and Practices of parallel Programming*, 2011.

[43] J. Chong, E. Gonina, Y. Yi, and K. Keutzer. A fully data parallel WFST-based large vocabulary continuous speech recog nition on a graphics processing unit. In *Proceedings of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*, page 11831186, Brighton, UK, September 2009.

[44] J. Chong, E. Gonina, K. You, and K. Keutzer. Exploring recognition network representations for efficient speech inference on highly parallel platforms. *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 1489–1492, 26-30 September 2010.

[45] J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer. Data-parallel large vocabulary continuous speech recognition on graphics processors. In *Proceedings of the 1st Annual Workshop on Emerging Applications and Many Core Architecture*, pages 23–35, Beijing, China, June 21, 2008.

[46] J. Chong, K. You, Y. Yi, E. Gonina, C. Hughes, W. Sung, and K. Keutzer. Scalable HMM-based inference engine in large vocabulary continuous speech recognition. In *IEEE International Conference on Multimedia & Expo (ICME)*, pages 1797–1800, July 2009.

[47] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. PhD thesis, University of Glasgow,, United Kingdom, 1989. `http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz`.

[48] M. I. Cole. A skeletal approach to the exploitation of parallelism. In *Proceedings of the conference on CONPAR 88*, pages 667–675, New York, NY, USA, 1989. Cambridge University Press.

[49] M. Cooke, J. Barker, S. Cunningham, and X. Shao. An audio-visual corpus for speech perception and automatic speech recognition. *J. Acoust. Soc. Am.*, 120(5):2421–2424, 2006.

[50] J. Coplien. *Advanced C programming styles and idioms.* Addison-Wesley Pub. Co., 1992.

[51] CUDA CUBLAS Library version 3.1. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf`, May, 2010.

[52] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`.

[53] CUDA CUFFT Library. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUFFT_Library_3.1.pdf`, May, 2010.

[54] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, 1 1998.

[55] M. Danelutto. Dynamic run time support for skeletons. Technical report, Pisa, Italy, 1999.

[56] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. pages 319–334, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[57] J. Darlington. Fortran-S: A uniform functional interface to parallel imperative languages. In *Proc. of 3rd Parallel Computing Workshop (PCW'94)*, pages P1–C–1 to P1–C–5. Fujitsu Laboratories Ltd., November, 1994. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.452`.

[58] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 146–160, London, UK, 1993. Springer-Verlag.

[59] L.P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. *Software Reusability, Volume II, Applications and Experiences, Biggerstaff and Perlis, editors. Addison-Wesley*, pages 57–71, 1989.

[60] G. Dionne, P. Duchesne, and M. Pacurar. Intraday Value at Risk (IVaR) using tick-by-tick data with application to the Toronto Stock Exchange. *Journal of Empirical Finance*, 16(5):777 – 792, 2009.

[61] M. Dixon, J. Chong, and K. Keutzer. Acceleration of market Value-at-Risk estimation. pages 5:1–5:8, 2009.

[62] P. R. Dixon, T. Oonishi, and S. Furui. Fast acoustic computations using graphics processors. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 4321–4324, Taipei, Taiwan, 19-24 April 2009.

[63] P. R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Comput. Speech Lang.*, 23(4):510–526, 2009.

[64] P. Dubey. A Platform 2015 Model: Recognition, Mining and Synthesis moves computers to the era of tera. *Platform 2015 White Paper*, February 2005.

[65] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley Computer Publishing, New York, NY, 1999.

[66] M. Fayad, D. Schmidt, and Ralph Johnson. *Implementing Application Frameworks: Object-Oriented Framework at Work*. Wiley Computer Publishing, New York, NY, 1999.

[67] Message Passing Forum. MPI: A Message-Passing Interface standard. Technical report, University of Tennessee, FUT-CS-94-230, Knoxville, TN, USA, April 21, 1994. `http://www.netlib.org/tennessee/ut-cs-94-230.ps`.

[68] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[69] P. N. Garner, J. Dines, T. Hain, A. El Hannani, M. Karafiat, D. Korchagin, M. Lincoln, V. Wan, and L. Zhang. Realtime ASR from Meetings. *Proceeding of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 2119–2122, September, 2009.

[70] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGARCH Comput. Archit. News*, 38:347–358, March 2010.

[71] M.B. Giles, F.Y. Kuo, I.H. Sloan, and B.J. Waterhouse. Quasi-Monte Carlo for finance applications. In *ANZIAM Journal*, volume 50, pages 308–323, 2008.

[72] P. Glasserman. *Monte Carlo Methods in Financial Engineering (Stochastic Modelling and Applied Probability)*. Springer, August 2003.

[73] A. Godberg. *Smalltalk-80, The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[74] B. Gold and N. Morgan. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[75] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and applications. *High Performance Computing for Computational Science - VECPAR 2002, Lecture Notes in Computer Science*, 2565:15–36, 2003.

[76] L. Grimmer. Interview with David Heinemeier Hansson from Ruby on Rails, February, 2006. `http://dev.mysql.com/tech-resources/interviews/david-heinemeier-hansson-rails.html`.

[77] E. Grinspun, P. Krysl, and P. Schröder. CHARMS: a simple framework for adaptive simulation. *ACM Trans. Graph.*, 21(3):281–290, 2002.

[78] Khronos Group. The Khronos Group releases OpenCL 1.0 specification, December 8, 2008. `http://www.khronos.org/news/press/releases/the\_khronos\_group\_releases\_opencl\_1.0\_specification/`.

[79] K. Gupta and J. D. Owens. Three-layer optimizations for fast GMM computations on GPU-like parallel processors. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition & Understanding*, pages 146–151, December 2009.

[80] A. Hall. *A Methodology for System Engineering*. D. Van Nostrand Company, London, UK, 1962.

[81] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[82] E. Von Hippel. Lead Users: A source of novel product concepts. *Management Science*, 32(7):pp. 791–805, 1986.

[83] L. Howes and D. Thomas. Efficient random number generation and application using CUDA. In *GPU Gems 3, Chapter 37*, pages 805–830, 2007.

[84] Intel. Intel math kernel library. `http://software.intel.com/en-us/intel-mkl/`.

[85] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura. Parallel LVCSR Algorithm for Cellphone-Oriented Multicore Processors. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, pages I–177 – I–180, 2006.

[86] A. Janin. *Speech recognition on vector architectures*. PhD thesis, University of California, Berkeley, Berkeley, CA, 2004.

[87] F. Jelinek, L. Bahl, and R. Mercer. Design of a linguistic statistical decoder for the recognition of continuous speech. *Information Theory, IEEE Transactions on*, 21(3):250 – 256, May 1975.

[88] S. Joe and F. Y. Kuo. Remark on algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 29:49–57, March 2003.

[89] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[90] E. Jondeau, S. H. Poon, and M. Rockinger. *Financial modeling under non-Gaussian distributions*. Springer Finance. Springer, 2007.

[91] P. Jorion. *Value at Risk: The New Benchmark for Managing Financial Risk*. McGraw-Hill, 2000.

[92] S. Kanthak, H. Ney, M. Riley, and M. Mohri. A comparison of two LVR search optimization techniques. In *Proc. Intl. Conf. on Spoken Language Processing (ICSLP)*, pages 1309–1312, Denver, Colorado, USA, 2002.

[93] K. Kawaguchi. Continuous Integration in the Cloud with Hudson, 2009. JavaOne Conference 2009, `http://wiki.hudson-ci.org/download/attachments/37323793/Hudson+J1+2009.ppt`.

[94] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal, Addressing the Challenges of Tera-scale Computing*, 13(4):6–19, 2009.

[95] D. Kolossa, R. F. Astudillo, S. Zeiler, A. Vorwerk, D. Lerch, J. Chong, and R. Orglmeister. Missing feature audiovisual speech recognition under real-time constraints. In *Proc. 9th ITG conference on Speech Communication*, page paper 22, Bochum, Germany, October 6-8, 2010. VDE-Verlag Berlin.

[96] D. Kolossa, J. Chong, S. Zeiler, and K. Keutzer. Efficient manycore CHMM speech recognition for audiovisual and multistream data. *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 2698–2701, 26-30 September 2010.

[97] V. A. Korthikanti and G. Agha. Energy-performance trade-off analysis of parallel algorithms. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[98] C. Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, May 2002.

[99] H. Kuchen. A skeleton library. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 620–629, London, UK, 2002. Springer-Verlag.

[100] S. Kumar, C. J. Hughes, and A. D. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. Intl. Symposium on Computer Architecture (ISCA)*, pages 162–173. ACM, 2007.

[101] P. L'Ecuyer. Random numbers for simulation. In *Communications of the ACM*, pages 85–98, 1990.

[102] A. Lee, T. Kawahara, and K. Shikano. Julius - an open source real-time large vocabulary recognition engine. In *EUROSPEECH 2001*, pages 1691–1694, Aalborg, Denmark, September 3-7, 2001.

[103] K. F. Lee. *Large Vocabulary Speaker-Independent Continuous Speech Recognition: The Sphinx System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988.

[104] A. Leff and J. T. Rayfield. Web-application development using the Model/View/Controller design pattern. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:0118, 2001.

[105] An Interview with the Creator of Ruby, November 29, 2001. `http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`.

[106] B. T. Lowerre. *The Harpy Speech Recognition System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1976.

[107] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

[108] H. T. Macgillivray and R. J. Dodd. Monte-Carlo simulations of galaxy systems. pages 331–337, 2004.

[109] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. In *ACM Transactions on Modeling and Computer Simulation*, page 330, 1998.

[110] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison Wesley, New York, NY, 2004.

[111] P. R. McJones and G. F. Swart. Evolving the UNIX system interface to support multithreaded programs. Technical report, HP Labs Technical Reports, SRC-RR-21, Palo Alto, California, USA, September 28, 1987.

[112] M. Mohri, F. Pereira, and M. Riley. Weighted finite state transducers in speech recognition. *Computer Speech and Lang.*, 16:69–88, 2002.

[113] J. F. Moore. Predators and Prey: A new ecology of competition. *Harvard Business Review*, 71(5-6):75–86, 1993.

[114] J. F Moore. *The Death of Competition: Leadership & Strategy in the Age of Business Ecosystems*. HarperBusiness, 1996.

[115] B. Moro. The Full Monte. *Risk Magazine*, 8:57–58, February 1995.

[116] H. Murveit, M. Cohen, P. Price, G. Baldwin, M. Weintraub, and J. Bernstein. SRI's DECIPHER system. In *Proceedings of the workshop on Speech and Natural Language*, HLT '89, pages 238–242, Morristown, NJ, USA, 1989. Association for Computational Linguistics.

[117] K. V. Nagarajan, J. Fulcher, G. Awyzio, and P. J. Vial. Application of modelling and simulation to understand customer satisfaction patterns for telecommunication service providers, 2003. 4 pages, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.933`.

[118] A.V. Nefian, L. Liang, X. Pi, X. Liu, and K. Murphy. Dynamic bayesian networks for audio-visual speech recognition. *EURASIP Journal on Applied Signal Processing*, 11:1274–1288, 2002.

[119] C. Neti, G. Potamianos, J. Luettin, I. Matthews, H. Glotin, D. Vergyri, J. Sison, A. Mashari, and J. Zhou. Audio-visual speech recognition. Technical report, Johns Hopkins University, CLSP, 2000.

[120] H. Ney, R. Haeb-Umbach, B.-H. Tran, and M Oerder. Improvements in beam search for 10000-word continuous-speech recognition. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 9–12, 1992.

[121] H. Ney and S. Ortmanns. Dynamic programming search for continuous speech recognition. *IEEE Signal Processing Magazine*, 16:64–83, 1999.

[122] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[123] Inc Nike. Corporate Responsibility Report FY07-09. Technical report, Nike, Inc, FY07-09. `http://www.nikebiz.com/crreport/pdf/`.

[124] S. Ortmanns, H. Ney, and A. Eiden. Language-model look-ahead for large vocabulary speech recognition. In *Proc. Int. Conf. on Spoken Language Processing*, pages 2095–2098, 1996.

[125] UC Berkeley Parallel Computing Laboratory. `http://parlab.eecs.berkeley.edu/`.

[126] Douglas B. Paul. The Lincoln Continuous Speech Recognition system: recent developments and results. In *Proceedings of the workshop on Speech and Natural Language*, HLT '89, pages 160–166, Morristown, NJ, USA, 1989. Association for Computational Linguistics.

[127] E. Petajan, B. Bischoff, and D. Bodoff. An improved automatic lipreading system to enhance speech recognition. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, 1988.

[128] S. Phillips and A. Rogers. Parallel speech recognition. *Intl. Journal of Parallel Programming*, 27(4):257–288, 1999.

[129] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[130] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition, 1993. Computer Science and Automation, Indian Institute of Science, Bangalore, India. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.7596&rep=rep1&type=pdf`.

[131] Intel News Release. Intel unveils new product plans for high-performance computing: Intel many integrated core chips to extend intel's role in accelerating science and discovery. `http://www.intel.com/pressroom/archive/releases/20100531comp.htm`, May 31, 2010.

[132] D. C. Schmidt. Applying design patterns and frameworks to develop object-oriented communication software. In *Handbook of Programming Languages, Vol 1*, New York, April, 1999. MacMillan Computer Publishing. Edited by Peter Salus.

[133] R. Schwartz, Y. Chow, O. Kimball, S. Roucos, M. Krasner, and J. Makhoul. Context-dependent modeling for acoustic-phonetic recognition of continuous speech. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '85.*, volume 10, pages 1205 – 1208, April 1985.

[134] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[135] M. Shaw and D. Garland. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[136] J. Siek, L. Lee, and A. Lumsdaine. *The Boost graph library: User guide and reference manual*. Addison-Wesley Professional, 2001.

[137] C. C. W. Smithson. A LEGO Approach to Financial Engineering: An introduction to forwards, futures, swaps and options. *Midland Corporate Financial Journal.*, pages 16–28, Winter, 1987.

[138] M. Smotherman. History of multithreading. `http://www.cs.clemson.edu/~mark/multithreading.html`.

[139] I.M. Sobol'. Uniformly distributed sequences with an additional uniform property. In *U.S.S.R Comput. Maths. Math. Phys.*, page 236242, 1976.

[140] A. Srinivasan. Parallel and distributed computing issues in pricing financial derivatives through quasi Monte Carlo. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 14 –19, 2002.

[141] S. Srinivasan and J. Vergo. Object-oriented reuse: experience in developing a framework for speech recognition applications. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 322–330, Washington, DC, USA, 1998. IEEE Computer Society.

[142] A. Stolcke, X. Anguera, K. Boakye, O. Cetin, A. Janin, M. Magimai-Doss, C. Wooters, and J. Zheng. The SRI-ICSI Spring 2007 meeting and lecture recognition system. *Lecture Notes in Computer Science*, 4625(2):450–463, 2008.

[143] J. A. Stratton, S. S. Stone, and W. Hwu. M-CUDA: An efficient implementation of CUDA kernels on multi-cores. IMPACT Technical Report IMPACT-08-01, UIUC, February 2008.

[144] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. (Intel CPU Introductions graph updated in August 2009, available at `http://www.gotw.ca/publications/concurrency-ddj.htm`).

[145] Thrust: Code at the speed of light. `http://thrust.googlecode.com/files/An\Introduction\To\Thrust.pdf`, Apr 28, 2010.

[146] G. Tur, A. Stolcke, L. Voss, J. Dowding, B. Favre, R. Fernandez, M. Frampton, M. Frandsen, C. Frederickson, M. Graciarena, D. Hakkani-Tr, D. Kintzing, K. Leveque, S. Mason, J. Niekrasz, S. Peters, M. Purver, K. Riedhammer, E. Shriberg, J. Tien, D. Vergyri, and F. Yang. The CALO meeting speech recognition and understanding system. In *Proc. IEEE Spoken Language Technology Workshop*, pages 69–72, 2008.

[147] U. Vahalia. *UNIX Internals: The New Frontier*. Prentice Hall, 1996.

[148] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[149] R. Vuduc, J. W Demmel, and K. A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[150] Gregory V. W. Programmers tool chest: The OpenCV library. *Dr. Dobbs Journal*, November 1, 2000.

[151] X. Wang and I. H. Sloan. Low discrepancy sequences in high dimensions: How well are their projections distributed. volume 213, pages 366 – 386, 2008.

[152] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[153] Business Wire. Impact 360 speech analytics software from Verint Witness Actionable Solutions enhances performance and customer experiences at telefonica o2 ireland. *Financial Tech Spotlight*, July 19, 2010.

[154] P.C. Woodland, J.J. Odell, V. Valtchev, and S.J. Young. Large vocabulary continuous speech recognition using htk. In *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, volume ii, pages II/125 –II/128 vol.2, April 1994.

[155] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[156] Q. Xu, W. Wang, and S. Bao. A new computational way to Monte Carlo global illumination. *Int. J. Image Graphics*, 6(1):23–34, 2006.

[157] K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y.K. Chen, W. Sung, and K. Keutzer. Parallel scalability in speech recognition: Inference engine in large vocabulary continuous speech recognition. In *IEEE Signal Processing Magazine*, number 6, pages 124–135, November 2009.

[158] K. You, Y. Lee, and W. Sung. OpenMP-based parallel implementation of a continuous speech recognizer on a multi-core system. In *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '09, pages 621–624, Washington, DC, USA, 2009. IEEE Computer Society.

[159] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland. *The HTK Book Version 3.3*. Cambridge University Engineering Department, 2005.

[160] P. Youngman. Procyclicality and Value-at-Risk. *Bank of Canada Financial System Review Report*, pages 51–54, June, 2009.

Appendix A

Sample Pattern: Monte Carlo Methods

Monte Carlo methods are an important set of algorithms in computer science. They involve estimating results by statistically sampling a parameter space with a thousands to millions of experiments. The algorithm requires a small set of parameters as input, with which it generates a large amount of computation, and outputs a concise set of aggregated results. The large amount of computation has many independent component with obvious boundaries for parallelization. While the algorithm is well-suited for executing on a highly parallel computing platform, there still exist many challenges such as: selecting a suitable random number generator with the appropriate statistical and computational properties, selecting a suitable distribution conversion method that preserves the statistical properties of the random sequences, leveraging the right abstraction for the computation in the experiments, and designing the an efficient data structures for a particular data working set. This appendix presents the Monte Carlo Methods software programming pattern and focuses on the numerical, task, and data perspectives to guide software developers in constructing efficient implementations of applications based on Monte Carlo methods.

The Monte Carlo Methods pattern is a computational pattern that resolves the recurring problem of constructing efficient analysis involving estimating results by statistically sampling a parameter space with a large number of experiments. It is presented following the standard format of the OPL patterns [94], with a *name* that alludes to the problem or the solution of the pattern that can invoke familiarity for the reader (Section A.1), a *problem* definition (Section A.2), a *context* explaining the background in which the solution should be used (Section A.3), a set of *forces* defining the design space (Section A.4), the *solution* structure and considerations (Section A.5), *invariants* that should be true in order to apply this pattern (Section A.6). It also include pedagogical *examples* (Section A.7), *known uses* (Section A.8), as well as a list of *related patterns* (Section A.9).

A.1 Name

Monte Carlo Methods

A.2 Problem

The results of some analysis can be best estimated by statistically sampling a parameter space with thousands to millions of experiments using different parameter settings. How do we efficiently construct and execute this large number of experiments, and estimate the result?

A.3 Context

How do you estimate the market risk your investment portfolio is exposed to? How do you determine if a particular molecular dynamics system will reach equilibrium? Often, it is easy to solve a problem by setting up experiments to sample the parameter space and analyze the distribution of the results, but hard to solve the problem analytically.

For example, in a volatile financial market, it is crucial to keep track of the downside risks for an investment portfolio. Given a set of market trends and volatility measures, we can estimate the market Value-at-Risk (VaR) by simulating the effects of market trends on the portfolio value through a set of experiments. Specifically, the inputs are sets of market scenarios generated with certain statistical distribution, each experiment simulates the value of the portfolio given one market scenario, and the result is an analysis of the set of portfolio values produced given the distribution of market scenarios.

Market VaR measures the probability that a portfolio would suffer a loss greater than some threshold. For example, a VaR could be a 15% loss or more on the portfolio value with a probability of 1% over a day. Given changing market conditions, a portfolio's exposure to market risk also changes. A portfolio manager must regularly modify the distribution of the investments to adapt to changes in market condition to meet the investors' risk tolerance.

By statistically sampling a problem's parameter space, we can gain valuable insights into a complex problem that would be impossible or impractical to solve analytically. The quality of a solution estimated through sampling has several properties:

1. **The experiments are independent and parallelizable**: the approach assumes that experiments are independent and identically distributed (i.i.d.), such that the set of experiments provides a statistically valid sampling of the parameter space

2. **The process is computationally expensive**: by the central limit theorem, the statistical significance of solution, or more specifically, the statistical error (standard error) in the solution is proportional to the inverse square-root of the experimental size, i.e. to achieve 10x more precision in the result, one needs to run 100x more experiments

3. **The approach is flexible**: Experiments can closely mimic business logic. It does not require significant simplifying assumptions as compared to many analytical solutions. This allows a more realistic model to be used, which reduces the risk of violating key assumptions in using analytical solutions.

We call the process of sampling of the parameter space, simulating experiments, and aggregating the results, the Monte Carlo Method[1]. The ease and intuitiveness of setting up the experiments makes the Monte Carlo Method a popular approach [72]. However, because of the large number of experiments required to achieve statistically significant solutions, it is extremely computationally intensive. Efficient setup and execution of the experiments can make this method practical to a wide range of applications.

A.4 Forces

A.4.1 Universal forces

1. *Complexity of parameter modeling*
   We want to use as simple a statistical model as possible to model the parameters. Simplicity leads to ease of understanding and programming, and allows for efficient computation. At the same time, the parameter model needs to be detailed and complex enough to properly represent the statistical variations in the input parameters.

2. *Fast vs rigorous random number generation*
   We want to quickly generate many scenarios for experiments. At the same time, the scenarios generated must be independent and identically distributed based on *suitable* sequences of random numbers [101] to achieve the intended coverage of the parameter space, requiring a more rigorous random number generation process than the typical random number generator in a standard math library.

A.4.2 Implementation forces

1. *Easy experiment mapping vs load balancing*
   Given the complete independence of the experiments, the easiest way to exploit this concurrency is to map each experiment to a unit of execution (UE). However, the experiments vary in size, thus we need to consider task size to achieve a load balanced execution. This requires for more complex experiment-to-UE mapping.

2. *Experiment data working set size*
   We can allow a single experiment to use each UE to be sure to have data working set fit in cache, but execution is exposed to long memory latency. We can allow multiple experiments to share a processor to hide long latency memory fetches, but the working set may not fit in limited cache space.

3. *Static vs. dynamic sample generation*
   The sampling process uses random number sequences. The sequence can be generated before running any experiments, or it can be generated as part of the experiment. Generating it before running any experiment can save computation time if the same

[1]Due to Stan Ulam, John Von Neuman, and Enrico Fermi, who were Physicists who first used the approach for neutron diffusion calculations in the field of high energy physics.[15]

sequence will be used multiple times, but it could take significant memory resources to store. Generating it as part of the experiment could be a more memory efficient approach, but care must be taken as to how different threads share their random number generation state.

A.5 Solution

The Monte Carlo method uses a simple solution structure where experiments are generated, executed, and the experimental outputs are aggregated to estimate the result. We describe the structure of the solution and illustrate the perspectives and considerations that go into the implementation of the solution. The solution is outlined here:

1. Solution Structure (A.5.1)
2. Solution Considerations (A.5.2)
   (a) **Numerical-centric perspective** (A.5.2)
      i. Random Number Generation (A.5.2.1)
      ii. Parameter Distribution Conversion (A.5.2.2)
      iii. Multi-parameter Correlation (A.5.2.3)
   (b) **Task-centric perspective** (A.5.2)
   (c) **Data-centric perspective** (A.5.2)
      i. Experiment Generation - RNG step (A.5.2.1)
      ii. Experiment Generation - Distribution conversion step (A.5.2.2)
      iii. Experiment Execution Step (A.5.2.3)
      iv. Result Aggregation (A.5.2.4)

A.5.1 Solution Structure

Solving a problem with the Monte Carlo method involves a sampling of the parameter space with thousands to millions of experiments. The number of independent experiments provides significant opportunities for parallelization, which high performance computing (HPC) experts often call "embarrassingly parallel". While this describes the ample parallelism opportunities, there are still many implementation decisions that need to be made to achieve efficient execution. Each experiment requires a three-step process:

1. **Experiment generation**: generating a random set of input parameters for an experiment
2. **Experiment execution**: executing the experiment with its input parameter
3. **Result aggregation**: aggregating the experimental results to into statistical relevant solutions.

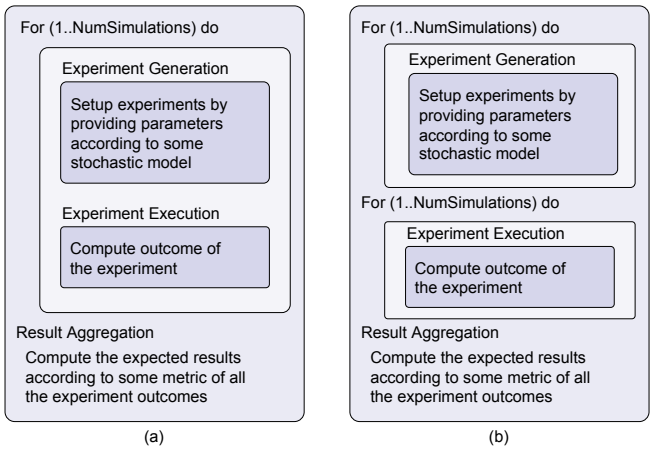


Figure A.1: Monte Carlo Methods solution structures

As there are millions of experiments to be computed, there are many valid ordering of these computations. Figure A.1 illustrates two types of ordering that enables static or dynamic generation of the random sequences: (a) illustrate the case with dynamic generation of random sequence, the values are generated per experiment and immediately consumed; (b) illustrate the case with a priori generation of random sequences, which could be generated once and stored for multiple uses.

At the top level, a Monte Carlo simulation may also involve not only one problem to be solved, but multiple very similar problems to be solved. For example, in Pi estimation, we have the one value of Pi to be estimated, where as in option pricing, we often have a batch of thousands of options potentially to be priced in parallel. When there is more than one problem to solve, we have the multiple levels of parallelization opportunities to explore.

A.5.2 Solution Considerations

The goal of using the Monte Carlo method is to quickly converge on a solution that is within a certain error bound. This motivates us to look at the **numerical-centric perspective** of the Monte Carlo method, which focuses on improving the convergence rate. The key is to select a parallelizable and scalable parameter generation technique that works well with the parallelization requirements of the rest of the implementation.

Experiment execution has the most amount of concurrency available, but also has the most variation in problem size and computation requirements. This calls for the **task-centric perspective**, where the key is to be able to quickly re-factor the computation with patterns to take advantage of existing libraries and frameworks.

The implementation of the Monte Carlo method is very data intensive. In the **data-centric perspective**, we focus on optimizing memory access patterns between different stages of an implementation.

Numerical-centric perspective

There are many existing techniques for generating parameters for Monte Carlo experiments. The parameters are usually generated in two to three steps. The two required steps are: *Uniform Random Number Generation* and *Parameter Distribution Conversion*. The third step is *Multi-parameter Correlation*, which may be performed separately, or combined with Experiment Execution.

Specifically, *Uniform Random Number Generation* produces an independent and identically distributed sequence of values uniformly over a sampling domain, e.g. a real value from 0.0 to 1.0. *Parameter Distribution Conversion* converts the sampled parameter to a distribution that matches the distribution of the real parameter, e.g. a normal distribution, or a Poisson distribution. *Multi-parameter Correlation* correlates the multiple parameters of an experiment to produce scenarios that are coherent with the assumptions of the experiment input parameters. We now go into each of the three steps that make up the experiment generation module in more detail.

1. **Uniform Random Number Generation**: Uniform number generators have been studied for several decades [139]. There are many existing RNG approaches and packages available [83]. The types of RNGs and their properties are described in Appendix A. To choose a suitable RNG, one must be aware of several important attributes of RNGs: **a)** random number type, **b)** state sizes, **c)** output value types. The detailed considerations for RNG selection are listed below.

   a) Random number type: We distinguish between two types of random sequences: pseudo-random numbers and quasi-random numbers. Pseudo-random numbers are generated using an algorithm that deterministically produces sequences of random bits. Quasi-random number generators produce a special type of random sequences that satisfy the low-discrepancy criteria [139]. They allow a parameter space to be more evenly covered with less experiments. Choosing quasi-random number generation approaches can help improve the convergence rate of Monte Carlo methods (see Appendix A). The important parallelization consideration is that the sequence must support "skip-ahead", which allows the generator to arbitrarily skip N-elements ahead in the quasi-random sequence such that a sequence can be deterministically generated in parallel.

   b) State-size: the ISO C standard specifies that the rand() function uses a state size of 32-bits, and provides a random sequence with a period of at most 2^{32} (Appendix A illustrates how state is used in random number generation). This is often not large enough for experiment generation in Monte Carlo methods. To solve an option-pricing problem, for example, it may require 16 variables per time step, a simulation scope of 1000 time steps, and the results of 1 million experiments to provide a result of the desired accuracy. To obtain independent and identically distributed experiments, it would require a sequence with a period of at least 2^{34}.

   To resolve this issue, there are special pseudo random generators such as the Mersenne Twister, which provides a period of (2^{19937} - 1). The Mersenne Twister [109] is derived

from the fact that the period length is a Mersenne prime. The generator has an internal state of 624 32-bit integers.

The state size is important to parallelization of random number generators as it determines the data working set of each random number generation stream. For example, on some parallel processors, the 624 32-bit integer internal state required by Mersenne twister method may be too large to keep in L1 cache if we are using the vector unit to produce multiple sequences in parallel.

c) Output value type: Many uniform random sequences are generated in 32-bit unsigned integers through logical bit operations (Appendix A illustrates how a random number generator outputs values). A typical usage model is to convert it into a 32-bit single precision floating-point (SP FP) value from 0.0 to 1.0. One should be aware that there is a loss of precision in this conversion. A SP FP value only has 24-bits of mantissa, and the number of values it can represent between 0.0-0.5 v.s. 0.5-1.0 is very different. Some algorithms may be sensitive to this difference.

2. **Parameter Distribution Conversion** RNG produces uniformly distributed sequences, where input of parameters of the experiments may require a variety of statistical distributions. For example, the volatility of a stock in the market can be modeled with a lognormal distribution in its price; the performance of a computer circuit element affected by manufacturing process variation can be modeled by a normal distribution in its delay.

   In performing the distribution conversion, it is important to provide an array of values to be converted where possible, as many library routines are optimized to use vector instructions to efficiently take advantage of instruction level parallelism in processing an array of inputs. There are many off-the-shelf math libraries that are optimized for distribution conversion. One such example is the Intel Math Kernel Library (MKL) [84].

   For implementations that use quasi-random sequences with good uniformity properties, it is also important for the distribution conversion model to preserve the uniformity of the quasi-random sequences. In [61], it was observed that practically, some conversion methods were better at conserving the uniformity properties and lead to better convergence rate than others.

   The Box-Muller method [21] is based on a polar transformation and uses two of the coordinates of each random point in the unit N hypercube to generate pairs of independent standard normal variates. This method can conserve the uniformity properties of a quasi-random number sequence.

   Figure A.2 illustrates the Box-Muller method. The method takes in two FP values "u0" and "u1", both between 0.0 and 1.0, and produces a pair of floating point values in the normal distribution. One should note that the uniformity property can only be inherited if "u0" and "u1" are produced by two independent random sequences of quasi-random numbers. If they come from the same sequence of quasi-random numbers, the uniformity property will be destroyed.

```
1 float2 BoxMuller(float u0, float u1)  {
2    float r=sqrt(-2 log(u0));
3    float theta=2*PI*u1;
4    return make_float2(r*sin(theta), r*cos(theta));
5 }
```

Figure A.2: A sample implementation of the Box-Muller method

Having to maintain two quasi-random sequences increases the complexity of parallelization. This complexity is manageable as the internal states of a quasi-random number generators such as Sobol are only two 32-bit values (32-bit gray code representation of the index and a 32-bit state) [16].

3. **Multi-parameter Correlation**

   The cross correlation between N parameters is usually handled by a $N \times N$ correlation matrix. This operation can be efficiently handled by BLAS library [29] available on various platforms. Specific specialized handling of matrix operations can follow the **Dense Linear Algebra Pattern**.

Task-centric perspective

Statistical sampling of the parameter space with the Monte Carlo method involves running thousands to millions of independent experiments. With respect to the structure of an implementation of the Monte Carlo method, the experiment execution step often has the most amount of parallelism.

When there is an analytical equation being applied in each scenario, the experiment execution step may be re-factored into a set of matrix operations using existing well-parallelized BLAS library routines [148], or by referencing the **Dense Linear Algebra Pattern**. Figure A.3 illustrates a sample problem reconfiguration into a dense linear algebra routine. The problem is to calculate the values of m financial instruments using k experiments, each consisting of n coefficients and n random variables. We can formulate this problem using Dense Linear Algebra and use the BLAS3 library to efficiently solve it.

For simple experiments, this step is often perceived as the "Map" part of a **MapReduce Structural Pattern**. The experiments can be trivially mapped onto separate execution units using **Task Parallelism Strategy Pattern**. In this case the problem is expressed as a collection of explicitly defined tasks, and the the solution is composed of techniques to farm out and load balance the task on a set of parallel resources. However, the operations to determine value of an instrument in a given experiment might not be as simple as an "add" and "multiply", and some non-linear "max" and "min" operations are required. In that case, it is hard to come up with a simple linear algebra formulation, and we need to use other parallelization strategies to improve performance of the application.

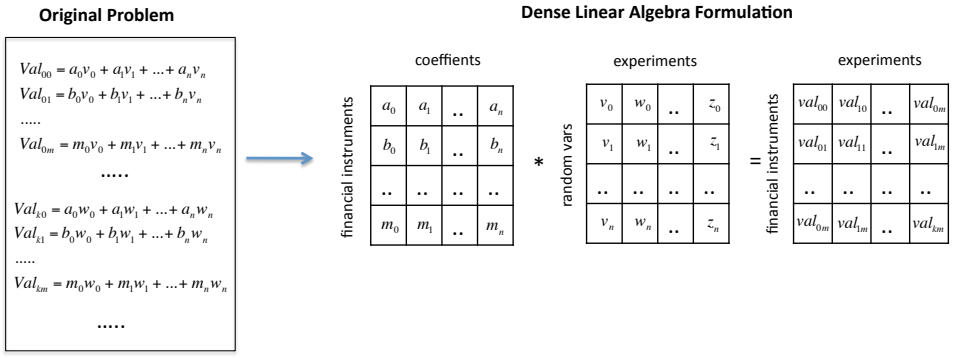


Figure A.3: An example of mapping the original problem of generating val_i values for k experiments, using n coefficients $a...m$ for m financial instruments and n random variables v_i

Data-centric Perspective

A problem to be solved by the Monte Carlo Methods often requires a small set of parameters, involves a large amount of simulation, and outputs a concise aggregation of the results. In the data-centric perspective, we focus on optimizing the memory access patterns between RNG, distribution conversion step, experiment execution, and result analysis.

1. **RNG step**:

   In the RNG step, one usually generate as many sequences of random numbers as there are parameters,as different parameters often require random sequences that are independent and identically distributed. This is often achieved using the same algorithm for random number generation, but using a different random seed for each parameter. This technique is ideal for parallelization using vector instructions with the **SIMD Pattern**. In this case, each SIMD lane will be generating one sequence of random numbers and all lanes will be going through the *state-transition function* and *output function* (as shown in Figure A.8) in a synchronized fashion.

   When the experiments require hundreds to thousands of parameters to be generated, there is enough concurrency in the application to parallelize the RNG on a highly parallel platform. However, when the experiments only require a few parameters, there needs to be a different set of parallelization strategies. With RNG that provides skip-ahead capability, state-transition functions can be constructed that transition the internal state of the RNG n steps ahead. This allows multiple segments of the same random number sequence to be generated at the same time. The Sobol quasi-random number generator allows skip-ahead to be implemented [139].

   With no skip-ahead capability, the most efficient output data layout will be storing different sequences into consecutive memory locations as they are produced. With skip-ahead capability, output data can be produced with consecutive elements storing

data in the same sequence, or with consecutive elements storing data across different sequences.

2. **Distribution conversion step**:

   The distribution conversion step usually is a one input to one output operation where an algorithm converts one value $0 < v \leq 1$ according to the cumulative probability density function of a particular statistical distribution. If this is the case, this step is very flexible in terms of its data structure requirement.

   However, with the Box-Muller method [21], each computation requires two independent input values from two independent random sequences and produces two independent normally distributed values. One approach to meet this special need is to allow each thread process two random sequences at a time. The implication of this approach is that the working set of the algorithm will be doubled, which may increase memory resource contention during execution.

3. **The experiment execution step**:

   The experiment execution step is the most-compute intensive step. If there exist particular high-performance libraries that can accelerate this step well, the data structure should be based on the input requirement of the high-performance library routines.

   When there does not exist high-performance library routines that can implement the experiments efficiently, the experiments can mapped onto separate execution unit using **Task Parallelism Strategy Pattern**. In this case, each experiment can be mapped to a SIMD lane on an execution unit, and it would be more efficient for the RNG to output random sequences from the same generator consecutively in memory.

4. **Result analysis step**: The results from all the experiments can be stored as a vector or can be reduced as they are produced. If the results are stored as a vector, it can be manipulated efficiently with a variety of data parallel operations.

   In conclusion for the data-centric perspective, the most efficient data layout depends on the implementation strategy of the experiment execution step. In this case, **Geometric Decomposition Pattern** can be used to block the computations between the RNG step and the experiment execution step.

A.6 Invariant

Precondition: Experiments are independent and identically distributed to achieve statistically valid sampling.

Invariant: Solution converges as more experiments are conducted.

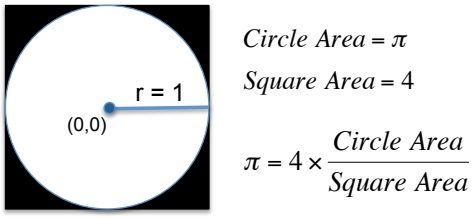


Figure A.4: π estimation problem

A.7 Examples

We illustrate this pattern with four examples: the π estimation, the financial market Value-at-Risk estimation, an option pricing application, and a molecular dynamics application. The first is a pedagogical example demonstrating the implementation structure of a Monte Carlo method. The later examples are real application example.

A.7.1 Example 1: π Estimation

Application Description

This is a simple example to illustrate basic principles of Monte Carlo Methods. The problem is to estimate π by throwing darts at a square inscribed with a circle. We approximate π by calculating percentage of the darts that fall in the circle. Figure A.4 illustrates the problem setup.

We use the formulas: Area of square $= (2r)^2 = 4$
Area of circle $= \pi * r^2 = \pi$

Application Structure

1. **Experiment generation**: Uniformly choose x,y positions in a 2x2 square at random. The center of the square is co-located with the center of circle at $x = 0$, $y = 0$
2. **Experiment execution**: If $x^2 + y^2 < 1$, then point is inside circle
3. **Result aggregation**: $\pi = \frac{4 * \#\ points\ inside}{\#\ points\ total}$

A.7.2 Example 2: Financial Market Value-at-Risk Estimation

With the proliferation of algorithmic trading, derivative usage and highly leveraged hedge funds, there is an increasing need to accelerate financial market Value-at-Risk (VaR) estimation to measure the severity of potential portfolios losses. VaR estimation of portfolios uses the Monte Carlo method. It requires a small set of parameters to setup the estimation process, involves a large amount of computation, and outputs a concise set of risk profiles as the result. As shown in Figure A.5, there are four main steps in the application and the

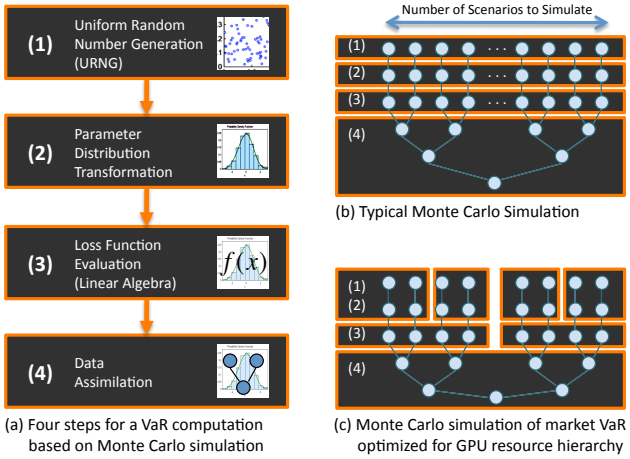


Figure A.5: Solution Structure for the value at risk estimation application

parallelization involves many levels of optimization.

Application Description

The inputs to VaR are assumption about the market and the portfolio, which are encoded in the distribution characteristics of the parameters generated and the coefficients (or influence) of the market risk factors. The outputs are the estimated portfolio prices when they are affected by hypothetical sets of market risk factors.

The application follows the three steps described in the solution section. Figure A.5a illustrates the steps and explicitly describe the experiment generation in two steps. Figure A.5b illustrates the standard implementation, where each white circle represents the computation for each scenario at each algorithmic step. In the standard implementation, each step is performed separately, with the grouping shown with the dark boxes.

Mathematically, consider a portfolio whose value $P(R_1, R_2, \ldots, R_N)$ is a non-linear function in N correlated market risk factors $R_i(t)$ at time t. Under a *delta-gamma* approximation, the portfolio incurs a change of value on the profit and loss (P&L) account due to market movements of the form

$$dP = \sum_{i,j} \underbrace{\Delta_i dR_i}_{delta} + \underbrace{\frac{1}{2} dR_i \Gamma_{ij} dR_j}_{gamma}, \tag{A.1}$$

where the first and second derivatives in the portfolio value with respect to the risk factors are denoted $\Delta_i = \frac{\partial P}{\partial R_i}$ and $\Gamma_{ij} = \frac{\partial^2 P}{\partial R_i \partial R_j}$ - i is the index for each risk factor whose change in value dR_i over a chosen time period is a log-normal random variable. In a typical usage model on a global-scale portfolio, each experiment involves thousands of risk factors describing market parameters such as interest rate trends over time in various currencies.

1. **Experiment generation**: From the **numerical-centric perspective**, the application uses the Sobol' quasi-random low-discrepancy sequence that has been shown in [34] to satisfy *property A* for uniformity for dimensions up to 16900. Box-Muller method [21] is used to transform the uniform random variables to normally distributed random variables, while preserving the distribution uniformity property of the original sequence. The correlation between risk factors is taken care of during the experiment execution. From the **data-centric perspective**, the application requires enough random variables for each thread to be generating its own Sobol quasi-random sequences. The distribution conversion step using the Box-Muller method is merged into Sobol sequence generation step for reduced overhead. This optimization is illustrated in Figure A.5c, where the dark box over step 1 and 2 are merged.

2. **Experiment execution**: From the **task-centric perspective**, the application uses an analytical equation that is being applied in each experiment. The experiment execution can successfully leverage the **Dense Linear Algebra Pattern**, and be re-factored into a sequence of calls to BLAS libraries to estimate the risk exposure. Further optimizations involving reformulating the portfolio loss function to use a precomputed, deterministic part of the delta term. This precomputation enables the bottleneck matrix-matrix computation to be replaced with a matrix-vector operation, reducing computation by a factor of $O(N)$, where N is the number of risk factors. The computation can be further optimized with the **Geometric Decomposition Pattern**, where the problem can be broken down to blocks that fit into memory. This is illustrated in Figure A.5c, where the dark box over scenarios are separated into blocks.

3. **Result aggregation**: The results from the experiment with various market conditions are sorted and the thresholds at 5 percentile or 1 percentile worst outcome are used to provide the VaR estimate for the amount of 1-in-20 or 1-in-100 chance of portfolio loss.

A.7.3 Example 3: Option Pricing

We are trying to determine the present-day-value of a contract with this scenario:
"In 3 months' time a buyer will have the option to purchase Microsoft Corp. shares from a seller at a price of $25 per share."

The key point is that the buyer has the option to buy the shares. Three months from now, the buyer may check the market price and decide whether or not to exercise the option. The buyer would exercise the option if and only if the market price were greater than $25, in which case the buyer could immediately re-sell for an instant profit.

This deal has no downside for the buyer once he/she pays the up-front cost of purchasing the option. Three months from now the buyer will either make a profit or walks away. The seller, on the other hand, has no potential gain and an unlimited potential loss. To compensate, the cost for buyer to enter into the option contract must be carefully computed. The process in which we determine the cost of the up-front payment is the option-pricing

```
1 for i = 0 to M     1 do
2    t := S * exp ((r - 0.5*sigma^2)* T +
3                 sigma * sqrt(T) * randomNumber())
4    trials [ i ] := exp(-r * T )  max{t - E, 0}
5 end for
6 mean := mean( trials )
7 stddev := stddev( trials , mean)
```

Figure A.6: A Sequential Monte Carlo simulation of the Black-Scholes model

problem[2].

Merton's work expanded on that of two other researchers, Fischer Black and and Myron Scholes, and the pricing model became known as the Black-Scholes model. The model depends on a constant (Sigma), representing how volatile the market is for the given asset, as well as the continuously compounded interest rate r.

The Monte Carlo Method approach takes M number of trials as input, where M could be 1,000 to 1,000,000 large depending on the accuracy required for the result. The pseudo code for one individual experiment is shown in Figure A.6, where: S is the asset value function, E is the exercise price, r is the continuously compounded interest rate, $Sigma$ is the volatility of the asset, T is the expiry time, and M is the number of trials.

The pseudo code also uses several internal variables:

- trials : array of size M , each element of which is an independent trial (iteration of the Black-Scholes Monte Carlo method)
- mean: arithmetic mean of the M entries in the trials array
- randomNumber(), when called, returns successive (pseudo)random numbers chosen from a Gaussian distribution.
- mean(a) computes the arithmetic mean of the values in an array a
- stddev(a, mu) computes the standard deviation of the values in an array a whose arithmetic mean is mu.
- confwidth: width of confidence interval
- confmin: lower bound of confidence interval
- confmax: upper bound of confidence interval

The solution structure is:

[2]In 1973, Robert C. Merton published a paper [28] presenting a mathematical model, which can be used to calculate a rational price for trading options. (He later won a Nobel prize for his work.) In that same year, options were first traded in the open market.

- **Random Number Generation**
  In a parallel implementation, the computation in line (2-3) can be computed by concurrent threads of execution. The function randomNumber() must be generating uncorrelated (pseudo)random number sequences using components such as Mersenne Twister PRNG or SPRNG Ver4.0 for the concurrent threads of execution.

- **Scenario Simulation**
  The execution can be implemented to target multiple levels of parallelism in modern highly parallel execution.

- **Result Aggregation**
  Line (5-6) in the algorithm aggregates the results of the trials and provide a summary of the simulation results.

A.7.4 Example 4: Molecular Dynamics

In the field of molecular dynamics, Monte Carlo methods are used to statistically model the mechanics of the molecular system rather than explicitly reproducing the dynamics of the system. In this example, the *Metropolis Monte Carlo* simulation is a Markov Chain Monte Carlo method used to obtain a sequence of random samples from a probability distribution for which direct sampling is very difficult. The method employs a Markov Chain procedure in order to determine a new state for a system from a previous one. According to its stochastic nature, this new state is accepted at random. Metropolis Monte Carlo can be used to compute the set of canonical ensemble averages for the Ising model [3] in the field of Molecular Dynamics.

More formally, the Metropolis Monte Carlo is a computational approach for generating a set of N configurations of the system $\eta_1, \eta_2, \eta_3 ... \eta_N$ such that $\lim_{N \to +\infty} \frac{N_\eta}{N} = P(\eta)$ where $P(\eta)$ is a given probability distribution (the Boltzman distribution) and N_η is the number of configurations η (e.g. the number of configurations generated with particular spins $S(\eta)$ of the Ising model).

The algorithm is shown in Figure A.7, and evolves any arbitrary distribution toward the equilibrium distribution where $\frac{N_{eta}}{N} = P(\eta)$. It is able to generate an ensemble of configurations with the probability distribution $P(\eta)$ by simply computing the probability ratios. Thus, this is an efficient method for computing a sequence of configurations leading to system equilibrium.

- **Random Number Generation**
  Random numbers are used to determine if the system takes on a new configuration in the next simulation step.

[3] The Ising model is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables representing spins. The spins can be in one of two states and they are arranged in a lattice or graph. If a molecular system reaches equilibrium by going through a series of configurations drawn randomly from a probability distribution of the configurations. A canonical ensemble of a system is the statistical ensemble used to represent probability distribution of microscopic states of the system.

```
Pick any configuration eta_n

For N where N is large
   Pick a trial configuration eta_t
   Compute ratio R=eta_n/eta_t
   Generate Random number p  (between 0 and 1)
   If p<=R
      eta_next = eta_t;
   Else
      eta_next = eta_n;
```

Figure A.7: Pseudo code fot the Metropolis Monte Carlo algorithm

- **Scenario Simulation**
  The scenario being simulated in this case is randomly selecting the configuration η of the system by sampling from the Boltzman probability distribution. The new configuration is then used at random, by comparing the ratio of the probabilities of two consecutive configurations to a random number.

- **Result Aggregation**
  Result of this simulation is to determine if the system remains at an equilibrium going through a sequence of configurations subject to the experiment constraints, i.e. if the limit is satisfied.

A.8 Known Uses

1. Design and visuals

   - Global illumination computations
     Monte Carlo methods have also proven efficient in solving coupled integral differential equations of radiation fields and energy transport. These methods have been used in global illumination computations which produce photorealistic images of virtual 3D models, with applications in video games, architecture, design, computer generated films, special effects in cinema [156, 35] .

2. Telecommunications

   - Quality of Service Simulations
     Wireless network must be designed to handle a wide variety of scenarios depending on the number of users, their locations and the services they want to use. The network performance is evaluated with a Monte Carlo engine generating user states. If service levels are not satisfactory, the network design goes through an optimization process [117].

3. Physical Sciences

   - Statistical Physics
     Monte Carlo methods are used in Molecular modeling as an alternative for molecular dynamics. It is also used to compute statistical field theories of particle models [108].
   - Particle Physics
     Monte Carlo methods are used for designing detectors for understanding and comparing experiment data to theoretical prediction in large galaxy modeling [20].

4. Circuit Design

   - Statistical Timing Analysis
     With continued semiconductor device scaling, devices are becoming so small that their parameter variation, if not managed properly, are causing chip designs to fail. Monte Carlo Methods can be used where the random sequences are generated to model device parameter variations and experiments are run to measure the expected performance of the chip designs [27].

A.9 Related Patterns

Structural Patterns

- MapReduce
  To manage experiment execution and result aggregation steps we can use the MapReduce pattern. Each experiment is mapped onto an execution unit and results are reduced to produce a global solution.

Computational Patterns

- Dense Linear Algebra
  The cross correlation between N parameters of the model are usually handled by a N×N correlation matrix. This operation can be efficiently handled by BLAS library available on various platforms. Specific specialized handling of matrix operations can follow the Dense Linear Algebra Pattern.

Parallel Algorithm Strategy Patterns

- Task Parallelism
  This pattern is used for managing experiment execution. Each experiment is a task. Tasks are independent and can be executed in parallel.

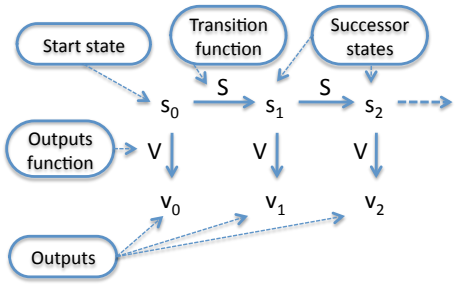


Figure A.8: A general random number generation algorithm structure

- Data Parallelism
  We can view the experiment execution as the same operation being performed on different pieces of data (the experiment parameters). This can be efficiently mapped to data-parallel platforms using the Data Parallel Algorithm Strategy Pattern.

Implementation Strategy Patterns

- SPMD
  We can map each experiment to a set of instructions (single program), that executes on separate sets of data (multiple data).

Concurrent Execution Patterns

- SIMD
  This pattern can be used for grouping experiment execution. We perform one experiment (one set of instructions) on multiple pieces of data.

A.10 Notes on: Random Number Generation

There are three types of Random Number Generators (RNG): true-random RNG, pseudo-random RNG, and quasi-random RNG. True-random RNG is based on some physical phenomenon that is expected to be random, and the sequence generated is believed to be truly random and cannot be reproduced. Pseudo-random RNG and quasi-random RNG are based on algorithms that use computational methods to produce long sequences of apparently random numbers, usually determined by an initial value (or seed). Given the same seed, these sequences are reproducible. For computational problems, reproducibility is crucial for software development. We focus on pseudo-random RNG and quasi-random RNG.

Both pseudo-random RNG and quasi-random RNG are generated using state machines with: a) *a start state* (or random seed), b) *a state transition function*, and c) *an output function*, shown in Figure A.8. The difference between the two RNGs lie in the state transition function induced the output sequence property. Figures A.9 and A.10 show the

output model of pseudo-random and quasi-random values. Quasi-random RNG such as the Sobol sequence generates low-discrepancy sequences that conform to the uniformity condition known as *property A* [139]. This means that they have a specific uniform coverage of the parameter domain, which if used, can lend to faster convergence rate for the Monte Carlo method. In a quasi-random generator such as the Sobol sequence, this is achieved in the state transition function by using a set of binary fraction states called direction numbers. This function toggles the binary representation of a value in the sequence with varying frequencies, to achieve the expected parameter space coverage.

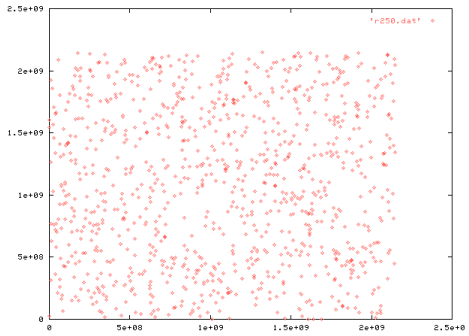


Figure A.9: A pseudo-random distribution

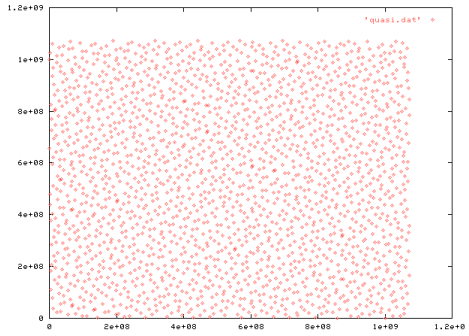


Figure A.10: A quasi-random distribution