A Framework for Productive, Efficient and Portable Parallel Computing

Ekaterina I. Gonina

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-216
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-216.html

December 17, 2013

A Framework for Productive, Efficient and Portable Parallel Computing

by

Ekaterina I. Gonina

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor Armando Fox
Professor David Wessel

Fall 2013

A Framework for Productive, Efficient and Portable Parallel Computing

Abstract

A Framework for Productive, Efficient and Portable Parallel Computing

by

Ekaterina I. Gonina

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Kurt Keutzer, Chair

Developing efficient parallel implementations and fully utilizing the available resources of parallel platforms is now required for software applications to scale to new generations of processors. Yet, parallel programming remains challenging to programmers due to the requisite low-level knowledge of the underlying hardware and parallel computing constructs. Developing applications that effectively utilize parallel hardware is restricted by poor programmer productivity, low-level implementation requirements, and limited portability of the application code. These restrictions in turn impede experimentation with various algorithmic approaches for a given application. Currently, the programming world is divided into two types of programmers: application writers who focus on designing and prototyping applications and algorithms, and efficiency programmers who focus on extracting performance for particular compute kernels. The gap between these two types of programmers is referred to as "the implementation gap".

In this dissertation, we present a software environment that aims to bridge the implementation gap and enable application writers to productively utilize parallel hardware by reusing the work of efficiency programmers. Specifically, we present PyCASP, a Python-based software framework that automatically maps Python application code to a variety of parallel platforms. PyCASP is an application-domain-specific framework that uses a systematic, pattern-oriented approach to offer a single productive software development environment for application writers. PyCASP targets audio content analysis applications, but our methodology is designed to be applicable to any application domain. Using PyCASP, applications can be prototyped in Python code and our environment enables them to automatically scale their performance to modern parallel processors such as GPUs, multicore CPUs and compute clusters. We use the Selective Embedded JIT Specialization (SEJITS) mechanism to realize the pattern-based design of PyCASP in software. We use SEJITS to implement PyCASP's components and to enable automatic parallelization of specific audio content analysis application

patterns on a variety of parallel hardware. By focusing on one application domain, we enable efficient composition of computations using three structural patterns: MapReduce, Iterator and Pipe-and-Filter.

To illustrate our approach, we study a set of four example audio content analysis applications that are architected and implemented using PyCASP: a speaker verification system, a speaker diarization system, a music recommendation system and a video event detection system. We describe the detailed implementation of two computational components of PyCASP: a Gaussian Mixture Model (GMM) component and a Support Vector Machine (SVM) component and their use in implementing the example applications. We also analyze composition of computations using the three structural patterns and implement the available optimizations for composing computations in audio analysis applications.

We evaluate our approach with results on productivity and performance using the two computational components and the four example applications. Our results illustrate that we can prototype the full-functioning applications in Python using $10 - 60\times$ less lines of code than equivalent implementations using low-level languages. Our PyCASP components and example applications achieve and often exceed the efficiency of comparable hand-tuned low-level implementations. In addition to specialization, adding the optimizations for composing components in these applications can give up to 30% performance improvement. We show that applications written using PyCASP can be run on multiple parallel hardware backends with little or no application code change. PyCASP also enables applications to scale from one desktop GPU to a cluster of GPUs with little programmer effort. Combining all of the specialization and composition techniques, our example applications are able to automatically achieve 50-1000$\times$ faster-than-real-time performance on both multi-core CPU and GPU platforms and 15.5$\times$ speedup on 16-node cluster of GPUs showing near-optimal scaling.

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisor Kurt Keutzer, for his guidance and support throughout the process of getting a PhD at UC Berkeley. I also want to sincerely thank Dr. Gerald Friedland for his thoughtful guidance and valuable advice about the audio analysis application domain. Throughout my career at UC Berkeley, I have had the privilege of working with exceptional PhD students. In particular, I would like to thank my co-authors, Henry Cook, Shoaib Kamil, Eric Battenberg, Penporn Koanantakool and Michael Driscoll for devoting their time to develop various parts of the PyCASP framework and the example audio applications. I would like to thank Jike Chong for teaching me the ropes of GPU programming and guiding me through the first years of graduate school research. I also want to thank David Wessel and Armando Fox for serving on my dissertation committee and giving advice on improving the manuscript. Finally, I want to express my sincere gratitude to my parents, who have supported me throughout my whole life and especially to my father for instilling in me the love for science and engineering.

Chapter 1

Introduction

Historically, processor capabilities in the computing industry have been driven by Moore's Law, which states that the number of transistors that can be put on a processor die doubles every 18 months [73]. Moore's Law is based on the trend that the transistor size decreases every generation, allowing hardware manufactures to put more transistors on one die. Thus, following this trend, since the mid 1980s, processor clock speeds increased exponentially every year, as shown in Figure 1.1. This trend, in turn, enabled software applications to see an increase in performance for "free", i.e. without any programming effort - every two years a new, faster processor automatically improved application performance.

However, in the early 2000s, this growth was disrupted by the physical limitation on energy required to power more transistors on one die. As shown in Figure 1.1, the peak processor frequency could not scale above 3GHz, due to the power limitations; the computing industry hit what is now referred to as the *"Power Wall"*. While Moore's Law remained in effect, instead of putting more transistors on one processor die, the computing industry shifted to putting *multiple processor cores on one die*. Thus, the power constraint caused a shift in the industry to designing and manufacturing *parallel* hardware.

The shift to parallelism in the computing industry enabled higher performance as parallel processors now allowed for higher computational throughput at lower power costs. However, this shift was (and still is) absolutely *disruptive* to the software industry. Instead of automatically seeing performance improvement with every new generation of processors, applications now have to be *explicitly rewritten* to take advantage of the new hardware capabilities.

When efficiently mapped onto parallel platforms, computationally-demanding, large-scale and low-latency applications can achieve several orders of magnitude in performance improvement allowing for real-time processing and scaling to large datasets [15] [74] [23]. Table 1.1 shows the potential in performance improvement when going from high-level single-core CPU code (typically preferred by application programmers) to low-level parallel code. Reimplementing algorithms

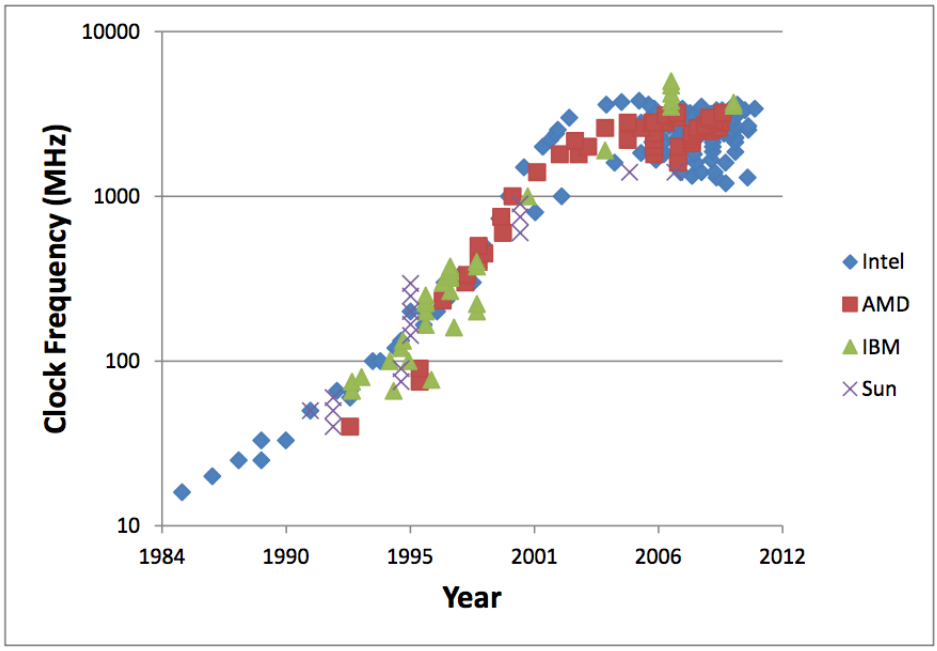

Figure 1.1: Scaling of the processor clock speeds.

in low-level code from high-level code for a single sequential CPU core alone can give at least one to two orders of magnitude performance improvement [15]. Furthermore, when porting the application code from sequential low-level code to multi-core CPU code, applications typically gain 2-10× in performance [111]. When porting sequential single core CPU code to GPUs we can see anywhere from 10× to 200× performance improvement [23, 15, 16]. Finally, distributing the computation across a cluster of parallel machines can give further one or two orders of magnitude in performance improvement [44]. Thus, starting from a high-level, sequential single core CPU application implementation and combining several parallelization techniques we can boost application performance by *several orders of magnitude*. This performance improvement is extremely enticing - higher application performance can mean significant improvement in application accuracy and user experience [26]. In addition, in order to scale applications to new hardware, we must implement applications to explicitly utilize parallelism in the new processors.

While the benefits of parallelizing applications are very appealing, programming parallel hardware is a challenging task. Writing low-level code takes a significant amount of programmer effort and expertise, not available to every application programmer. In order to see application performance improve with new generations of hardware, application developers are now required to understand not only their application domain, but also the new parallel computing challenges,

Implementation Shift	× Perf Imp*
High-level single core CPU -> Low-level, single core CPU	$10 - 100\times$
Low-level single core CPU -> Low-level multi-core CPU	$2 - 10\times$
Low-level multi-core CPU-> Low-level GPU	$10 - 200\times$
Low-level multi-core CPU -> Low-level multi-node cluster	$10 - 100\times$

Table 1.1: Performance improvement possible when reimplementing applications in lower-level languages and explicitly utilizing parallel hardware. *× Perf Imp = Performance Improvement Factor

previously unfamiliar to them. For example, application programmers now need to understand the intricate details of parallel hardware architectures, parallel programming environments and the synchronization and communication mechanisms of parallel tasks. The required focus on the details of parallelism and the hardware architecture significantly impedes programmer productivity. Furthermore, because of the complexities associated with writing parallel code, applications are typically written and tuned for one particular platform. This dependency on particular specifications of the target platform makes application code non-portable; porting the application to a different platform requires at least a partial application code rewrite. These parallel programming challenges make it very difficult for application programmers to develop applications that can utilize new parallel hardware.

Because of the challenges of developing parallel applications, the current practice of creating and deploying software applications is divided between application developers and expert parallel programmers, creating a dichotomy that we refer to as *The Implementation Gap*. Figure 1.2 shows a typical setup of programmers in an application development project. On one end, there is the *application domain expert*, who is developing the application for the target user. Application domain experts are deeply familiar with the application domain (for example speech recognition or computer vision), they work to advance the state of the art in this domain under a set of implementation constraints such as recognition accuracy or recommendation latency. They are typically not familiar with the details of the underlying hardware or parallelization strategies, but would rather focus on analyzing the data, implementing particular algorithms, and experimenting with algorithm parameters and alternative data representations. For their tasks, application developers prefer to use high-level languages such as MATLAB or Python to quickly prototype their applications and algorithms. On the other end, there are the *expert parallel programmers* who are deeply familiar with the underlying parallel hardware and low-level languages used to program the hardware and extract best performance for particular computations. They prefer to work in low-level languages such as C and CUDA and focus on enabling low-level optimizations in

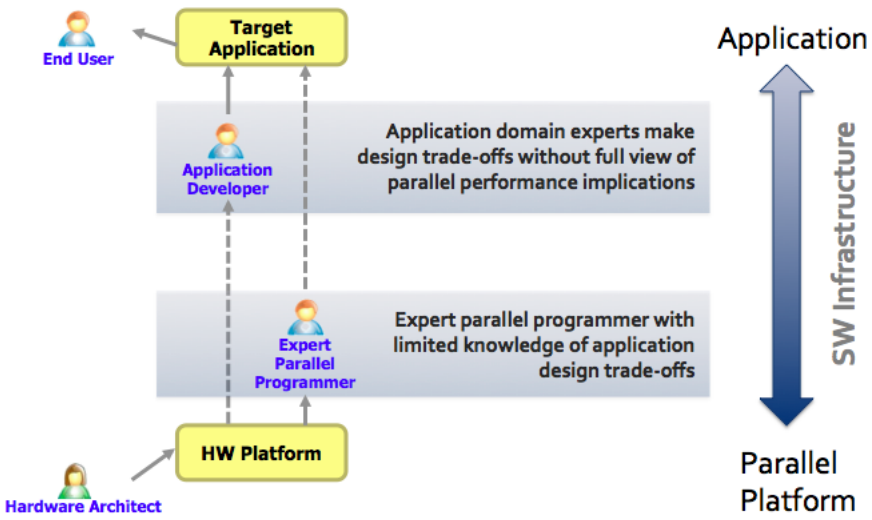

Figure 1.2: The Implementation Gap in application development.

their code to fully utilize and use all intricate architectural features of parallel hardware. Expert parallel programmers, however, typically lack broad application domain understanding and thus cannot take advantage of domain knowledge to improve application performance.

In order to develop efficient applications that utilize underlying parallel hardware, expert parallel programmers need to work together with the application domain experts to combine their application and parallel efficiency knowledge and develop high-performing, parallel applications that achieve state-of-the-art accuracy. This communication is *required* for the computing industry to move forward and enable scalability of applications on future hardware platforms. One way to solve this problem is to use a *software environment* that enables application developers and efficiency programmers to work together to create efficient applications that utilize parallel hardware. In this dissertation, we set out to answer the following question:

How can we build a software environment to bridge the implementation gap and enable application writers to productively utilize parallel hardware and develop efficient, scalable, and portable applications?.

1.1 Research Goals

To answer the above question, we focus on answering more specific questions and set concrete goals:

1. **Bridging the Gap**: How can a software environment bridge the implementation gap and allow for communication between application developers and expert parallel programmers? We aim to show how we can achieve communication between the two types of programmers using a software environment.

2. **Scope**: What is the scope of such a software environment? What components and abstractions should such a software environment support? We aim to define the scope for a specific application domain and show how the approach can be generalized to other domains.

3. **Vocabulary**: What language / common vocabulary should the software environment use such that both types of programmers can understand and use it? We aim to find such a language that facilitates both programmer's understanding of the software environment and its functionality.

4. **Productivity**: How can our software environment provide productivity of high-level languages? We aim for 10-100× lines-of-code reduction compared to applications written in a low-level efficiency language.

5. **Efficiency** How can our software environment allow for efficient performance of low-level implementations? We aim to obtain within 30-50% of hand-coded performance and within an order of magnitude faster performance than pure Python code.

6. **Portability** How can our solution ensure that the same application code is portable to new generations of processors and across a variety of hardware platforms? We aim to demonstrate the same application running on a multicore CPU, a GPU and a computer cluster without significant application code change.

7. **Scalability** How can our software environment allow programmers to go from experimentation on a single node to a cluster of processors, from processing a sample subset of data to the entire dataset? We intend to show application scaling from one compute node to a cluster of compute nodes, using our same framework, without significant programming effort.

8. **Flexibility** Even when achieving the above requirements, how flexible is such a software environment when designing and prototyping different applications? We aim to investigate the advantages and disadvantages of restricting the application scope of the framework to a particular application domain in terms of programmer flexibility.

9. **Composition**: How can our software environment support composition of computations to further improve application performance?

The list above presents a very broad set of topics and goals. In this thesis, we focus in depth on bridging the implementation gap and providing application programmer productivity and application efficiency and portability. In addition, we outline the methodology to define the scope of the framework and enable composition of computations. Finally, we provide a brief discussion on flexibility of our approach and ways of achieving scalability to serve as a a guide for future work.

The main contribution of this thesis is the proposal of a systematic way to bridge the implementation gap using an application-domain-specific, pattern-oriented approach for the design of the framework and the Selected Embedded Just-in-Time Specialization (SEJITS) mechanism to realize this design in software. We present our proposed solution in the form of *PyCASP: a pattern-oriented, application-domain-specific, specialization framework*.

We use the audio content analysis domain as our target application domain. Audio content applications implement algorithms for extracting information from audio data such as speech, music and video soundtracks. Audio analysis applications have specific latency and throughput constrains that make it a lucrative target for demonstrating performance improvement using parallelization. We use four state-of-the art audio analysis applications to guide us through the design and implementation of our proposed solution: a speaker verification system, a meeting diarization system, a music recommendation system and a video event detection system. We outline the design process of our framework, the realization of the design in software as well as specific implementation details that enable us to achieve our goals. We use the four selected state-of-the-art to illustrate our approach throughout this work.

1.2 Thesis Contributions

The contributions of this thesis work are as follows:

- We propose using a *pattern-oriented design* for designing our software environment to bridge the implementation gap. The pattern-oriented design uses application, computational and structural patterns to define the scope and aid in defining a comprehensive and modular software solution. Patterns are used to construct application software architectures. A framework designed using patterns can support a variety of applications in the selected application domain.

- We propose using an *application-domain-specific approach* for designing the software solution to bridge the implementation gap. By focusing on one application domain, we restrict the scope of the framework to particular

patterns present in this domain resulting in a clear scope and a comprehensive software environment.

- We propose restricting the scope of the software environment to one application domain to aide in *composition of computations*. By focusing on one application domain and analyzing the types of compositions that are common this domain, we aim to design a software framework that can understand and optimize the composition of computations to enable higher application efficiency.

- We propose using the *Selected Embedded JIT Specialization (SEJITS)* approach for realizing our pattern-based design in software. SEJITS enables programmer productivity and flexibility as well as application efficiency and portability. We propose using SEJITS to realize application patterns in software to enable domain experts to automatically utilize parallel hardware.

- We propose and develop a framework called *PyCASP (Python-based Content Analysis using SPecialization)* that uses the proposed approaches. Specifically, PyCASP uses the pattern-oriented, application-domain-specific specialization approach to enable application programmers to write applications that automatically utilize parallel hardware.

- We describe how PyCASP can allow for *high application programmer productivity* by being embedded in Python as well as employing the pattern-oriented design methodology. We hope that using these approaches, we can achieve productivity of high-level languages in our software environment.

- We describe how PyCASP can allow for *high application efficiency* by using the SEJITS approach. We hope that using specialization applications that are written in a highly-productive environment, we can achieve efficiency that is on par with the efficiency of hand-tuned low-level implementations.

- We describe how PyCASP's components can be *portable to a variety of parallel hardware* including multi-core CPUs, NVIDIA GPUs and clusters. We aim to enable applications written using PyCASP to run on many platforms with little or no any application code change.

- We describe how we can *allow the application developers to use a computer cluster* to run their applications. We aim to enable application programmers to port their applications to use more nodes of a cluster of machines as well as process more data by enabling the MapReduce functionality in PyCASP.

- We propose that the *three structural patterns* that are common in the audio content analysis applications are sufficient for describing a large variety of

applications in the domain. We aim to show that these three structural patterns are sufficient and flexible enough to express the common compositions of components in audio content analysis applications.

- We aim to show that we can implement a *large variety of state-of-the-art audio content analysis applications* using our software environment. We look at a variety of applications ranging from speaker verification to video event detection and classification. We aim to show that these applications can all be written in Python and use PyCASP to automatically utilize parallel hardware.

1.3 Thesis Outline

- Chapter 2 describes the background information for this work. It describes the programmer challenges and the specifics of parallel hardware architectures and programming environments related to and used in this work. The chapter also describes related work on alternative approaches to application development and bridging the implementation gap. It then gives details on the background for specific methodologies that we propose use to develop our software solution

- Chapter 3 describes the audio content analysis application domain. It gives a detailed overview of the four example audio analysis applications and the algorithms they employ.

- Chapter 4 proposes using a pattern-oriented design for our software framework solution. It discusses the pattern-oriented design methodology and discusses the specific pattern-mining process to analyze the audio content analysis applications. It describes the application, computational and structural patterns in the audio content analysis domain. The chapter then discusses how we can use the patterns to construct software architectures of the sample audio analysis applications.

- Chapter 5 proposes using the specialization approach to realize the pattern-based design described in Chapter 3 in software. Specifically, the chapter discusses the Selected Embedded JIT Specialization (SEJITS) approach that we propose to use in our solution. It discusses how application patterns can be realized in software using SEJITS. It then gives details of the implementation and specialization mechanisms of two specific application patterns: the Gaussian Mixture Model (GMM) Parametric Clustering component and the Support Vector Machine (SVM) Linear Classification component.

- Chapter 6 proposes restricting the framework's scope to enable composition of computational components in applications developed using our framework. The chapter focuses on the three structural patterns we mined in Chapter 3 (Pipe-and-Filter, Iterator and MapReduce) and discusses how we can enable composition using these patterns. The chapter describes how each pattern can be used to compose computations in applications using the sample audio applications as a guideline.

- Chapter 7 shows how the sample audio content analysis applications can be implemented using the proposed framework. Specifically, it describes how the speaker verification application, speaker diarization system, music recommendation system and the video event detection system can be implemented in Python using PyCASP. Each section describes the implementation of the application algorithm using PyCASP illustrating the use of its components and composition mechanisms.

- Chapter 8 evaluates the software environment solution against the goals described in Chapter 1 and describes the results of this research. The chapter talks about the results on productivity, efficiency and portability of each example application as well as individual performance of each component of PyCASP. It then describes the results on performance of applications from using the composition mechanisms.

- Chapter 9 concludes this work and presents a discussion of the drawbacks and some directions for future work.

Chapter 2

Background

2.1 Programmer Challenges

When writing applications, programmers have to trade-off productivity with efficiency and portability. On one end, application writers want to stay productive, develop applications rapidly and find bugs quickly. On the other end, developers want applications that are performance-efficient and portable. Developing efficient applications requires low-level implementation and careful tuning of the algorithm, which significantly impedes productivity. In addition, writing portable code requires rewriting the application to run on different platforms, which also heavily impacts productivity.

2.1.1 Productivity

Productivity measures the programming effort needed for an application writer to develop a full-functioning application. Measuring software productivity is a difficult undertaking as there is no standardized measurement system or criteria for evaluating productivity and most conclusive arguments can be made only about large-scale software systems [95]. Several studies indicate that software reuse is one of the most contributing factors to software productivity [11]. While acknowledging that extensive data collection and knowledge-based approaches are needed to fully evaluate productivity of software development, in this work, we restrict our evaluation to comparing the lines of code required to implement applications as well as code reuse. While this evaluation approach is fairly surface-leveled, it provides a quantitative way to estimate productivity and serves as a good initial estimate of productivity improvement.

When analyzing programmer productivity, case studies have found that high-level scripting languages such as MATLAB or Python allow programmers to express the same programs in $3-10\times$ fewer lines of code and in one fifth to one third the development time [22, 52, 84]. Thus, high-level scripting languages en-

```
def kernel(inArray, outArray):
  for pt in inArray.interior():
    for x in pt.neighbors(radius=1):
      outArray[pt] += 1/6 * inArray[x]
```

Figure 2.1: Python code for the 3D heat equation. From S. Kamil [59].

able programmer productivity; they express computations at a high-level, provide libraries and abstractions to hide implementation details and reuse code. They also provide a set of tools to aide in application development such as plotting and data analysis tools as well as file management mechanisms that are easy to use. To develop an application prototype in a productivity language, programmers do not have to think about underlying hardware architecture but rather focus on the application and the algorithm at hand. For example, Figure 2.1 shows a sample Python code snippet for the 3D heat equation; the entire computation is expressed in 4 lines of Python code. Productivity languages also typically do not require compilation or linking, freeing the programmers from infrastructure concerns as well.

2.1.2 Performance

Performance measures the efficiency of a particular application running on a given hardware platform. As mentioned in Chapter 1 and shown in Table 1.1, when we look at the landscape of parallel programming to compare application performance, there are several orders of magnitude performance improvement available when going from a high-level sequential implementation to combining several parallelization techniques and using low-level languages.

While low-level, parallel implementations of computations are very efficient, they present some significant drawbacks. They take significant amount of programming effort, are difficult to understand and maintain and are non-reusable and non-composable. Figure 2.2 shows a code snippet of the 3D heat equation implemented in C for efficiency, to contrast the one shown implemented in Python and shown in Figure 2.1. Applications written in C/C++ contain *orders of magnitude* more lines of code as the equivalent implementations in Python. The application code is also more cumbersome to understand and maintain. These problems become a lot more signficant when we move to parallel implementations with Pthreads, OpenMP or CUDA. In order to extract best performance out of a hardware platform, application code needs to be heavily tuned with optimizations such as loop unrolling, operand reordering and memory usage optimization such as zero-padding. Thus, while low-level, heavily-tuned, parallel implementations pro-

```
int c2;
for (c2=chunkOffset_2;c2<=255;c2+=128) {
 int c1;
 for (c1=chunkOffset_1;c1<=255;c1+=64) {
  int c0;
  for (c0=chunkOffset_0;c0<=255;c0+=256) {
   int b2;
   for (b2=c2 + threadOffset_2;b2<=c2 + 127;b2+=128) {
    int b1;
    for (b1=c1 + threadOffset_1;b1<=c1 + 31;b1+=16) {
     int b0;
     for (b0=c0 + threadOffset_0;b0<=c0 + 255;b0+=256) {
      int kk;
      for (kk=b2 + 1;kk<=b2 + 128;kk+=1) {
       int jj;
       for (jj=b1 + 1;jj<=b1 + 16;jj+=1) {
        int ii;
        for (ii=b0 + 1;ii<=b0 + 256;ii+=1) {
         dst[_dst_Index(ii - 1,jj - 1,kk - 1)] = ...;
}}}}}}}}}
```

Figure 2.2: C code for the the 3D heat equation. From S. Kamil [59].

vide high performance, they require substantial programmer effort, significantly impeding productivity.

2.1.3 Portability

Productivity measures how many different hardware platforms one application can run on without application code change. As mentioned in the previous subsection, in order to extract the most efficiency out of a hardware platform, application code needs to be heavily optimized and tuned to that specific hardware architecture. This makes source code developed for one particular platform not portable to a different platform. For example efficiency-level code written for a single desktop GPU platform is not portable to a multi-core CPU platform or even to a previous generation of the GPU. In order to run the application on a different platform, current practice is to rewrite the application source code for every platform. This amount of effort is typically not feasible for a typical software development project.

Furthermore, when one wants to port an application from a single-node platform to a cluster of machines, application scalability becomes an issue. In order to enable application scalability to a larger dataset, application code needs to be rewritten to explicitly scale to clusters of machines. To achieve portability and scalability of their applications to clusters of processors, programmers need to rewrite their applications using distributed programming frameworks such as Hadoop [107] and MapReduce [33]. In addition, commodity clusters and datacenters typically consist of a large set of multi-core nodes, thus requiring *composition*

of parallel code using the distributed frameworks and the low-level parallel implementations for a single multi-core node, further impeding application programmer productivity.

2.2 Parallel Programming Challenges

In order to scale application performance to new parallel hardware, applications have to explicitly express parallelism in their implementation. This requirement makes application development very difficult to application writers due to several challenges, described below.

2.2.1 Identifying available parallelism

Application algorithms can exhibit several possible parallelism opportunities. For example if the algorithm operates on a multi-dimensional data structure, we can focus on identifying dimensions that can be processed independently (referred to as "data-parallelism"), or there may be multiple tasks in the algorithm that are independent and can be executed simultaneously (referred to as "task-parallelism"). Some algorithms are, on the other hand, inherently sequential. For example, in a path-finding algorithm for a graph, each algorithm step depends on the results of the previous step, thus disallowing parallelization across iterations of the algorithm. In order to parallelize an application, one needs to identify independent computations as well as operations and data elements that can be operated on in parallel. It is also important to understand the *degree* of parallelism of each set of independent tasks or data elements. For example, there may be two or two thousand independent tasks in an algorithm. Parallelization strategies will differ depending on the degree of parallelism in addition to the type of parallelism available.

2.2.2 Understanding the variety of parallel hardware

With the computing industry shifting toward parallel processors, there is a large variety of parallel hardware available for application developers. The most common consumer-facing parallel computing platforms include multi-core CPUs, Graphics Processing Units (GPUs) and commodity clusters. Each parallel platform has its own hardware architecture specifications and execution models. Understanding the intricacies of the parallel hardware architectures is important in order to write efficient parallel code. For example, failure to fully understand the details of the memory hierarchy, task scheduling or communication mechanisms between processing units of a particular platform can result in losing several orders of magnitude in application performance. In addition, it is not always clear which

parallel platform is best suited for a particular application, further exacerbating the problem of hardware selection.

2.2.3 Understanding the variety of parallel programming frameworks

In addition to understanding the details of parallel platforms, application programmers must also understand the different parallel programming languages and frameworks that can be used to program these platforms. In order to utilize parallel hardware, it is necessary to write the applications in low-level frameworks such as OpenMP [78], Pthreads [67], CUDA [75] and OpenCL [77] (discussed in more detail later in this chapter). These frameworks support different programming and threading models, some allow for different amount of control of synchronization and communication mechanisms between parallel threads of execution. Furthermore, writing applications using these low-level frameworks typically results in cumbersome, hard-to-maintain, platform-specific application code, making application maintenance challenging as well.

2.3 Parallel Platforms

Currently, there are three main types of consumer-facing parallel computing platforms - multi-core CPUs, GPUs (Graphics Processing Units) and commodity clusters. We do not discuss supercomputers or other specialized parallel platforms as we are focused on economical approaches to computing.

2.3.1 Multi-core CPUs

Figure 2.3 shows an example block diagram of a multi-core CPU. Multi-core CPUs contain two or more processor cores per die. The processor cores are traditional compute units that execute compute (add, multiply), data movement (load and store) and branch instructions. Each core can also support vector instructions to exploit data-level parallelism in a set of instructions. Each core on a multi-core processor is optimized for single-thread performance and thus multi-core processors perform best on coarse-grained parallel tasks. In multi-core processors, each core typically has a local cache. The cores can also share an L3 cache (as shown in Figure 2.3). The cores on the chip use a shared memory or message-passing protocol to communicate with each other (Figure 2.3 shows a CPU architecture with cores using one shared DRAM). Some examples of multi-core CPUs include the AMD Phenom, Intel's i3, i5, and i7 (quad-core) and Intel Xeon (eight-core) processors.

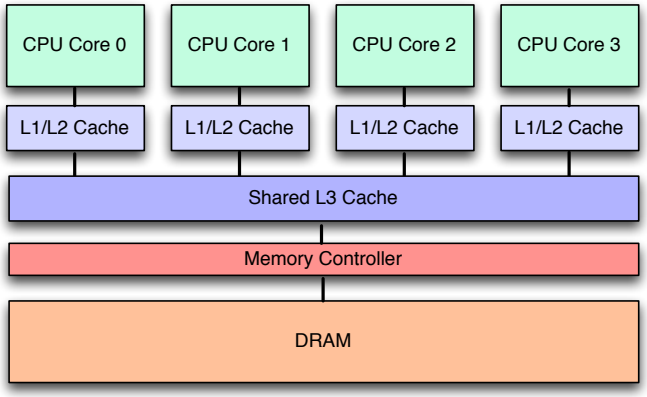

Figure 2.3: Multi-core CPU block diagram.

2.3.2 GPUs

Figure 2.4 shows an example block diagram of a CPU-GPU system. GPUs were originally developed to efficiently execute computer graphics operations such as rendering and texture mapping, but have recently gained traction as general purpose compute platforms (GP-GPUs). GPUs are high-throughput co-processors that are highly efficient at executing arithmetically-intense, data-parallel, streaming computations. GPUs contain a set of cores, each core implementing the SIMD (Single Instruction Multiple Data) execution model where same instructions operate on multiple pieces of data. Each core has a software-programmable local memory (shown as Local Store, "LS", in Figure 2.4). Efficiency comes from amortizing instruction load as well as performing computations on coalesced data. GPUs can be manufactured as a stand-alone piece of hardware (as in the NVIDIA Tesla model) or can be integrated with the CPU on the same die (as in the Intel Sandybridge model).

In this work, we focus on the NVIDIA GPU model, where the GPU is manufactured as a separate compute board and is added to the compute pool as a separate device using a PCI connection. As shown in Figure 2.4, in this model the GPU is coupled with the host CPU to create a CPU-GPU co-processor system. The CPU acts as a *host* processor, issuing memory allocation and copy requests from the CPU memory to the GPU memory and calling kernel functions to execute computations on the GPU *device*. The programmer is responsible for explicitly managing data structure allocations on the GPU. To perform computations on the GPU, data structures must first be allocated in the GPU memory and input data must be copied from the CPU to the GPU. After the computation is complete, the result data structures must be transferred back to the CPU memory and the GPU data structures must be explicitly deallocated.

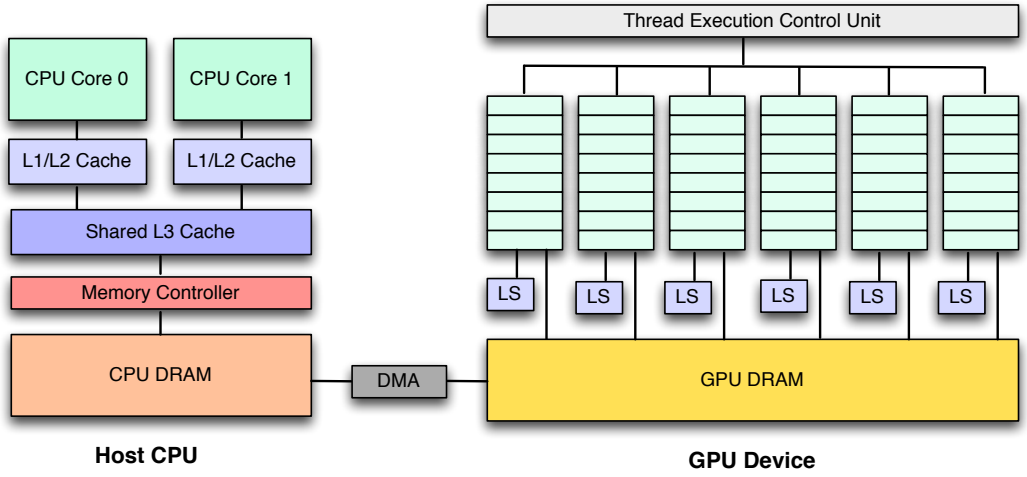

Figure 2.4: CPU-GPU block diagram.

2.3.3 Clusters

Figure 2.5 shows an example block diagram of a computer cluster. A computer cluster consists of a set of inter-connected general-purpose computers (cluster nodes) that can operate together to execute a computation. Clusters were a result of several computing trends - availability of affordable computers, high speed networks and innovation in distributed computing. Early adopters of compute clusters were companies like Google and Yahoo! whose goal was to enable faster cost-effective data processing. Efficient execution on a cluster requires exploiting coarse-grained parallelism. Communication between the nodes of a cluster is expensive since messages have to move across low-bandwidth interconnection networks. Thus, reducing communication and maximizing inter-machine parallelism is essential to obtaining high performance for an application running on a cluster.

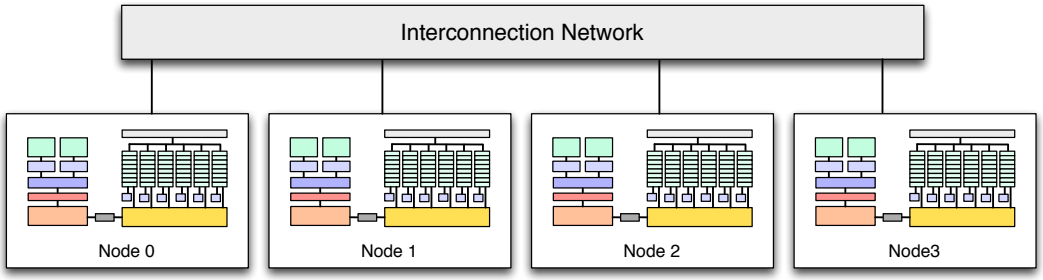

Figure 2.5: Computer cluster block diagram. Four CPU-GPU nodes are connected using an interconnection network.

2.4 Alternative Approaches for Application Development

There are many programmer tools that aim to bridge the implementation gap. Table 2.1 summarizes some representative approaches. We note that there are other categories of alternative approaches, here we focus on the main types that have gotten the most traction. Each row in Table 2.1 shows whether the solution is a language, library or framework, whether its primary focus is efficiency of productivity, whether it targets parallel platforms and whether or not it is application-domain-specific. Subsections below describe each approach in more detail.

Programmer Tool	Efcy/Prod*	Lib/Lng/Fwk**	Parallel?	DS?***
Python	Productivity	Language	No	No
Pthreads, OpenMP, CUDA	Efficiency	Language	Yes	No
BLAS, LAPACK	Efficiency	Library	Yes	No
Marsyas, HTK, OpenCV	Efficiency	Library	No	Yes
Cactus, CHARMS	Efficiency	Framework	No	Yes
Numpy, Gnumpy, Cudamat	Productivity	Library	Yes	No
Theano, Copperhead, MR	Productivity	Framework	Yes	No
Delite, OptiML	Productivity	Framework	Yes	Yes
MapReduce	Productivity	Framework	Yes	No

Table 2.1: Summary of alternative solutions for parallel programming. All identifiers are specific to *application development* process. *Efficiency / Productivity; **Library / Language / Framework; ***DS = Domain Specific?

2.4.1 Efficiency languages

Pthreads (POSIX threads) [67] are one of the oldest mechanisms for parallel programming, stemming from the UNIX operating systems. They employ a low-level programming model that is efficient and non-domain specific. Pthreads are implemented as a set of C types, functions and constants. Each operating system can implement its own version of Pthreads and expose the API to the programmer. The Pthread model views each thread as a stream of independent computations. Each thread maintains its own stack pointer and registers. Threads are created and are assigned a particular function to execute, they can be forked or joined. Threads share data using read/write locks and barriers making synchronization a very challenging programmer task. All thread behavior is *explicitly* controlled by the programmer, thus, while being one of the most challenging parallel pro-

gramming mechanisms, Pthreads provide the most flexibility to the programmer requiring very little structure.

OpenMP (Open Multi-Processing) [78] is an API that supports shared memory multiprocessor programming in C, C++, and Fortran. OpenMP is a multi-threaded programming model, where a master thread controls a set of worker threads; tasks are divided among the threads in a particular specified way using a *work-sharing* mechanism. Programmers mark the parallel sections with preprocessor directives that tell the compiler which sections of the code can be executed by multiple threads. Threads synchronization is expressed using synchronization clauses (atomic, barrier, critical etc.). Task scheduling among threads can be controlled using scheduling clauses (static, dynamic, custom). The programmer is tasked with identifying the parallel sections in his/her code and marking them accordingly; the OpenMP runtime creates and runs the threads and joins them after execution is done.

Intel Cilk+ language [27] is a set of C/C++ extensions for programming multi-core processors. The programmer exposes parallelism in a C/C++ program by identifying elements that can be executed concurrently and marking it with a Cilk-specific keyword. The programming environment decides exactly how the work is split among threads and how the tasks are scheduled, thus the same program can run on one as well as many processors without code change. Similar to the OpenMP model, the programmer is tasked with identifying parallel sections in the code and marking them with the Cilk keywords, while the runtime handles thread creation and execution.

CUDA (Compute Unified Device Architecture) [75] is a framework for programming NVIDIA GPUs. A CUDA application is organized into sequential *host* code written in C running on the CPU and many parallel *device* kernels running on the GPU. The kernel executes a set of scalar sequential programs across a set of parallel threads on the GPU. The programmer can organize these threads into thread blocks, which are mapped onto the processor cores at runtime. Each core has a small software-managed fast local memory. To run an application on the GPU, data structures must be explicitly transferred from host to device. Task scheduling and load balancing are handled by the device driver automatically.

OpenCL [77] is a framework for writing parallel programs that execute across heterogeneous platforms such as CPUs and GPUs. Similar to CUDA, OpenCL uses C extensions for writing host code and device kernels, as well as APIs that are used to define and control the platforms. The OpenCL model uses task-based and data-based parallelism and thus can be used to program a variety of parallel devices.

2.4.2 Efficiency libraries & frameworks

There are several efficient libraries, such as BLAS [10] and LAPACK [2] for numerical and linear algebra computations that aim to bridge the implementation gap by packaging up complex, efficient implementations of particular computations allowing for reuse and productivity. However, libraries typically implement a specific computation for a specific platform presenting a brittle approach. Compiler optimizations and auto-tuners such as those described in [105] and [60] can address the portability issue by tuning the application code to a particular platform. Libraries and auto-tuners, however, do not provide guidance on how to design an application *as a whole* to allow for most efficient and scalable implementation. Efficiency application frameworks, like Cactus [45] and CHARMS [47], on the other hand provide a flexible environment for application development, while still enabling high performance. Programmers can develop their applications using the framework that guides them in the composition of various computations in a pre-defined way. Both of these approaches allow for increased productivity and efficiency in application development, but are still low-level solutions.

2.4.3 Domain-specific efficiency libraries & frameworks

There is an extensive list of domain-specific libraries and frameworks for audio and visual media analysis, here we name a few important ones. OpenCV [46] is a library for developing computer vision applications, that includes Python, C/C++ and Java interfaces and targets CPU platforms. The Hidden Markov Model Toolkit (HTK) [49] is a portable toolkit for building and manipulating hidden Markov models as well as performing other processing on audio, used for speech recognition. Marsyas [104] is a popular music information retrieval (MIR) software framework for rapid prototyping, design and experimentation with audio analysis and synthesis with specific emphasis on processing music signals in audio format. CLAM [1] is a C++ framework, for development and research in audio and music signal processing applications. These frameworks provide an extensive API for processing audio and image data, however most of them are presented in a low-level language such as C++ and do not target parallel platforms. While some frameworks, such as OpenCV have some modules that are ported to parallel backends, parallelism is not the primary goal of such frameworks.

2.4.4 Productivity languages

Many application domain experts use productivity languages such as MATLAB and Python to prototype and develop their applications. Such productivity languages provide high-level abstractions for many computations (for example linear algebra or machine learning computations) as well as provide light-weight scripting

"glue code" capabilities such file I/O, data plotting and data flow logic. Prototyping applications in such high-level languages can be done in a couple hundred lines of code, allowing application writers to easily experiment with application algorithms and data. However, these languages are usually impeded by performance limitations. Unless there are libraries (such as BLAS) written in low-level languages and linked to Python or MATLAB, applications written in pure MATLAB or Python show very poor performance [15]. Thus, even though application logic implementation can be done quite rapidly in such languages, experiment turn-around is typically slow due to performance limitations.

2.4.5 Productivity libraries

There is a set of Python libraries that call efficient multi-core CPU and GPU implementations of particular computations. For example Numpy [5] and Gnumpy [102] contain APIs for numeric processing in Python that execute the computation on CPUs and GPUs respectively. Numpy runs on top of BLAS and Gnumpy runs on top of Cudamat [72], both of which contain basic data parallel and linear algebra computations such as vector and matrix addition and multiplication. These libraries provide a convenient, non-domain-specific way of performing numeric and linear algebra computations in Python that run on parallel hardware. They provide a clean interface to common numeric operations and are easy to use for domain experts. However, their APIs do not provide flexibility that is sometimes required for the application implementation.

2.4.6 Productivity frameworks

There are several frameworks that aim to bridge the implementation gap by coupling productivity of Python with the efficiency of C-level implementations. Some examples include Theano [7], a CPU/GPU math-compiler for Python, the Copperhead language [14] which provides a set of data-parallel abstractions expressed as a restricted subset of Python. The Delite framework [18] is a Domain Specific Language (DSL) creation framework and runtime for stencil operations and OptiML [100] is a framework for executing efficient machine learning algorithms from productive Scala code developed by Stanford University. Both Theano and Copperhead provide automatic mapping of data-parallel computations onto GPUs allowing for productivity and performance and are non-domain-specific. Similar approaches are used in domain-specific frameworks. Delite is a DSL and compiler for stencil operations with code generation capabilities for Scala, C++ and CUDA and OptiML is a DSL for machine learning applications written in Scala that extracts task-parallelism from high-level machine learning algorithm descriptions.

2.4.7 MapReduce

Finally, there is a set of frameworks that aim to enable productive programming of computer clusters. They use the MapReduce paradigm originally developed by Dean et al. [32]. These frameworks facilitate inter-machine parallelism by providing building blocks with which the application programmer can construct parallel dataflows using *map* and *reduce* functions on key-value pair objects. MapReduce frameworks are designed to be productive - all of the machine communication and parallelism details are hidden from the programmer.

The MapReduce paradigm is supported by a number of languages and frameworks, for example Hadoop [107] and MapReduce [33]. The DryadLINQ [112] programming model provides a declarative, sequential, single machine programming abstraction and the computations are expressed in a high-level language similar to SQL. There are also several domain specific languages (DSLs) implemented on MapReduce including Sawzall [82], Cascading [51] and Papyrus [50].

MapReduce frameworks have two main drawbacks. First, they are not suitable for all scientific workloads since they are designed for data-intensive workloads. Second, they still require a modest understanding of parallel programming despite the fact that they are designed for productivity and hide many of the details of distributed computing.

2.4.8 Other approaches

We've covered some of the main representative approaches to developing applications. There are a lot of other alternative approaches that we have not covered in this section, including message-passing frameworks (such as MPI [48], Charm++ [56]), other threading libraries (such as TBB [90]) and cluster programming frameworks (Spark [113], Pig [76]).

2.5 Comparing Alternative Approaches

Figure 2.6 shows the relative efficiency and productivity of the alternative approaches presented in the previous section. The efficiency scale denotes the *efficiency of applications* that is possible to achieve when they are written using these approaches. The productivity scale denotes the *productivity of application developers* when developing the applications using the specific approach. It is important to note that the placement of each approach does not capture the exact measured efficiency and productivity, but is rather meant to show relative placement of the approaches on the global spectrum.

Figure 2.6 illustrates, that, while efficiency languages such as Pthreads and CUDA provide the highest efficiency for applications, they are very low on the

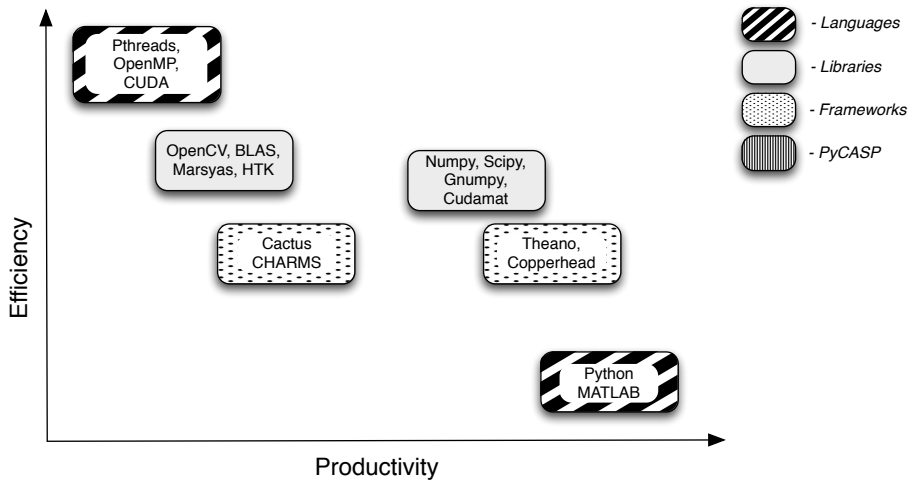

Figure 2.6: Comparing efficiency and productivity of alternative approaches.

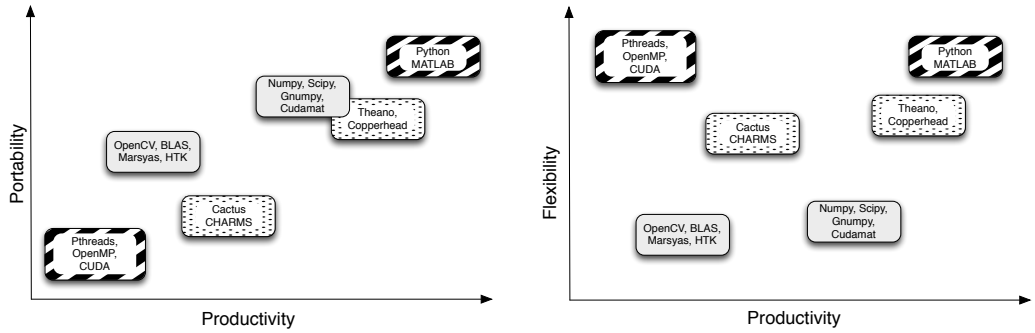

Figure 2.7: Comparing portability and productivity (left) and flexibility and productivity (right) of alternative approaches.

productivity scale. Using these languages, the application programmers are required to write low-level code and tune it to particular computations and platforms. On the opposite end, Python and MATLAB are high-level languages that allow application programmers to stay highly productive, but do not yield efficient code. Efficiency libraries (such as OpenCV, BLAS and HTK) and frameworks (such as Cactus and CHARMS) provide efficient implementations of common computations, and give more productivity than the efficiency languages. They are, however, not as productive as Python-based productivity libraries such as Numpy, Gnumpy and Cudamat and productivity frameworks such as Theano and Copperhead. Those productivity libraries and frameworks provide efficient implementations that are wrapped in Python code, and thus are also more efficient.

Figure 2.7 places the alternative approaches on the portability-productivity spectrum (left) and flexibility-productivity spectrum (right). As before, the pro-

ductivity scale denotes the *productivity of application developers* when developing the applications. The portability scale denotes *how many different platforms* applications that are written using these approaches can run on without application code change. The flexibility scale denotes the *application developer flexibility* in specifying application functionality when using these approaches. While the positioning of each approach doesn't change on the horizontal (productivity) axis, they show quite different placement on the vertical (productivity and flexibility) axes.

Applications written in Pthreads or CUDA tend to be non-portable, i.e. they are usually tuned to one particular platform and in order to enable them run on another type of platform, application code has to be rewritten from scratch. On the other end, applications written in Python or MATLAB can run on any platform. Efficiency and productivity libraries usually can support multiple platforms in the backend (different types of multi-core CPUs and GPUs for example) making applications more portable to a variety of hardware platforms.

The flexibility of each approach also differs. Efficiency and productivity languages provide the highest degree of flexibility - programmers can express arbitrary computations using these approaches. Flexibility becomes more restricted when we move to frameworks since they only allow for specific types of computations, but do give programmers flexibility in specifying exact details of code functionality. Libraries provide the least amount of flexibility as they implement very specific computations and usually do not allow for any modifications of the underlying computation.

2.6 Our Approach

In the previous section, we discussed the major alternative approaches to application development. While some prior approaches present reasonable solutions for bridging the gap, they have several deficiencies. We summarize them below:

- Efficiency languages allow for very efficient application implementations but do not allow for programmer productivity or portability of applications.
- Productivity languages allow for very high programmer productivity and allow for application portability but yield inefficient applications.
- Efficiency libraries and frameworks are efficient but do not allow for significant programmer productivity, are fragile and non-extensible, and allow for modest amount of portability of applications.
- Productivity libraries are efficient and allow for more programmer productivity than efficiency libraries but are still fragile and non-extensible.

- Productivity frameworks are efficient, productive and reasonably flexible and portable, yet they typically are non-domain specific and thus cannot take advantage of high-level domain knowledge for further application optimization.
- Prior efficiency and productivity approaches do not allow for efficient composition of computational components in applications.

In this thesis, we aim to find a software solution that:

1. Bridges the implementation gap between application developers and efficiency programmers to enable application developers to utilize parallel hardware,
2. Achieves the goals of bringing productivity and flexibility to the application development process and efficiency and portability to the resulting applications,
3. Allows for application-domain-specific optimizations, and
4. Allows for composition of computations in applications.

*We propose the **PyCASP (Python-based Content Analysis using Specialization) framework** as the software solution to bridge the implementation gap and achieve application programmer productivity and flexibility as well application efficiency and portability to a variety of parallel hardware.*

As mentioned in Chapter 1, bridging the implementation gap is a difficult task. Application developers and efficiency programmers prefer to work in their comfort zones, yet each type of programmer lacks the required understanding to build full-functioning applications that scale to parallel hardware. In addition, as illustrated by the comparison of alternative approaches, productivity, efficiency, portability and flexibility are opposing forces in an application development process. Writing efficient code requires low-level implementations and careful tuning which impedes programmer productivity. Developing portable applications requires rewriting each application for each platform, also significantly impeding productivity. Allowing flexibility usually takes away from efficiency since we cannot provide an optimized implementation for each use case in an application. Composition of computations is also a notoriously hard problem: there are too many computational workloads in software applications, each varies greatly in the type of computation and data representation it uses. Composing computations requires high-level knowledge of the computation and data format and low-level optimizations, making it difficult to automate. Thus, creating a single software

environment that achieves all of these aspects in addition to bridging the implementation gap is a difficult undertaking.

To find the right solution, we have extensively studied prior work on the subject. We aim to bridge the implementation gap by carefully defining the scope and using code specialization to achieve the goals of productivity, efficiency, flexibility and portability in PyCASP. In addition, we hypothesize that making our software framework *application-domain-specific* will enable productivity, as well as allow for application-domain-specific optimizations and efficient composition of computations in applications. For this purpose, we choose to use the audio content analysis application domain as the target application area, but our aim is to design a software solution that can be applied to any application domain.

In order to bridge the implementation gap between application developers and efficiency programmers, we aim to design and develop a software solution that provides an infrastructure for both types of programmers. Our goal is to enable application developers to utilize parallel hardware, thus, we set the target audience for our software solution to be application developers. Specifically, since we aim to make our software solution application-specific, *we set our target audience to be the audio content analysis researchers and domain experts.* Audio content analysis researchers and application developers focus on analyzing and obtaining high-accuracy information from audio sources such as speech and music. These programmers typically prefer to use high-level languages such as MATLAB or Python and focus on the algorithms used in their applications and not low-level implementation details. On the other end, we aim to enable efficiency programmers to develop efficient parallel implementations of particular computations that can then be used by the application developers. Our software solution needs to provide efficiency and portability to applications developed by the domain experts. In addition, our software solution needs to provide reasonable amount of flexibility to allow for implementation of a variety of applications in the domain. Thus, we need to focus carefully on the design, the scope and the infrastructure of our solution to bridge the implementation gap between the two programmer types and achieve our goals.

2.6.1 Patterns & Our Pattern Language

In prior work, we have seen that using a *pattern-oriented approach* for the design of software frameworks and libraries can provide modularity and productivity to software infrastructures [25, 99]. Patterns give a common language to the domain experts and efficiency programmers to communicate with each other and provide a set of abstractions that help define the scope of software frameworks. Specifically, a pattern-oriented approach based on design patterns developed by Keutzer and Mattson in the OPL (Our Pattern Language) project [62] has shown a lot of potential in designing comprehensive software environments. Patterns

provide a generalizable solution to a class of recurring problems that occurs in the design of software. OPL provides a pattern-based language to describe all software applications using patterns. With OPL patterns, we can describe both the structure of the applications as well as the computation of each component of the application. This provides a modular, comprehensive way of structuring and communicating the organization of various applications.

Pattern-oriented design has a strong foundation in the field of software engineering, from the design patterns in object-oriented programming [42] and architectural styles [97], to patterns in parallel programming [70] and performance modeling and analysis of applications [98]. In our work, we aim for patterns to provide us with guidance on the framework's scope and performance implications as well as define a common vocabulary to the software environment developers and domain researchers who use it.

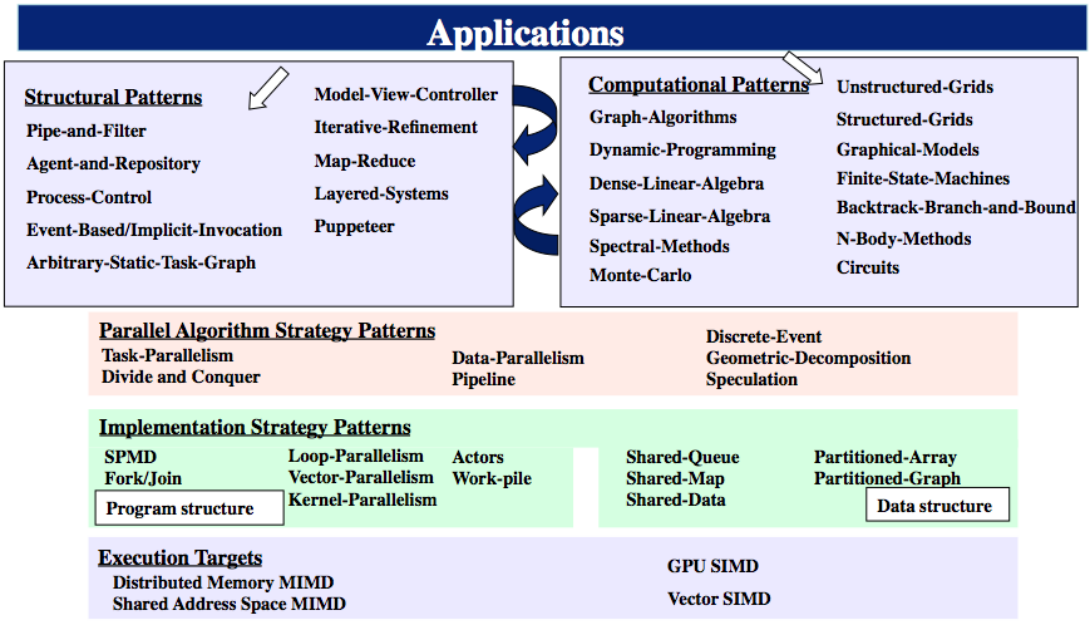

Figure 2.8: Patterns in Our Pattern Language (OPL)

Figure 2.8 shows the patterns that make up OPL. Applications (at the top of the Figure) composed of application patterns (not shown) are decomposed into *computational* and *structural* patterns. Computational patterns describe the workload of the computation while the structural patterns describe the control-flow and composition of computations in the application. The parallel algorithm strategy patterns describe the parallelization strategies available for extracting parallelism out of computations. The implementation strategy and parallel execution patterns describe how the parallel computation is further decomposed into

parallel tasks and executed on parallel hardware.

Together, computational and structural patterns describe the *software architecture* of applications. Software architecture describes the organization, i.e. the high-level computational and control-flow of an application. A software architect designs applications by constructing software architectures. Software architectures can be captured as boxes and arrows and drawn on a white board. They give information about the structure and internal control- and data-flow of the application. They provide modularity as the control flow between computations is clearly described by the boxes and arrows. Software architectures are essential for a systematic, modular application design and communication between application designers and developers.

2.6.2 Selected Embedded JIT Specialization (SEJITS)

In analyzing previous work, we have seen progress in achieving productivity, efficiency and flexibility with *productivity frameworks* such as Copperhead and Theano [14, 7]. These frameworks use separation of concerns to enable application developers and efficiency programmers to stay in their comfort domains and provide a software infrastructure to bridge the gap between the two types of programers. Specifically, these frameworks use code specialization and just-in-time (JIT) compilation techniques to bridge the implementation gap and achieve productivity, efficiency and flexibility. Specifically, a technique called Selected Embedded JIT Specialization (SEJITS) has shown a lot of potential in bringing efficiency to applications written in a high-level (productive) language [17, 57].

SEJITS-based frameworks contain a set of *specializers* that automatically parallelize specific computations on different parallel hardware. Specializers are mini-compilers for domain specific embedded languages (DSELs) glued together by Python code. SEJITS has the capability of rendering templated low-level code as well as lowering Python code using AST (abstract syntax tree) transformations to low-level code such as C or CUDA to enable specializer customization (see [17]).

To allow for productivity and efficiency, SEJITS focuses on the separation of concerns: the application Python programmer focuses on innovating the application and the efficiency programmer focuses on developing specializers for specific computations. The specializers are then reused across applications by domain researchers. The challenge of verification is eased due to programs being shorter and easier to understand. The separation of concerns also allows for SEJIT specializers to target multiple platforms in the back-end of the framework without changing the application code, allowing for portability.

When working with SEJITS-based frameworks, scientists express their applications entirely in Python using Python libraries and tools. They import the framework's specializers as Python objects into their code. When calling the specialized functions from the application, SEJITS *automatically* generates low-

level parallel code for a given back-end platform and problem size, compiles and executes it on the parallel platform. From the Python application programmer's view, this experience is like calling a pure Python library, except that performance is potentially several orders of magnitude faster.

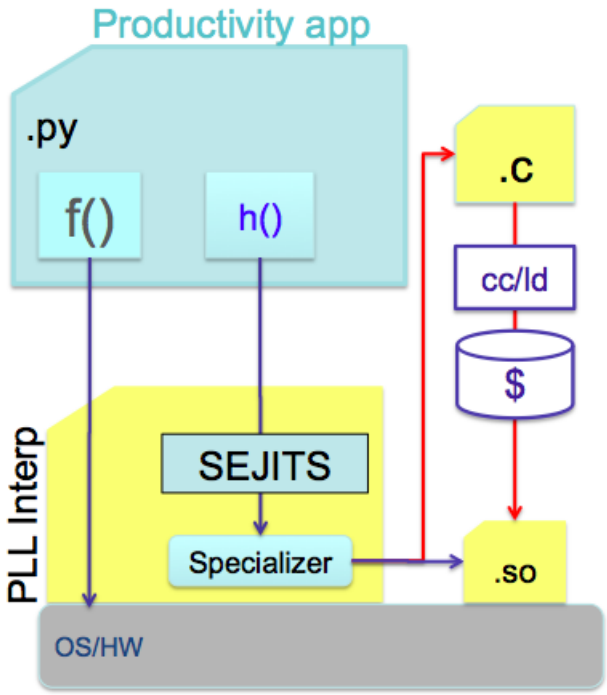

Figure 2.9: SEJITS logical flow

Figure 2.9 shows the logical flow of using SEJITS in an application. At the top, there is the productivity application written in Python. The application calls several functions such as $f()$ and $h()$. While the typical Python function $f()$ gets directly interpreted by the Python interpreter and executed on the underlying hardware, the call to the *specialized* function $h()$ gets intercepted by SEJITS. When SEJITS intercepts the call to $h()$ from the application, it calls the specializer code to generate an efficient C version of $h()$ (right side of Figure 2.9). The C code gets compiled, cached and linked back to the Python program and executed on the underlying hardware. This process is transparent to the application writer and the entire logic of specialization is captured in the specializer (written by the parallel programmer/specializer developer).

A specific framework for developing specializers called Asp (A SEJITS for Python) was proposed and developed by Kamil [57]. Asp contains facilities to automate the process of determining the best variant to use, emit source code corresponding to that variant, compile and call the optimized code, and pass the

results of the computation back to the Python interpreter. Asp has shown a wide applicability to a variety of application domains [58].

2.7 Summary

The shift to parallel programming presents a set of challenges to the application developers and introduces an implementation gap between efficiency programmers and application developers. There is a variety of approaches for bridging the implementation gap ranging from efficiency languages, libraries and frameworks to productivity languages, libraries and frameworks. Each approach has its advantages and drawbacks, and most importantly, neither approach achieves application programmer productivity and flexibility as well as application efficiency and portability. In this thesis, we propose PyCASP - Python-based Content Analysis using Specialization framework as a software solution that achieves those goals and bridges the implementation gap. We will use pattern-oriented design methodology to define the scope of PyCASP and use the Selected Embedded JIT Specialization (SEJITS) methodology as the implementation infrastructure for our software environment. Finally, we hope that restricting the scope of our software environment to one application domain will can enable composition of computations as well as domain-specific optimizations to yield more efficient applications.

Chapter 3

Audio Content Analysis

We wish to find a compelling application domain which will benefit from a programming approach that provides productivity and efficiency when programming a variety of high-throughput parallel processors. The audio content analysis application domain is a great candidate for our approach. Audio content applications implement algorithms for extracting information from audio data such as speech, music and video soundtracks. With hundreds of videos and audio files being uploaded to the web every minute [65], there is a high demand for scalable solutions to large-scale audio content analysis. For example, appealing audio content analysis applications include automatic video and audio transcription and search, recommendation of new musical content, and tools for geo-location and privacy analyses of human speech. Accurate and robust applications require training on hundreds of thousands of learning examples requiring hours or days of processing time. In addition, such applications require real-time processing when integrated into interactive environments such as home entertainment systems and mobile applications, which require fast, low-latency, portable solutions to audio processing. With such intensive application demands this domain presents a lucrative target for a software solution that allows for automatic parallelization and scaling on a variety of parallel hardware.

3.1 Audio Content Analysis Applications

Audio content analysis applications use a large variety of signal processing and machine learning techniques to extract information from audio sources. Figure 3.1 shows a high-level diagram of audio content analysis applications. We start with raw audio from speech or music. Typically, the first step in analyzing the raw audio data is to extract features from raw audio signals using signal processing algorithms. For example, the Mel Frequency Cepstral Coefficients (MFCC) features are commonly used to extract and represent short-term spectral informa-

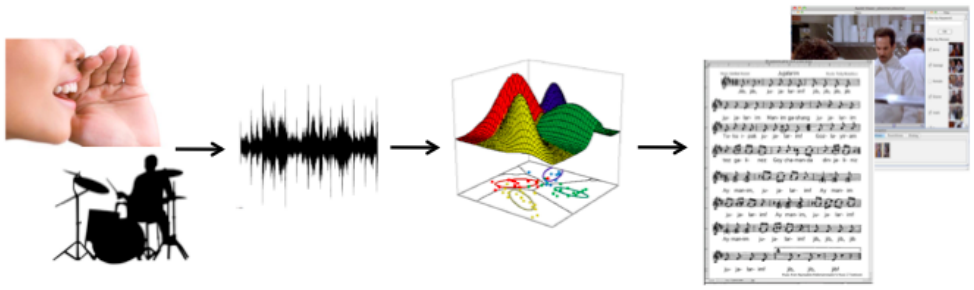

Figure 3.1: High-level overview of audio content analysis applications.

tion from audio sources (commonly employed in speech processing applications). After feature extraction, a variety of machine learning algorithms are used to construct models that best explain the specific aspects of the data relevant to the application. For example, in speech recognition applications, Gaussian Mixture Models (GMMs) or Neural Networks (NNs) are used to model acoustic properties of human speech. Several machine learning blocks can be composed together into more complex applications. For example Hidden Markov Models (HMMs) are composed with GMMs or NNs in speech recognition inference algorithms [111].

There are many appealing audio content analysis applications, for example:

- Automatic speech recognition and transcription
- Speaker modeling and verification
- Meeting diarization
- Music recommendation
- Beat tracking in a music performance
- Music genre identification
- Music fingerprinting / identification
- Video soundtrack transcription and event detection
- Audio search
- Video geo-location and privacy analysis

As mentioned in Chapter 1, we use four audio content analysis applications as working examples to drive our design process and illustrate our approach throughout this work. We summarize the applications below.

1. **Speaker verification** application identifies if a piece of input audio belongs to a particular target speaker.

2. **Speaker diarization** application segments an audio recording into speaker-homogeneous regions, addressing the question "who spoke when" without any prior knowledge of the recording or speakers.

3. **Music recommendation** application finds a set of songs that are most similar to a given song or artist.

4. **Video event detection** system identifies a set of audio events (for example "feeding an animal" or "birthday party") in a set of videos by analyzing the audio soundtracks.

We choose these four applications for several reasons:

- These applications present diverse workloads. Speaker verification works on very small inputs of human speech (2-5 seconds) while the music recommendation system uses a large database of one million songs to find similar songs based on their audio properties.

- They analyze different types of audio. Speaker verification and diarization analyze human speech. Music recommendation system analyzes a variety of songs (pop, rock, jazz, hip hop) and video event detection analyzes audio from consumer-produced videos.

- They take a step away from traditional speech recognition workloads. Speech recognition is a sub-field of audio content analysis domain which has been extensively studied since the 1940's. While we still address speech in our sample applications, our goal is to design a framework that can support *arbitrary audio analysis* and thus, focusing on speech recognition would be too restrictive.

In this thesis, we will use these applications to support and inform our decisions about the design and implementation of our software solution. We discuss the applications and their underlying algorithms in more details in the next set of sections.

3.1.1 Speaker verification

A speaker verification system automatically identifies if a piece of input audio belongs to a particular target speaker. Such a system can be employed in a biometric authorization application, i.e. when a user wants to unlock a device for example his/her phone. The speaker verification system requests an audio

password and determines whether the recorded audio belongs to the owner of the device or an intruder.

Algorithm

Our speaker verification uses Gaussian Mixture Models (GMMs) to model acoustic properties of the speakers and Support Vector Machines (SVMs) to classify the speakers into categories. This algorithm was originally developed and described by Campbell et al. in [12], we summarize it below. The algorithm consists of two phases: training and classification. The training phase is shown in Figure 3.2 and contains the following steps:

1. **Train a Universal Background Model (UBM)** on audio samples from a variety of speakers. A UBM is modeled by a GMM that "averages" the audio features from a random sample of the training data and places the average in a multi-dimensional space.

2. **Adapt the UBM to target speaker audio**. For each audio file from our target speaker, adapt the UBM to our target speaker to compute a *supervector*. The GMM initialized with the UBM parameters. The adaptation loop consists of a few (typically one to three) GMM training iterations using the audio from the target speaker data. After UBM adaptation, we concatenate the means of the adapted GMMs into a supervector to be used as our speaker classification features for the SVM. UBM adaptation approach for speaker verification was originally described in [92].

3. **Adapt the UBM to intruder examples**. For each audio file from a set of intruder examples, adapt the UBM to intruder speaker examples to compute supervectors of example non-target-speaker audio.

4. **Create a label set**. Create a set of labels for the supervectors from our target speaker (with value 1) and from the intruder set (with value -1), and train a SVM on the set of positive and negative examples for our verification system.

The classification phase of the speaker verification is shown in Figure 3.3. When a new audio file comes in, we adapt the UBM and use the SVM to determine whether the audio came from our target speaker or an intruder.

3.1.2 Speaker diarization

Speaker diarization application segments an audio recording into speaker-homogeneous regions, addressing the question "who spoke when" without any prior

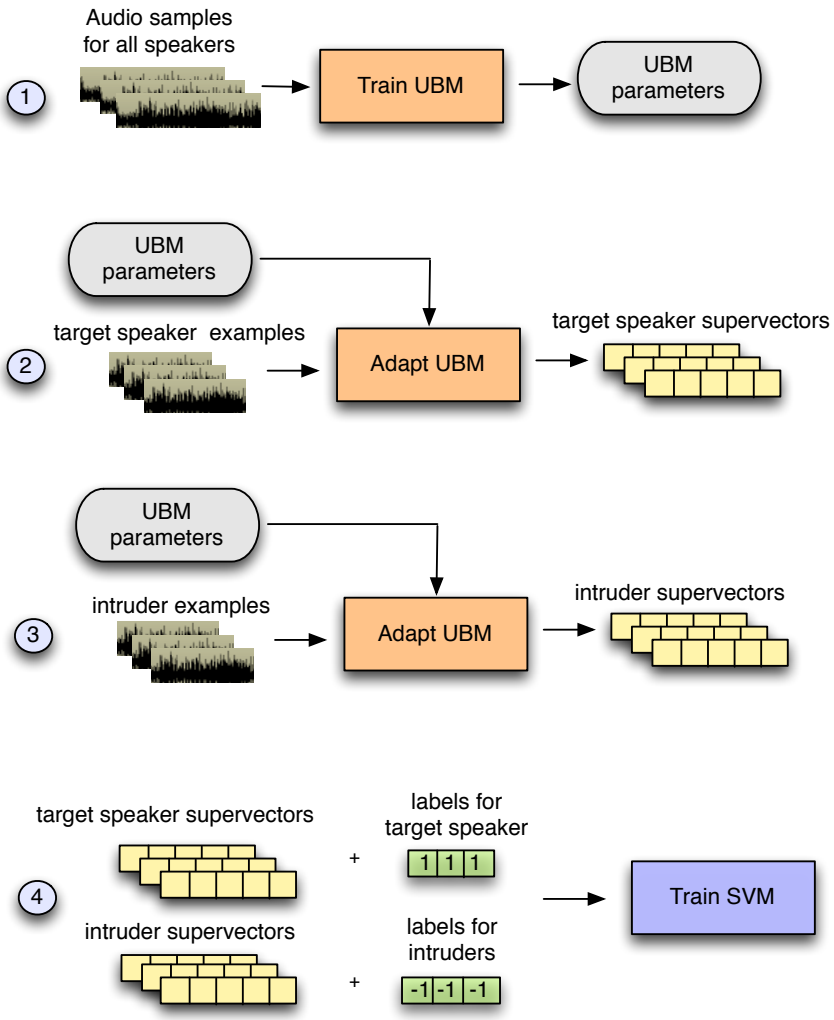

Figure 3.2: Training phase of the speaker verification system.

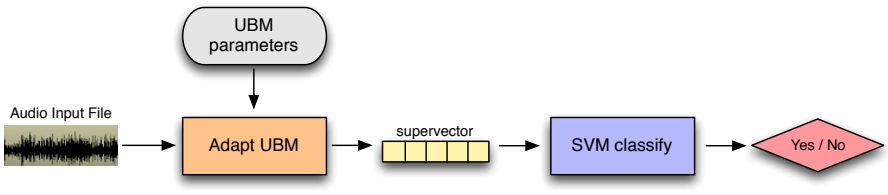

Figure 3.3: Classification phase of the speaker verification system.

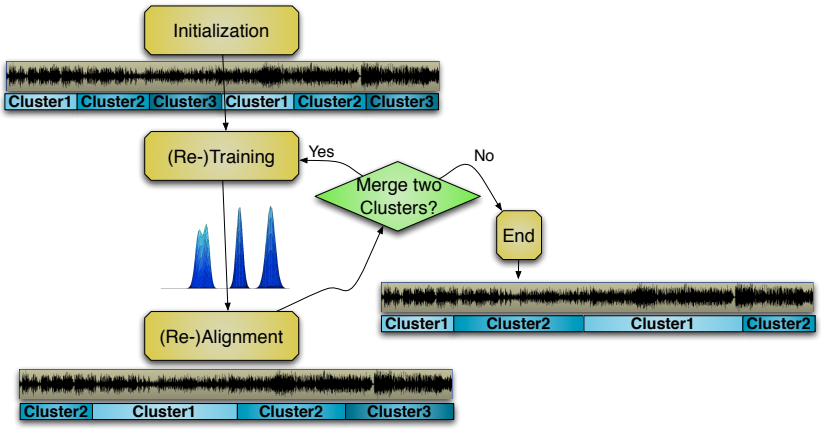

Figure 3.4: llustration of the segmentation and clustering algorithm used for speaker diarization.

knowledge of the recording or the number of speakers. One popular diarization method uses agglomerative hierarchical clustering with the Bayesian Information Criterion (BIC) and GMMs trained with frame-based cepstral features [4]. This method combines the speech segmentation and segment clustering tasks into a single stage using agglomerative hierarchical clustering, a process by which many simple candidate models are iteratively merged into more complex, accurate models. Figure 3.4 shows the general organization of such a diarization system.

Algorithm

The diarization is based on 19-dimensional, Gaussianized, Mel-Frequency Cepstral Coefficients (MFCCs). We use a frame period of 10 ms with an analysis window of 30 ms in the feature extraction, as well as the speech/non-speech segmentation used in [108]. In the segmentation and clustering stage of speaker diarization, an initial segmentation is generated by uniformly partitioning the audio track into K segments of the same length. K is chosen to be much larger than the assumed number of speakers in the audio track. For meeting recordings of about 30 minute length, previous work [53] experimentally determined $K = 16$ to be a good value.

The procedure for diarization is shown in Figure 3.4 and takes the following steps (more details can be found in [108]):

1. **Initialize**. Train a set of GMMs, one per initial segment.
2. **Re-segment**. Re-segment the audio track using majority vote over the GMMs' likelihoods.
3. **Re-train**. Retrain the GMMs on the new segmentation.

4. **Agglomerate**. Select the most similar GMMs and merge them. At each iteration, the algorithm checks all possible pairs of GMMs, looking to obtain an improvement in BIC scores by merging the pair and re-training it on the pair's combined audio segments. We use the unscented-transform-based KL-divergence optimization [**fastmatch**] to not have to compare all pairs of GMM. The GMM clusters of the pair with the largest improvement in BIC scores are permanently merged. The algorithm then repeats from the re-segmentation step until there are no remaining pairs whose merging would lead to an improved BIC score.

The result of the algorithm consists of a segmentation of the audio track with n segment subsets and with one GMM for each subset, where n is assumed to be the number of speakers.

3.1.3 Music recommendation

Music recommendation is one of the most challenging applications in Music Information Retrieval (MIR). The goal of a music recommendation system is to recommend a set of songs that are most similar to a given song or artist. Most current recommendation systems such as Pandora (www.pandora.com) and Last.fm (www.last.fm) use collaborative filtering [101] or manually label songs with tags. These approaches require tedious, manual labeling of high-level audio features thereby severely limiting scalability of the system. Instead, we can use the *audio content* of the recordings to find similar songs. This is the approach we take in our music recommendation system.

Algorithm

Our music recommendation system uses the UBM-GMM supervector approach and Locality Sensitive Hashing (LSH) described in [20] and [71] respectively to retrieve a set of most similar songs given a query song title or artist name. Figure 3.5 (top and bottom) shows the offline and online phases of the recommendation system. In the offline phase, the system prepares the data for online querying. In the online phase, the system uses the data structures pre-built during the offline phase to retrieve a list of most similar songs to a given query. We use a Python interface to SQLite database to store and retrieve song meta-data and features (shown as "SqliteDB" in the figures).

We use the Million Song Dataset [8] assembled by Columbia University using data provided by Echo Nest. The Million Song Dataset contains audio features (onsets, timbre) as well as metadata (title, artist name, duration etc.) for 1 million popular songs. Our system uses the timbre features to recommend a set of similar songs out of the 1 million songs based on the features of the songs that

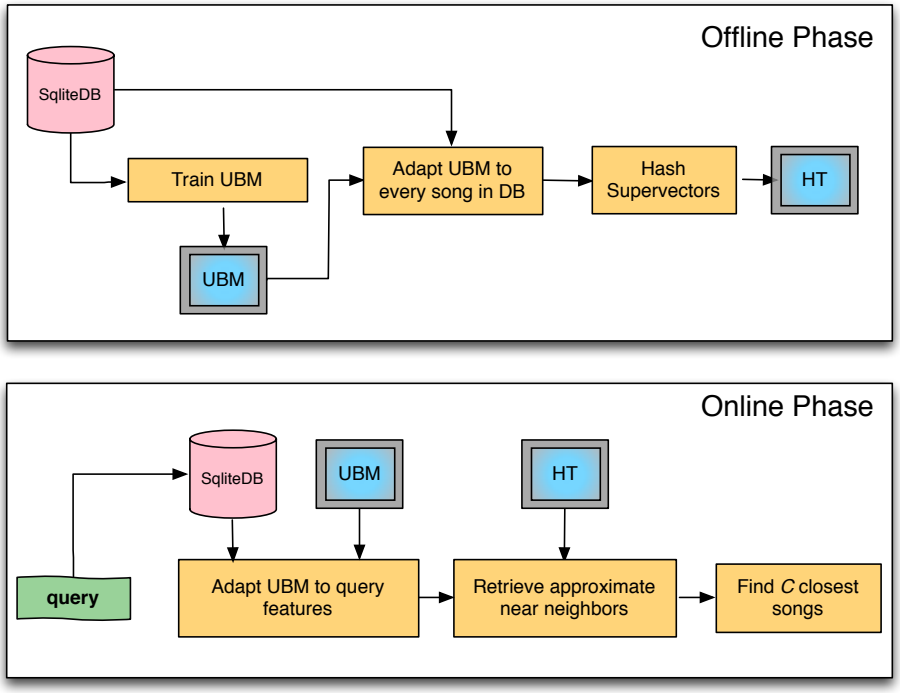

Figure 3.5: Top: Offline data preparation phase of the content-based music recommendation system. Bottom: Online song recommendation phase of the content-based music recommendation system.

matched the query.

Offline Data Preparation Phase

In the offline phase (shown at the top of Figure 3.5), we prepare data to be used during recommendation in the following way:

1. **Train UBM**. We first train the UBM on a random subset of timbre features of all songs in the Million Song Dataset to obtain UBM parameters (weights, means and covariance of the GMM). The UBM is a GMM of 64 components.

2. **Adapt the UBM to all songs**. After we compute the UBM parameters from the previous step, we use UBM MAP adaptation (described in [20]) to compute supervectors (mean vectors of the trained GMM) for each of the songs in the Million Song Dataset and normalize them using the MCS-norm [20].

3. **Hash song supervectors**. After computing and normalizing the supervectors, we use a Locality Sensitive Hashing (LSH) technique to hash the supervectors to a hash table. LSH is a general technique for computing

approximate nearest-neighbors in a high-dimensional space originally described by Andoni and Indyk [3]. We base our implementation on the one described by Casey in [71]. We use $K = 8$ projections and $L = 11$ hash tables with the quantization bin size of $w = 1.291$ to retrieve (on average) 90% of songs within $R = 0.3$ radius of the query point and reject 89% of songs farther than $c * R = 0.72$ radius from the query point.

Online Recommendation Phase

In the online recommendation phase of our music recommendation system (shown at the bottom of Figure 3.5), we use the data structures constructed during the offline phase and compute the set of songs similar to a given query.

1. **Get the query from the user**. Our system can recommend songs based on song title, an artist name or a list of song titles and/or artist names. After receiving a query, we retrieve the features of the songs that match the query (i.e. songs with the given artist name) from the SQLite database and concatenate the set of timbre features.

2. **Adapt the UBM on the query**. After obtaining the timbre features for all the songs that matched the user query, we then adapt the UBM on the query song features to obtain the query timbre supervector.

3. **Get approximate nearest-neighbors for the query supervector**. We use our LSH hash functions to retrieve the set of nearest neighbors to the query supervectors.

4. **Compute the closest C songs**. We use p-norm distance (described in [20]) to compute the C closest songs out of the nearest neighbors returned by our LSH table.

The resulting song IDs are sent back to the query request service and the audio is retrieved and played back to the user.

3.1.4 Video event detection

A video event detection system aims to identify a set of audio events in a set of videos [36]. Most state-of-the-art approaches rely on manual definition of predefined sound concepts such as "engine sounds" or "outdoor/indoor sounds". These approaches require manual event definitions and are very domain specific. The goal of our video event detection application is to detect events in consumer-produced videos found on the web. The definition of "event" goes beyond simple object recognition to more abstract concepts such as "feeding an animal", "wedding ceremony" or "attempting a board trick". This system is designed for large scale retrieval and tested on the TRECVid MED 2011 development data set. It

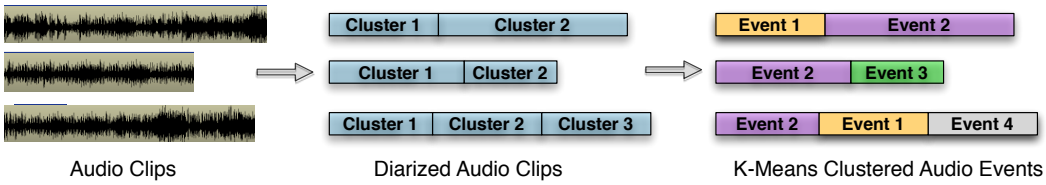

Figure 3.6: Overview of the video event detection system. Each video soundtrack file gets diarized, i.e. clustered based on the audio event content. Then all clusters across all audio files get clustered into a global set of audio events using k-means clustering.

performs learning based on arbitrary low level audio features and can be used to detect high level concepts like those given in the dataset.

Algorithm

The event detection system is generalized from speaker diarization for indexing audio contents. We use the detection system to automatically identify low-level sound concepts similar to annotator defined concepts and then use these concepts for indexing. The applicability of speaker diarization to video indexing and event detection is most similar to the approaches described in [68] and [21].

We use GMMs to represent the audio concepts. In order to match low level audio concepts across training videos and also to classify low level feature models found in testing videos, the system reduces the per-event GMM to a single vector that consists of the sums of the weighted means and the sums of the weighted variances of each Gaussian (we call this vector a simplified supervector). A K-means method is then used to cluster the simplified supervectors that were generated from all of the low level acoustic concepts, resulting in clusters that represent audio event abstractions. These can then be mapped back to the concepts in each video by calculating the distance between the video's event models and the abstract simplified supervectors.

Figure 3.6 shows the overall structure of the video event detection system. Starting with the features of audio files, each file is diarized using the speaker diarization algorithm. K-means clustering is then performed on the simplified supervectors. This results in each audio file being segmented into events, and all the same events clustered together. Thus, we obtain a global view of the different audio events present in a video collection and can then perform indexing and search based on the events.

3.2 Parallelizing Audio Analysis Applications

3.2.1 Paralleizing speech processing applications

There is much prior work on accelerating speech processing applications using parallelism. In the 1993 Ravishankar et al. [89] implemented parallel speech recognition engines on shared memory multiprocessors (SMP) by statically partitioning data and tasks among threads. Their implementation achieved up to 3.85× speedup using 5 threads. In 1999 Phillips et al. [81] parallelized a speech recognition engine on an SMP and and obtained factors of 3-6 speedup on 4-12 processors respectively. Then in 2006, Ishikawa et al. [54] implemented a Large Vocabulary Continuous Speech Recognition (LVCSR) system on cellphone-oriented platforms achieving real-time speech processing performance. You et al. [110] achieved about 2× faster than real-time performance on a 20,000 word speech recognition task by parallelizing the algorithm on multi-core processors using OpenMP [78]. All of these efforts focused on parallelizing the acoustic observation probability computation phase of the inference engine since it took up majority of the execution time, leaving the network traversal phase sequential. Only recent work (including our own) parallelized the entire engine on parallel processors.

Most recently, Graphics Processing Units (GPUs) emerged as programmable highly parallel processors and presented an interesting new target for speech recognition applications. Chong et al. [24] first parallelized the entire recognition engine on a GPU obtaining 10.5× speedup compared to a sequential version of the algorithm. GPUs have also been used to accelerate acoustic model computations in speech recognition and speaker diarization applications by [34] and [66]. Dixon et al. ([34]) offloaded the observation probability computation in a speech recognition engine to the GPU, while Kumar et al. ([66]) used the GPU to train Gaussian Mixture Models to obtain 164× speedup on these sub-computations over a sequential CPU implementation. EhKan et al. [35] implemented a Gaussian Mixture Model-based speaker identification system on FPGAs, achieving 90× speedup over sequential software version.

3.2.2 Paralleizing music processing applications

There has not been as much work on parallelizing music processing applications compared to the efforts in the speech recognition community. There has been some work done in parallelizing acoustic and audio rendering on GPUs by Tsingos et al. [103] as well as parallelizing signal processing kernels of music information retrieval (MIR) tasks on GPUs by Saviola et al. [94]. Both of them obtained one or two orders of magnitude faster performance than comparable systems. For music information retrieval tasks GPUs and multi-core CPUs have been used by

Battenberg et al. [6] to accelerate non-negative matrix factorization for an audio source separation application and by Ferraro et al. [38] to accelerate a query-by-humming application. These application also saw one to two order of magnitude performance improvement from GPU acceleration.

3.3 Summary

We use the audio content analysis application domain as the example domain for the software solution developed in this work. Audio content analysis applications use signal processing and machine learning techniques for extracting information and analyzing audio content such as speech and music. This domain presents a lucrative target due to a variety of reasons, including specific throughput and latency requirements. We choose four state-of-the-art audio analysis applications as the driving examples to help us with the design and implementation decisions throughout this work. These are speaker verification, speaker diarization, music recommendation and video event detection applications. These applications present a variety of interesting workloads with real-world constraints, comprising a good evaluation set for our approach.

Chapter 4

Pattern-Oriented Design

4.1 Finding the Common Vocabulary

Starting with the initial application design process, one effective software engineering practice is to use *software architectures* to design and develop applications [63]. Software architectures provide a systematic way of structuring applications as well as describing and communicating the application details to others. They consist of a set of boxes and arrows, each box corresponds to a particular computation and the arrows correspond the control-flow in the application. Software architectures are hierarchical - each box can be further decomposed into smaller software architectures, giving more detail of the particular computation. When designing applications, application domain experts can draw the software architecture on a white board and concisely describe what computation and control flow are present in the particular application.

The goal of our work is to create a software framework, PyCASP, that provides a productive way for application developers to reuse the work of efficiency programmers to create applications that are efficient and portable to a variety of parallel hardware. Our software environment needs to enable application developers to design and implement their applications based on software architectures. In order to create software architectures that are familiar to both application domain experts and efficiency programmers, we need a *vocabulary* for describing the boxes and arrows of software architectures. As we discussed in Chapter 2, design patterns provide a useful set of vocabulary for describing computations and structures of applications. They seem to fit our goal quite well. Design patterns provide a common language for designing libraries and frameworks based on software architectures and thus give a systematic way of thinking through applications and creating modular, self-contained software environments. The practice of using design patterns to create application frameworks has gained a lot of traction with the Our Pattern Language (OPL) project developed by Keutzer and Mattson [62].

OPL uses the methodology and taxonomy of parallel patterns. Previous libraries and frameworks described in [25] and [99] used the pattern-oriented design to enable modularity and provide comprehensive scope and efficient performance of applications. Thus, we choose to use OPL and pattern-based methodology for PyCASP's design. In addition to providing a language for our framework's users, patterns provide a systematic way to define the scope of our framework and provide a way to create a modular software environment that will comprehensively span a variety of audio content analysis applications.

4.2 Pattern-Mining Audio Content Analysis Applications

In order to use the OPL-based pattern-oriented methodology for designing PyCASP, we need to identify what computational and compositional building blocks are common in our application domain. Specifically, we need to identify three specific types of patterns that are common in state-of-the-art audio content analysis applications. Those are:

1. Application Patterns
2. Computational Patterns
3. Structural Patterns

Application patterns describe the *solutions to commonly-occurring problems in an application domain*. Computational patterns describe the *specific types of computations* that occur in each application pattern. Structural patterns describe the *control-flow of applications*. Using the pattern-oriented approach, identifying the core application, computational and structural patterns can allow us to understand the software architectures of applications in our domain. We hope that understanding software architectures of applications will give us a systematic way of developing a modular, comprehensive software environment that enables development of a large variety of applications in our domain. Furthermore, by supporting the primary application and structural patterns used in the audio content analysis applications we can enable the application writers to use a familiar vocabulary enabling their productivity, which is one of our primary goals.

We must be careful about how many patterns we choose to support. Too many patterns will lead to a convoluted framework scope and confuse the application developer, while too few patterns will not allow for comprehensive application coverage. Thus, our goal is to find a minimal set of patterns that we can use to construct a large variety of applications in the audio content analysis domain.

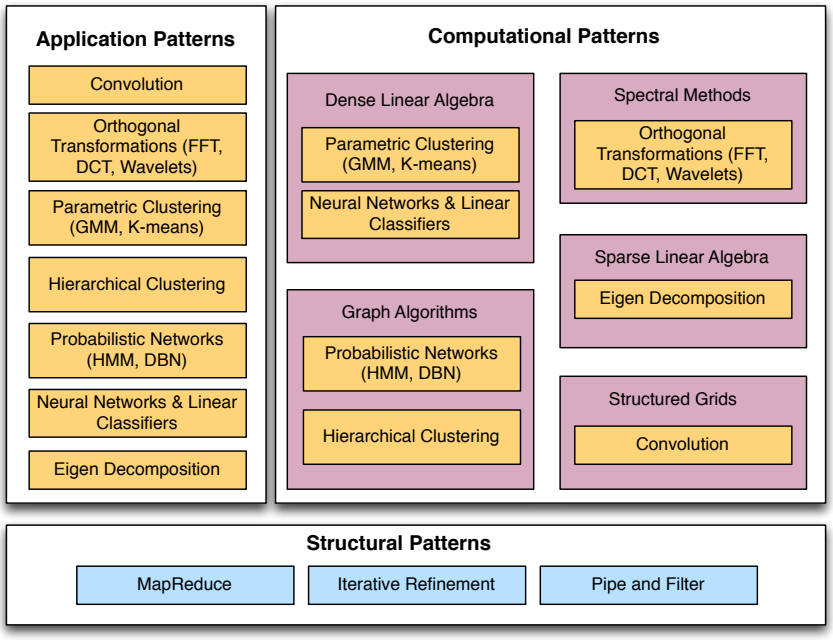

Figure 4.1: Application, computational and structural patterns in PyCASP.

The next three subsections describe in detail our process of identifying the three sets of patterns in audio content analysis applications.

4.2.1 Application patterns

Application patterns describe a set of solutions to a common set of problems in an application domain. Domain experts use application patterns as computational building blocks in their applications. In the audio content analysis application domain, application patterns are machine learning and signal processing techniques that have been developed, studied and standardized over time.

By studying a variety of applications in our application domain, we aim to determine the set of algorithms and patterns that are commonly used in these applications; we refer to this process as "pattern-mining". To perform pattern-mining in our application domain, we went through the following process. First, we extensively studied specific selected state-of-the-art audio content analysis applications we had familiarity with, such as speech recognition, music analysis and video event detection [4, 23, 26, 36, 111] to better understand what types and classes of algorithmic approaches they use. This helped us identify a set of candidate application patterns. Then, we went through accepted papers from the technical track of two top conferences in the area, Interspeech 2012 and ISMIR

2012, [1] to identify the common high-level algorithmic techniques used in state-of-the-art applications. We refined our original list of specific algorithmic approaches to include application patterns from these papers and then went through the list further grouping and refining it, based on similarity of the approaches (for example we grouped various clustering approaches together into the Parametric Clustering pattern). Finally, we had detailed discussions with domain experts in the field to confirm our categorization and gain a better understanding of how they think about the state-of-the art algorithmic techniques.

After our pattern-mining efforts, we obtained a list of seven application patterns, shown in the left panel of Figure 4.1. These seven patterns present high-level groupings of algorithmic techniques we mined from the application domain. These patterns are: Convolution, Orthogonal Transformations, Parametric Clustering, Hierarchical Clustering, Probabilistic Networks, Neural Networks and Linear Classifiers and Eigen Decomposition. Each of them can be decomposed into a list of specific algorithms, i.e. specific *instances* of the application pattern. We describe the application patterns below.

- **Convolution** is a common operation in signal processing applications. Given a function f and g, the convolution of the two functions is defined as the integral of their product after one is reversed and shifted. Applications of convolution in audio processing include computation of sound reverberation of the original sound with echos from objects surrounding the sound source, removing noise from the audio signal, as well as mapping the impulse response of a real-world environment to a digital audio signal.

- **Orthogonal Transformations** are common signal processing techniques to transform audio input to its orthogonal form. The most common orthogonal transformations are the discrete Fourier transform (DFT) and the discrete cosine transform (DCT) to transform the audio signal from time/space domain to the frequency domain [88]. These techniques are used in analyzing audio signals, as well as feature extraction, filtering and compression.

- **Parametric Clustering** is a machine learning technique of grouping data points into clusters where each cluster contains points that are similar by a particular metric. The clusters are defined by parameters, hence the "parametric" term. One simple example is the k-means clustering technique, where the N points are grouped into k clusters, each cluster contains points that are geometrically close to each other and the clusters are parametrized by their means. Gaussian Mixture Models (GMMs) are also a common parametric clustering technique. In GMMs, clusters are defined by Gaussian

[1] http://interspeech2012.org, http://ismir2012.ismir.net/

distributions, data points "belong" to the Gaussian component with highest likelihood defined by the Gaussian function (see Chapter 4). The cluster parameters are the weight, mean and covariance values of each Gaussian in the mixture. GMMs are used extensively in audio processing for clustering spectral features to create acoustic models of various audio sources.

- **Hierarchical Clustering** is a machine learning technique for creating a hierarchy of clusters. There are two types of hierarchical clustering: agglomerative and divisive. Agglomerative hierarchical clustering is a "bottom up" approach where each observation starts in its own cluster, and clusters are merged as one moves up the hierarchy until some optimization criterion is reached. The divisive hierarchical clustering is a "top down" approach where all observations start in one cluster, and the clusters are split as one moves down the hierarchy until a certain optimization criterion is reached. Hierarchical clustering is used in unsupervised machine learning processes where the final number of clusters is initially unknown and is learned during the clustering process. In audio processing, this technique is used for speaker diarization, where the task is to identify which part of a meeting recording belongs to each speaker and the initial number of speakers is unknown.

- **Probabilistic Networks** are a set of statistical modeling techniques that represent the model as a set of nodes in a directed graph. Each node represents a random variable and the directed edges represent conditional dependencies of the variables on each other. An example of a probabilistic network is the hidden Markov model (HMM) model where each node in the model is a hidden, unobserved event, but whose output (which is dependent of the event) is visible. Each node has a probability distribution over the possible outputs. The nodes are connected with transition edges that have the Markov property [86]. Probabilistic networks are useful in modeling data with temporal dependencies. For example, HMMs are widely used to model sequences of sounds and words in speech recognition applications.

- **Neural Networks and Linear Classifiers** are machine learning techniques for modeling data which assume that the observations from a particular process can be separated into classes by linear functions (i.e. the data is " linearly-separable"). When reasoning about prediction, linear models assume that the distribution of the output variable Y follows a weighted sum of observations X_i. The task of modeling Y is to determine the weights β_i that correspond to each random variable X_i. In addition, each random variable can be transformed by a non-linear function ϕ, a technique which is referred to as a basis expansion. The idea of the basis expansion is that if the observation data is not linearly-separable in the given space, applying a basis transformation will transform the data to a new (usually higher-

dimensional) space that will allow for linear separation. This technique is employed in one of the most common linear classifiers - the Support Vector Machine (SVM). SVMs are supervised binary linear classifiers that aim to find points in the training data (called support vectors) that separate the two classes with the largest margin (discussed more in Chapter 4). SVMs are extensively used in audio classification.

- **Eigen Decomposition** (also referred to as "spectral decomposition") is a linear algebra technique for factorization of a matrix in terms of its eigenvalues and eigenvectors. In audio processing, eigen decomposition allows for reasoning about components that make up a signal as well as transformation and modification of those components in a computationally-efficient way. A related approach to eigen decomposition called Singular Value Decomposition (SVD) is a common signal processing and statistics technique for separating an audio signal into its primary components. SVD is used in audio source separation applications whose task is to separate an audio signal into its components [114].

These application patterns correspond to a set of signal processing and machine learning techniques that are typically used to analyze audio content. Those techniques are composed together to create full-functioning audio analysis applications. For example, we found that in speech recognition, probabilistic models such as hidden Markov models (HMMs) are typically coupled with clustering models such as Gaussian Mixture Models (GMMs) or Neural Network models to model words, pronunciation and acoustics in speech recognition applications. In audio classification applications linear classifiers, such as Logistic Regression or Support Vector Machines (SVMs) are commonly used for audio content classification. In addition to summarizing the types of algorithms and techniques used in the application domain, this modular approach can help us to identify software already available in the community that may implement a specific computation on a given parallel hardware platform. We can then integrate existing solutions into the backend of our framework instead of reimplementing them from scratch. Using this pattern-oriented approach, we can develop a comprehensive framework for building parallel audio analysis applications leveraging our knowledge of the application domain and existing tools and software.

4.2.2 Computational patterns

Our next goal is to identify the set of *computational patterns* [62] that underlie the application patterns in the audio content analysis domain. Computational patterns describe the various types of computational workloads that can occur in software applications. They provide a useful way of summarizing and categorizing application patterns into classes of computations. Identifying the set of

computational patterns allows us to determine what types of computations our identified application patterns use. To identify the set of computational patterns, we start with the list of computational patterns in the OPL (see Figure 2.8) language. We then implement several example audio content analysis applications on a variety of parallel platforms and performed several levels of optimizations to gain a better understanding of the underlying computations of each application pattern. For example, we implement the Gaussian Mixture Model training and evaluation Parametric Clustering pattern and identify that the underlying computations are Dense Linear Algebra operations. For the application patterns that we do not have specific implementation examples, we study the theoretical mathematical formulations as well as look at academic references for those patterns that describe the implementation of those patterns. For example, we looked at specific papers describing the implementation of Singular Value Decomposition Eigen Decomposition pattern [114] to identify that the computational pattern the algorithm uses is Sparse Linear Algebra. From this process, we can identify what types of computations are present in each application pattern.

After our efforts, we determined that our seven application patterns use five computational patterns, as shown in the right panel of Figure 4.1. These patterns are Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods, Graph Algorithms and Structured Grids. We describe them in more detail below, additional information may be found at `https://patterns.eecs.berkeley.edu/`.

- **Dense Linear Algebra** are a large class of problems expressed as linear operations applied to dense matrices and vectors. Density is defined as the matrices and vectors having mostly non-zero elements. Solutions to this class of problems are defined in terms of basic linear algebra building blocks referred to as the Basic Linear Algebra Subroutines (BLAS). Application patterns that fall into the Dense Linear Algebra computational pattern are the Parametric Clustering and Neural Networks / Linear Classifier patterns.

- **Sparse Linear Algebra** are a large class of problems expressed as linear operations applied to sparse matrices and vectors. Sparsity is defined as the matrices and vectors having mostly zero elements. Sparse matrices come in a variety of shapes and there are several different formats for storing them [93]. Sparse linear algebra solutions can be direct or iterative, with iterative solutions relying on pre-conditioners that make finding the solution much faster. Application pattern that falls into the Sparse Linear Algebra computational pattern is the Eigen Decomposition pattern.

- **Spectral Methods** are a class of computational problems that involve systems that are defined in terms of several different representations. For example, a periodic sequence of a signal can be represented as a set of discrete points in time or as a linear combination of frequency components.

Going from one representation to another can reduce the computation of a difficult problem to a simple algebraic solution. For example, the Fast Fourier Transform (FFT) transforms an audio signal from the time domain to the frequency domain allowing for much faster signal filtering and transformations. Application pattern that falls into the Spectral Methods computational pattern is the Orthogonal Transformation pattern.

- **Graph Algorithms** are a large set of computational problems that operate on graphs. Graphs are made up of a set of nodes and edges (either directed or undirected). In order to use graph algorithms to solve a particular problem, the problem is first restructured as a graph. Then, graph algorithms such as breadth-first or depth-first traversal are used to understand the connectivity of the graph.The graph can be expanded or merged by operations on its nodes or edges, or graph partitioning algorithms can be used to partition the graph to prepare it for parallel processing. Application patterns that fall into the Graph Algorithms computational pattern are the Probabilistic Networks and Hierarchical Clustering patterns.

- **Structured Grid** are a set of problems that operate on discrete grids of data points, typically arising in computational science simulations. The grids represent natural systems (such as air space or biological processes) in terms of a discrete sampling of points, defined as a structured mesh (in contrast to unstructured mesh, which corresponds to the Unstructured Grid computational pattern). Each point in the mesh is updated using a function that operates on the mesh point and its neighbors; the set of neighbors is defined separately for each application. Application pattern that falls into the Structured Grid computational pattern is the Convolution pattern.

Identifying and distilling the application patterns to the core computational patterns allows for a more tractable framework scope and allows us to understand the computational properties of the application patterns. Moreover, we can identify specific parallel platforms for each computational pattern that allow for its most efficient implementation (for example GPUs are efficient at executing data-parallel dense linear algebra algorithms). Thus, computational patterns can give us more insight into what particular computation occurs in each application pattern and thus can guide us in selecting hardware platforms for the components that implement these patterns. They can also provide a modular organization of the application patterns for communication with the domain experts.

4.2.3 Structural patterns

At the final step in the pattern-based design process, our goal is to identify the set of *structural patterns* [62] that are common in audio content analysis appli-

cations. Structural patterns define specific ways in which application patterns are composed together in applications. They describe the arrows in the software architecture block diagrams of applications, and thus they describe the control flow of the computations in an applications. These patterns are very important when we try to understand the composition of patterns into applications as they describe the ways computational components are used together and describe how control- and data-flow between computations.

We start our structural pattern-mining process by leveraging previous work of Chong [25]. In his PhD thesis, Chong describes the details of speech recognition applications, their software architectures and pattern decomposition. Specifically, he identifies three structural patterns that are sufficient in constructing speech recognition applications: Pipe-and-Filter, Interator and MapReduce. Starting with this set of patterns, we set out to determine if these three patterns are in fact sufficient to construct most applications in the audio content analysis domain. In order to answer this question, we again turn to our pattern-mining process. We analyzed our sample applications and their software architectures in terms of the types of structural patterns they use. We looked at the types of data-flow patterns as well as control-flow patterns in each application. After careful consideration of a variety of audio analysis applications, our conclusion was that, indeed, the three structural patterns were sufficient to describe the software architectures of all of our sample applications. Thus, while we cannot guarantee that we have studied every single application in the domain, we are convinced that the three structural patterns are sufficient to describe the design of *most* applications in the domain.

Thus, to finish our pattern-mining process, we add these three structural patterns to the bottom panel of Figure 4.1 and describe them in more detail below:

- **Pipe-and-Filter** pattern refers to applications that are structured with data flowing through modular phases of computation. The solution constructs the program as state-less "filters" and "pipes" corresponding to the computational steps and data communication steps respectively. Data flows through the filters and pipes with the output of one filter flowing as input to the next filter.

- **Iterator** pattern refers to applications whose computation is repeated many times in a loop until either a particular number of iterations is reached or a certain termination condition is met. In each iteration, a particular set of computations is performed and then the iteration stopping criterion is computed and compared against the termination criterion. If those two conditions match, the iterative process terminates.

- **MapReduce** pattern refers to a class of computations that are structured in a set of parallel "map" and "reduce" phases. In the map phase computation is split into a set of independent computations and executed in parallel. In

the reduce phase, the results of the map phase are accumulated and either returned or processed further. The map and reduce phases can be chained together to yield more complex MapReduce applications.

We found that the structural patterns used in the audio content analysis domain correspond to *the way and the type of data is moved between components*. Since audio content analysis applications typically involve a set of machine learning and signal processing techniques, application patterns are composed into applications using data-flow: the output of one pattern is fed as input to the next pattern in a Pipe-and-Filter structural pattern or data is distributed across instances of a pattern in a MapReduce structural pattern. Thus, the arrows in a software architecture of an application correspond to the structural patterns, which in turn describe how and what types of data is communicated between the boxes / application patterns.

4.3 Pattern-Based Software Architectures of Example Applications

The pattern-mining process helped us identify the set of application, computational and structural patterns common in the audio content analysis application domain. We can now go through our four example applications and look in detail at their software architectures. Each software architecture consists of a list of boxes (corresponding to specific instances of application patterns) and arrows (corresponding to the structural patterns). The architectures are hierarchical - each box can be further decomposed into a smaller software architecture detailing the specific organization of that particular application pattern instance.

4.3.1 Speaker verification

Figure 4.2 shows the software architecture of the speaker verification application. For speaker verification training, at the top level, the feature extraction MFCC (instance of the Orthogonal Transformation pattern) is composed with the GMM supervector training (instance of the Parametric Clustering pattern) component using a Pipe-and-Filter pattern to compute the UBM parameters. The MFCC computation itself consists of several signal processing stages (FFT, Mel Scale, Log and DCT, each is an instance of the Orthogonal Transformation or Convolution application patterns) composed using a Pipe-and-Filter pattern. Then, given the UBM parameters, two parallel Pipe-and-Filter streams of MFCC feature extraction and GMM training are launched (one on our target speaker audio and one on intruder audio examples). The result of the GMM training (GMM parameters) are

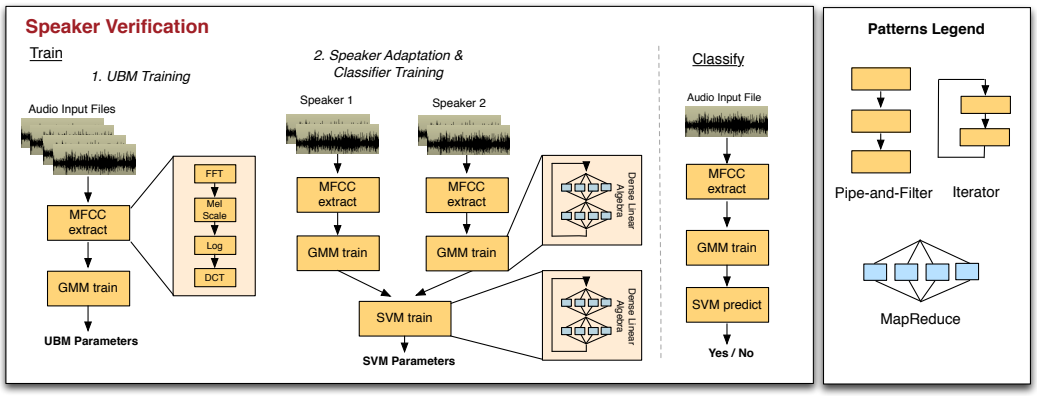

Figure 4.2: Software architecture of the speaker verification application. Internal architecture of each component is shown once.

passed to the SVM classifier training (an instance of the Linear Classification pattern) in a Pipe-and-Filter structural pattern. The GMM and SVM computations themselves are internally comprised of an Iterator structural pattern over several Dense Linear Algebra operations, each structured with a MapReduce pattern and composed using a Pipe-and-Filter pattern. The speaker verification classification phase consists of an MFCC Orthogonal Transformation pattern composed with a GMM Parametric Clustering pattern and a SVM Linear Classification pattern using a Pipe-and-Filter structural pattern. Algorithmically, the audio sample is processed to extract features, compute the adapted GMM parameters which are then passed to the SVM for classification of the audio.

4.3.2 Speaker diarization

Figure 4.3 (left) shows the software architecture of the speaker diarization application. At the top level, the diarization application consists of an Iterator pattern over a Pipe-and-Filter of GMM train, GMM evaluation (both instances of the Parametric Clustering application patterns) and Segment stages. The Segment stage is a loop that assigns chunks of audio to particular speakers based on GMM likelihoods from the GMM evaluation stage. The application iterates over GMM training, evaluation and segmentation phases until a stable number of speakers and audio segmentation has been identified. The MFCC feature extraction component and GMM training components consist of the same underlying computations and software architectures as in the speaker verification application. GMM evaluation computation internally is a Pipe-and-Filter structural pattern that composes several MapReduce stages, each implementing Dense Linear Algebra computations.

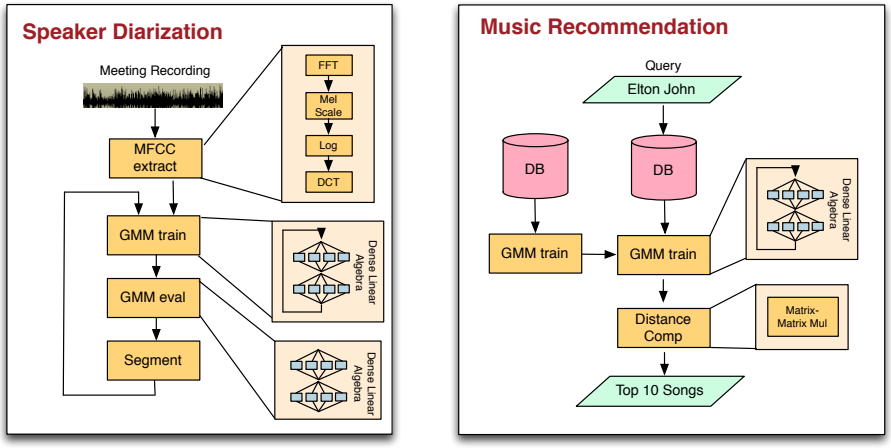

Figure 4.3: Software architecture of the speaker diarization and the music recommendation application.

4.3.3 Music recommendation

Figure 4.3 (right) shows the software architecture of the music recommendation application. At the top level, the GMM training (instance of the Parametric Clustering pattern) used to compute UBM parameters is composed with another GMM training pattern to adapt the UBM using features from a query song (in our example the query is "all songs by Elton John"). Then the result of the UBM adaptation is composed with the Distance Computation using a Pipe-and-Filter structural pattern to compute the set of top 10 most similar songs. The Distance Computation component is a matrix-matrix multiply operation, an instance of the Dense Linear Algebra computational pattern. GMM training instance of the Parametric Clustering pattern consists of an Iterator structural pattern that iterates over a Pipe-and-Filter composition of MapReduce stages, each MapReduce stage implements a Dense Linear Algebra computation.

4.3.4 Video event detection

Figure 4.4 shows the software architecture of the video event detection system. At the top level the application uses the MapReduce structural pattern with the map phase executing the diarization computation on each video and the reduce phase calling the K-means clustering to come up with the total list of audio events across all videos. The diarization computation has the software architecture described above.

After identifying the set of application, computational and structural patterns we are now able to describe the software architecture of each application using the pattern-based vocabulary. Now that we have identified the pattern-oriented

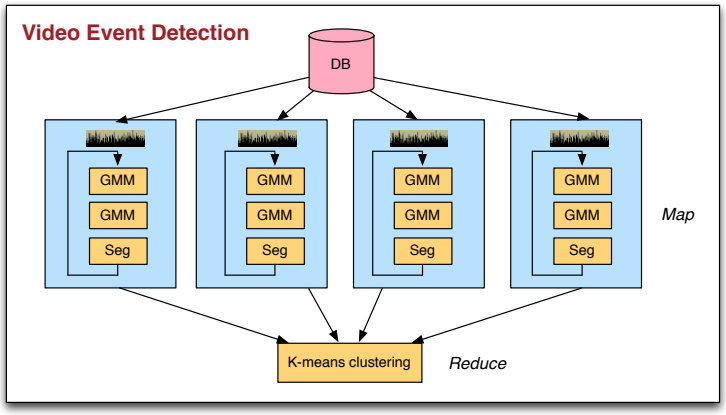

Figure 4.4: Software architecture of the video event detection system.

approach as a useful systematic way of designing application frameworks, we need to determine how this design will be implemented in software. The next chapter discusses this next step.

4.4 Summary

Pattern-based design provides a powerful methodology for designing a software environment that gives productivity, modularity and comprehensiveness while providing a common language for application writers and efficiency programmers. Patterns provide a vocabulary for designing and reasoning through software architectures of applications. Using the pattern-oriented approach, we have gone through the pattern-mining process to identify the set of application, computation and structural patterns that are used in the audio content analysis application domain. We then analyzed the software architectures of our four example applications composed of the three different types of patterns. Modularity and comprehensiveness are two primary goals for a software framework: it must support implementations of a variety of applications as well as provide modularity for clarity, debugging and scalability. Reasoning about composition in the audio analysis applications using structural patterns gives us a very powerful approach to gain modularity and clarity in application design. This clarity is extremely important when developing a framework that targets two types of programmers and aims to close the implementation gap. Each programmer must understand the framework abstractions and scope in order for the framework to serve a a software tool for both types of programmers. Thus, we choose the pattern-oriented design for defining the scope and the vocabulary of PyCASP. In the next chapter, we focus on implementation mechanisms for realizing our design in software.

Chapter 5

Implementing Patterns Using SEJITS

In the previous Chapter, we stepped through the design process of PyCASP using design patterns. In this chapter we set out to determine how our design will be realized in software. Pattern-based design enables our framework to provide programmer productivity; however, we must enable our other goals: efficiency, flexibility and portability. In the software implementation of our software framework we need to ensure that we achieve programmer productivity and flexibility as well as application efficiency and portability.

Previous successful implementations of productivity frameworks such as [14], use code specialization and JIT compilations from a high-level language to achieve efficiency, flexibility and portability. Specifically, a technique called Selected Embedded Just-in-Time (JIT) Specialization (SEJITS) has shown a lot of potential in bridging productivity, flexibility, efficiency and portability [17, 57]. By specializing selected functions in an application written in a high-level language, SEJITS is able to achieve application efficiency that is close to a hand-tuned low-level implementations [57]. In addition, SEJITS can support arbitrary computations defined by a domain specific language (DSL) that is created by the framework designer. This seems to fit our goals quite well. In order to achieve efficiency, flexibility and portability in addition to productivity we need a software mechanism that will specialize particular computations to a variety of parallel hardware and provide customizations and flexibility to the application programmer. Thus, we choose to employ the SEJITS methodology to implement our framework in software.

To make a decision about what kind of high-level language and the specializer development environment we will use for PyCASP, we look at successful implementations of SEJITS frameworks. Both Copperhead [17] and Asp [57] use Python. Python has recently gained traction as a high-level productivity language for scientific computing. It provides a free, high-level interpreted scripting

environment that is easy to use. A large array of tools and libraries is available for Python, making it very versatile for application development. For example, packages like Scipy [55] and Numpy [5] provide a variety of tools and functionality for scientific and numerical computing. Furthermore, Python is highly extensible - it allows for easy embedding of low-level modules using Python wrappers; in fact most functions in Scipy and Numpy are implemented using low-level languages (such as C) and then wrapped and exposed to Python. This extensibility allows for separation of concerns between the application and efficiency programmers. Thus, following the example of productivity frameworks and SEJITS methodology, we embed PyCASP in Python.

To choose the specializer development environment, we analyze previous SEJITS frameworks [17, 57, 7]. The Asp framework [1] presents a compelling implementation of SEJITS [57] as it provides methodology to create *domain-specific* languages (DSLs) and specializers. Asp contains facilities to automate the process of creating specializers, either based on templates or AST (abstract syntax tree) transformations, to emit source code corresponding to different code variants, compile and call the optimized code, and pass the results of the computation back to the Python interpreter. It has shown a wide applicability to a variety of application domains such as stencil computations, linear algebra and graph algorithms [58]. Thus, we choose to use it as the framework to develop SEJIT specializers for PyCASP.

5.1 From Patterns to Software

After outlining the scope of PyCASP using design patterns, and deciding to embed it in Python and use the Asp SEJITS framework to realize the patterns in software, we need to determine the API of our framework. The target audience for our framework are audio content analysis domain experts. As mentioned earlier, domain experts use application patterns as computational building blocks in their applications. Application patterns are familiar to audio analysis application writers as machine learning or signal processing techniques and thus have a clear scope and functionality. Application developers can customize specific algorithmic details of each pattern based on parameters of the specific algorithmic techniques. In addition, because of the clarity of scope and functionality, efficiency programmers can implement specific instances of application patterns on parallel hardware as stand-alone components. Thus, application patterns present a great candidate for the level of abstraction of our framework. Efficiency programmers can implement specific instances of application patterns as SEJIT specializers and application domain experts can then use the application patterns in their applications. Computational patterns allow us to understand the underlying computation of each

[1] Asp stands for "A SEJITS for Python".

application pattern to enable efficiency programmers to optimize the underlying computations. We realize application patterns as SEJIT specializers; these specializers become *components* of PyCASP.

Following our implementation decisions, we choose to develop a set of components of our framework that correspond to specific instances of the application patterns. For example, a component can be a Gaussian Mixture Model (GMM) training and evaluation specializer, which is an instance of the Parametric Clustering application pattern. Each instance of an application pattern has a specific scope and functionality, but can be customized to fit a particular application. Some application patterns are more customizable than others. For example, a GMM parametric clustering pattern has fixed functionality: training and evaluating GMMs, while Hierarchical Clustering application pattern can have several specific implementations depending on the types of models that are clustered, and thus, is more customizable. Thus, we design PyCASP to contain two types of components:

- Library components: have fixed functionality, fixed (maybe multiple) parallelization strategies, and a fixed software architecture.
- Customizable components: have customizable functionality, a fixed software architecture, and their parallelization strategy is determined at runtime.

Table 5.1 shows the broad scope of PyCASP. The table shows example customizable and library components and the application patterns they correspond to. Library components are *specific, common customizations* of customizable components, i.e. they implement common ways a specific component can be customized. For example, the Expectation-Maximization algorithm can be used to learn parameters of many clustering models. Using EM for GMM training is a specific common use of the EM training customizable component. By knowing the specific use case of a particular customizable component, we can implement a much more efficient version of the specializer. For the EM training example, targeting the component to the specific instance of using EM to train GMMs, we can implement GMM EM training as a "black box" specializer that uses low-level code templates to generate code. This enables us to achieve higher efficiency of commonly-used customizations of particular components of PyCASP.

Customizable components allow for more flexible user customizations. Customizations are familiar to the application writers as parameters of machine learning and signal processing techniques. For example, the application writer may want to specify the *functionality* of components instead of relying on fixed functionality. To customize the component, programmers can overload specific methods of the specialized class. For example, in the Hierarchical Clustering component, programmer may specify the information criteria during model selection (BIC, MDL, AIC [64]), or the merging function (parameter concatenation,

Application Pattern	**Customizable Component**	**Library Component**
Parametric Clustering	EM training (Model)	GMM training
Parametric Clustering	Geometric Neighbor Computation (Distance)	L-norm geometric (K-means)
Hierarchical Clustering	Hierarchical Clustering (Model, Merging)	Agglomerative Clustering (GMMs, BIC)
NN & Linear Classifiers	SVM (Kernel function)	SVM (Linear, Polynomial, Gaussian)
NN & Linear Classifiers	NN Classifier (Smoothing function)	NN Classifier (Softmax smoothing)
Convolution	Wiener filter (Noise model)	Wiener filter
Orthogonal Transformations	—	FFT, DCT, MFCC
Probabilistic Networks	HMM (Observation Model)	Speech Decoder (GMM)

Table 5.1: The broad scope of PyCASP's components. First column lists the application pattern, the customizable component column gives an example customizable component that is an instance of the application pattern. The customization point is given in parenthesis. The library component column gives an example library component, i.e. a specific implementation of a customizable component.

averaging etc.). In the hidden Markov model component, the programmer can specify the function to compute the observation probability given a hidden state (Gaussian, mixture of Gaussians, discrete etc.).

We can use the SEJITS technology to implement both types of components. The two types of components match the Asp framework functionality quite well: library components can be implemented using Asp templates and customizable components can be implemented using AST transformations and code lowering methods. Since the scope of PyCASP is quite broad, in this work, we restrict our focus to the library component implementations and leave customizable components as future work. Customizable components enable higher application programmer flexibility and thus are an important aspect of future work and will be discussed more in Chapter 8.

Library components implement specific types of computation and thus allow for high efficiency of the generated code. Restricting the scope of library components allows us to abstract away all of the internal functionality of a component and thus extract the most performance out of the particular pattern. We can then allow application writers to call these components from Python code as a black box, without thinking about their internal functionality. Using the SEJITS technology, we can allow PyCASP to automatically handle all underlying

parallelization and implementation details. In addition, when a computation has multiple parallelization strategies, the specializer can choose logic about what strategy to use, transparent to the application writer.

5.2 Efficiency and Portability Using SEJITS

Chapter 2 describes the background SEJITS approach we use to enable *efficiency*, *flexibility* and *portability* in application frameworks. We now step through the specific process of creating a specializer using Asp templates that we use to implement the library components of our framework.

Creating a specializer

The Asp framework provides functionality for creating specializers as Python classes. To create a template-based Asp specializer, specializer writers typically start with an efficient version of a particular instance of an application pattern and a target hardware platform (for example GMM training on NVIDIA GPUs). They carefully analyze the low-level implementation and identify tuning/specialization parameters for each function that need to be adjusted for a particular instance of a hardware platform such as number of thread blocks and number of threads per block in a CUDA implementation. They also define the data movement logic, if needed, for example moving training data to and from the GPU memory. They can also have multiple code variants of the same computation, for example different blocking strategies or loop reordering, whose efficiency differs depending on particular hardware and input data parameters.

The efficient implementation(s) for a particular component are then translated into `.mako` templates and placed into Asp code modules. Templates contain the original low-level code as well as place-holder variables for the tuning/specialization parameters for each function and modules to plug in different code variants of the same computation. The specializer logic specifies *how* to populate the place-holder values and select the code variants (for example, by querying for the specs of the hardware platform or the shape of the input data) and what compiler toolchain to call on the generated code. All of the specializer logic is implemented in Python, only the template code is implemented in a low-level efficiency language (this allows for *specializer writer productivity*, and is one of the advantages of Asp). When the specialized function gets called, Asp renders and transforms these templates into syntactically correct source code via the templating library Mako [69], and compiles, caches and links them using the Python extension CodePy [28]. During template rendering, Asp populates the place-holder variables, selects appropriate code variants according to specializer logic, and calls the appropriate compile toolchain on the resulting code. Once the code is compiled, it is automat-

ically executed and the results are brought back to Python. The compiled object code is cached to avoid redundant recompilation. This is the general mechanism for achieving *efficiency* in the component implementation.

To allow for *portability*, the specializer writer has to target multiple low-level backends; however, they can typically be grouped together into a few "general" backends, i.e. one CUDA specializer backend can target *all* CUDA-programmable GPUs, while another Cilk+ specializer can target *all* Cilk-programmable Intel x86 hardware. Thus, while it does take significant amount of effort to implement a specializer, it is intended to be done by an efficiency programmer who is familiar with low-level intrinsics of the hardware and typically implementing the same computation for multiple backends is not as difficult once the programmer is familiar with the algorithm. In addition, this effort has to be done *once*, and can then be reused by all application programmers who use resulting framework without knowing any details of the specializer implementation. At the cost of increased specializer developer coding time, this approach significantly improves application developer productivity.

Finally, given a multi-layered structure of the specializers, special debugging techniques need to be employed to guarantee specializer correctness, such as [109]; this is generally ongoing research work. Debugging applications that use specializers can be facilitated by ensuring the specializes have high code coverage and undergo thorough regression testing. Since specializer creation is an isolated process, this process is fairly self-contained. The application code is then written in compact Python using the specializers, which facilitates isolation of user errors.

We now illustrate the use of the Asp template-based approach to implement two sample components of PyCASP: the GMM training and evaluation component (instance of Parametric Clustering application pattern) and the Support Vector Machine (SVM) training and classification component (instance of Linear Classification application pattern). We choose these two components as they are extensively used in many audio analysis applications and thus, have a high potential of reuse across a variety of applications. Gaussian Mixture Models are extensively used in acoustic modeling for speech recognition and other audio applications, while Support Vector Machines are often used for classification of audio content. In addition, some audio analysis applications employ both components (such as a GMM-SVM speaker verification system) and thus we can use these two components to analyze component composition (discussed more in the next chapter).

Our goal is to illustrate in detail the underlying algorithms and specialization mechanisms for each component. Using the SEJITS approach we aim to enable efficiency and portability of the components in addition to application writer productivity. After designing and implementing the two components, we evaluate their performance both in terms of efficiency and portability and discuss the results in Chapter 8.

5.3 Parametric Clustering

As described in Chapter 4, Parametric Clustering application pattern corresponds to a machine learning technique of grouping data points into clusters where each cluster contains points that are similar by a particular metric. For our first component of PyCASP, we choose to implement the Gaussian Mixture Model (GMM) parametric clustering technique. GMMs are widely used in audio processing, as we have seen in our example applications, and thus, they present a lucrative target for specialization.

5.3.1 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are one of the most widely-used parametric probabilistic models for clustering data. GMMs are parametric probability density functions that consist of a weighted set of Gaussian component densities. A GMM is a weighted sum of M Gaussian components, each Gaussian g_i is parametrized by a D-dimensional mean μ_i and (full or diagonal) $D \times D$-dimensional covariance matrix Σ_i. The GMM is thus parametrized by the set of mean and covariance parameters and the weights π_i. x is a D-dimensional feature vector.

$$g(x \mid \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i)\} \tag{5.1}$$

The probability of observing data x_j for a Gaussian mixture then is:

$$\begin{aligned} p(x_j \mid \mu_i, \Sigma_i) &= \sum_i \pi_i g(x_j \mid \mu_i, \Sigma_i) \\ &= \sum_i \pi_i \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x_j-\mu_i)^T \Sigma_i^{-1}(x_j-\mu_i)\} \end{aligned} \tag{5.2}$$

Given N observed data points (x_js), each a D-dimensional feature vector, we need to learn the parameters μ_i, Σ_i for each component and the weight parameters π_i for combining them into the overall mixture model.

GMM parameters can be estimated from training data using the Expectation-Maximization (EM) algorithm [9] using a data set of N D-dimensional training data points. EM is an iterative algorithm. Given an initial estimate of the parameters, the EM algorithm iterates between two phases: the E-step and the M-step.

- The E-step computes the expectation of the log-likelihood of the events (i.e. the observations) given parameter estimates,

$$p_{i,j} = \frac{\pi_j^k g(x_j \mid \theta)}{\sum_i \pi_i g(x_j \mid \theta)} \tag{5.3}$$

where $p_{i,j}$ is the probability of event j belonging to component i and k is the iteration number.

- The M-step in turn computes the parameter estimates that maximize the expected log-likelihood of the observation data.

$$\pi_i^{k+1} = \frac{\sum_j p_{i,j}}{N} \tag{5.4}$$

$$\mu_i^{k+1} = \frac{\sum_j x_j p_{i,j}}{\sum_j p_{i,j}} \tag{5.5}$$

$$\Sigma_i^{k+1} = \frac{\sum_j (p_{i,j}(x_j - \mu_i^{k+1})(x_j - \mu_i^{k+1})^T}{\sum_j p_{i,j}} \tag{5.6}$$

- These two steps repeat either a pre-specified number of times (number of EM iterations) or until the updates of GMM parameters become epsilon-small, i.e. a convergence criterion is reached.

GMMs are very useful in modeling biometric data such as human voice or other sounds, due to their ability to represent a large class of variable distributions. GMMs have been used extensively in speech recognition [12, 39, 40, 41, 92] and music information retrieval [20, 91] to model sound variability and compute audio similarity.

5.3.2 GMM component overview

Our goal is to design and implement the GMM component such that it enables efficient execution of the EM algorithm on a variety of parallel platforms while enabling application developer to use the component productively, in a high-level language. Earlier in this chapter, we developed a methodology to enable these aspects by encapsulating the computations for a specific instance of an application pattern in a template-based specializer using Asp. The API for this specializer has to correspond to the standard way application developers use GMMs (we can look at example GMM libraries, for example Scikit-learn [96]); it needs to provide the following parameters:

- `num_components`: set the number of components in the GMM

- `num_iterations`: set the number of EM training iterations.
- `weights`: set weights to a particular vector value.
- `means`: set means of GMM to particular values.
- `covers`: set covariance matrices of GMM to particular values.
- `covariance_type`: set type of covariance matrix.

and support the following functions:

- `train(data_set)`: train a GMM given a set of training data using the EM algorithm.
- `score(data_set, GMM_parameters)`: compute the log-likelihood of a given observation given an already trained GMM.
- `aposteriori(data_set, GMM_parameters)`: compute the aposteriori probabilities for each Gaussian in the mixture.
- `get_parameters()`: return parameters of a trained GMM (weights, means, covariances).

An application programmer needs to be able to call each function from Python code and customize the computations using this set of parameters.

We aim to enable portability of our GMM component to a variety of platforms available on a typical application developer desktop. Thus, we set the GMM component to target two classes of platforms: NVIDIA CUDA and Intel Cilk+ [27] backends.

5.3.3 GMM component implementation

We aim to support two different types of backends - CUDA-programmable GPUs and Intel multi-core CPUs (with Cilk+ programming environment). The most important and computationally-intensive function that our GMM component needs to support is GMM training, thus, we focus on this function.

GMM training on the GPU

For the GPU backend, we start with the CUDA code implemented by Pangborn [80] for training GMMs; this code contains an implementation of the EM training algorithm as well as the functions for computing the likelihood given an already trained GMM. We analyze this code and further optimize the CUDA kernels. In this original code, each kernel is custom-written for each particular phase of the

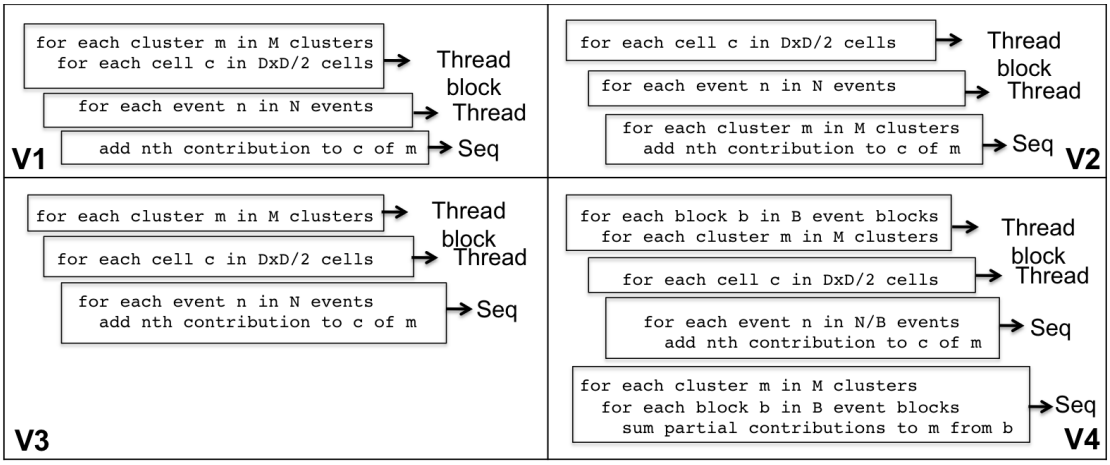

Figure 5.1: Four code variants for computing the covariance matrix during M step. The computation loops are reordered and assigned to thread blocks and threads as shown above. The "Seq" part of the computation is done sequentially by each thread.

EM algorithm, thus we pay careful attention to the tuning and parallelization strategies available for each phase. There are three integers that define the problem size: M - the number of Gaussian components, D - the dimensionality of the Gaussians and N - the number of feature vectors.

By carefully analyzing the computation of GMM training EM algorithm, we discover that there are several parallelization strategies available for the most compute-intensive phase of the EM algorithm, the covariance matrix computation during the M-step of the computation (Equation 5.6) which takes about 60% of the total computation time. The covariance matrix computation exhibits several degrees of available parallelism, namely:

- Computation of each component's covariance matrix is independent,
- Computation of each cell of each matrix is independent,
- Computation of each feature vector's contribution to a cell in each covariance matrix is independent.

Thus, given the several degrees of parallelism in the computation, we can derive four different parallelizaiton strategies that can be employed to perform the covariance matrix computation during GMM training. The parallelization strategies are summarized in Figure 5.1. Each code variant differs in what compute entity the different dimensions of the problems are assigned to (either thread

blocks, threads or sequential processing) as well as the different uses of per-block shared memory. Thus, the efficiency of each version depends on the values of M, D and N as well as the hardware characteristics of the GPU. We describe the implementation code variants below.

Code Variant 1 (V1), we use as baseline: This is the original implementation of the EM algorithm from Pangborn [80]. This strategy launches $M \times D \times \frac{D}{2}$ thread blocks - one for *one* cell for *one* component's matrix (shown by the first two for loops in Figure 5.1(V1)). Threads correspond to the loop over features (N). The mean vector is stored in local per-block shared memory, however only two values are used (corresponding to the row and column of the cell the block is computing).

Code Variant 2 (V2): Modifies Code Variant V1 by assigning each thread block to compute *one* cell for *all* components. Thread blocks correspond to the loop over $D \times \frac{D}{2}$ cells in the matrix. Threads correspond to feature vectors as in V1.

Code Variant 3 (V3): Makes better use of per-block shared memory by assigning each thread block to compute the *entire* covariance matrix for *one* component (M). Each thread in the thread block is responsible for one cell in the covariance matrix ($D \times \frac{D}{2}$ threads). Each thread loops through all events sequentially.

Code Variant 4 (V4-BXX): Improves upon V3 by making it more resilient to small number of components (M) by adding blocking across the N dimension. Launches $M \times B$ thread blocks, where B is a blocking factor, i.e. the number of desired feature blocks. Each thread block computes the contribution to its entire covariance matrix for its block of features ($\frac{N}{B}$), followed by a *sum()* reduction over the partial matrices across all feature blocks (Figure 5.1(V4) shows the additional blocking and reduction loops). In this work we use two values of B, 32 and 128.

In order to understand the trade-offs between the different code variants, we can test all the variants on latest GPU cards to see whether the variants' performance is predictable and can be programmed using logic in the speicalizer. We use NVIDIA GTX285 and GTX480 GPUs to analyze the trade-offs of the code variants; we use a regular sampling of problem sizes that are typical in audio content analysis. The GTX285 has more CUDA cores, but the GTX480 has longer SIMD vectors and better atomic primitives.

The best variant depends on both the problem size and the underlying machine. V1 and V2 have a large amount of thread parallelism while V2 has limited thread block parallelism if D is small. V3 can be limited in both thread block

and thread parallelism if M and D are small. V1 underutilizes the local memory and requires many streaming reads, whereas V2 and V3 utilize local memory more efficiently and V2 requires a factor of M fewer feature data streaming reads than V1. V4 mitigates V3's limited thread block parallelism by blocking over N, but requires a global reduction across B thread blocks, incurring synchronization overhead due to atomic operation latency.

Figure 5.2 shows some example results that we obtained form this experiment. Overall, for the space of problem sizes that are typical in our application domain ($1 \leq D \leq 36, 1 \leq M \leq 128, 10,000 \leq N \leq 90,000$), the best-performing code variant for a given problem instance gave a 32% average performance improvement in covariance matrix calculation time compared to always running the baseline code variant V1. This performance gap increases further with larger problem sizes, e.g. for ($D = 36$, $M = 128$, $N = 500,000$) the difference grows to 75.6%. Figure 5.2 plots a slice through the 3D space of possible input parameters for $N = 10,000$ and $N = 90,000$, allowing the average runtimes of different implementations of the covariance computation to be compared. We see that V1, V3 and V4 with different B parameters are mutually competitive and show trade-offs in performance when run on various problem sizes and on two GPU architectures. V2 shows consistently poor performance compared to the other code variants. While there are general trends leading to separable areas where one variant dominates the others (e.g. V1 is best with small D values for $N = 90,000$), we had difficulty formulating a hierarchy of rules to predetermine the optimal variant because each hardware feature affects each variant's performance differently. This finding suggests that variant selection cannot necessarily be reduced to a compact set of simple rules, even for a specific problem in a restricted domain of problem sizes.

Thus, we see that there is no simple logic that can be implemented in the specializer to perform variant selection. Instead, we can use a mechanism called "auto-tuning"' (pioneered in numerical libraries such as Atlas [106] and LAPACK [2]) to automatically select the best-performing variant by running, timing and selecting the best variant given a particular problem size and hardware configuration. Using auto-tuning, we test each variant on the given problem size and time its performance, after which, we select the best-performing variant and use it in all consecutive calls to the GMM training function. We can store the results of the auto-tuning process in a database provided by Asp so that we don't have to perform the (time-consuming) auto-tuning process every time we use the PyCASP's GMM component.

To continue with the implementation of our GMM component, we implement all code variants for covariance matrix computation in CUDA and add them to the optimized CUDA implementation from Pangborn. This gives us the CUDA implementation of all functions for our GMM component. We then transform the CUDA implementation to `.mako` templates, following the Asp methodology of

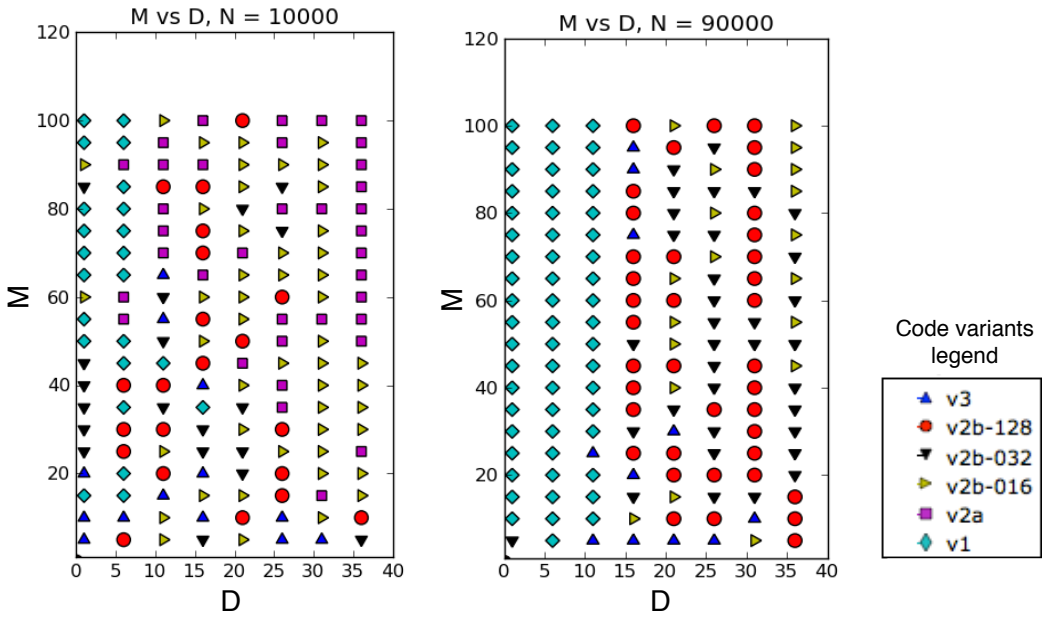

Figure 5.2: GMM code variant performance on NVIDIA GTX480 GPU with varying M and D parameter values for $N = 10,000$ (left) and $N = 90,000$ (right) training points. Each point shows the "winning" code variant (i.e. the code variant yielding fastest execution time of the training algorithm). Code variant legend is in the bottom right.

creating template-based specializers. Thus, as component/specializer writers, we hand-code templates for the CUDA implementations of GMM functions and code variants described previously in low-level code and translate them into template files that are then used by Asp to generate syntactically-correct CUDA code.

Because we use CUDA as our low-level implementation language, the GMM component emits two kinds of lower-level source code: C++ host code that calls CUDA kernels and controls the algorithm logic (such as convergence loop computation), and CUDA GPU kernel code that implements the different parallel kernels for training and evaluation of GMMs. Because the internal code structure of the different covariance matrix creation kernel variants is very dissimilar, we found it simplest to include all variants of the kernel in every dynamically-linked library generated by the specializer; instead of specializing the kernel, we specialize the host C code, causing it to call the best kernel based on the particular covariance matrix type, problem size and available hardware.

Thus, the specialization logic of our GMM component has two jobs: variant selection and parallel code parameter selection.

- **Variant selection.** As discussed above, one of the jobs of our GMM component is the variant selection logic. The best variant to use depends on properties of the input problem data (the size of M, N, and D). We choose to select a variant by telling Asp to examine the values of the parameters passed to the specialized function, and to treat functions with different parameter values as different functions. The current Asp implementation tries a different variant every time a particular function/problem size is specialized, until all variants have been tested. Thereafter, the component remembers what the best runtime was for that particular function/problem size, and always reuses the associated variant's cached binary. This variant selection method is naive; instead, it is desirable to use performance models or machine learning to make decisions about what variant to use without exhaustively searching all options. This will be addressed in future work. However, the important observation for this present work is that the mechanism and policy for variant selection can be well-encapsulated and can be replaced without touching the original application source code or the code variant templates themselves, allowing us to enable application writer productivity *and* application performance.

- **Parallel code parameter selection.** Our component has to also populate the template parameters for the `.mako` templates in order to generate valid CUDA code. The component can choose the optimal number of threads and thread blocks to use by querying the GPU specs and replacing the template parameters with those values. Thus, because the components emits valid CUDA code that is tuned to the particular GPU, we enable our GMM component to run on any CUDA-programmable GPU allowing for efficiency and portability across generations of parallel hardware.

GMM training on a multi-core CPU

We follow a similar methodology to enable our GMM component to specialize computation to run on the multi-core CPUs. We start by implementing our own Cilk+ version of the GMM training and likelihood computation functions. We use `cilk_for` keywords around loops whose elements can be computed in parallel in the E and the M stages. We also use the reducer operators to perform cumulative computations (reductions) in our training code. The number of processors that get utilized is controlled by `CILK_NWORKERS` variable without change to the specializer code. We then translate the Cilk+ code to `.mako` templates and enable our specializer to make decisions about the number of threads to launch for each computation. We also add compiler flags such that our specializer can use the Intel compiler optimizations to tune the code to the particular hardware platform. We follow the same mechanisms as in our CUDA implementation to select the

parallelization parameters such that the Cilk+ backend of the component can run on any Intel multi-core CPU.

Using the GMM component in Python code

After initial implementation in CUDA and Cilk+, creating templates and using Asp to implement logic of template rendering, code generation and compilation, we develop a pipeline for implementing functions of our GMM component. We follow the same methodology to implement all of the functions specified by our initial API design. We also add Python accessor functions to enable reading and writing of the GMM parameters from the GMM objects. Thus, using Asp and Python we enable our application developers to use the CUDA and Cilk+ implementations of the GMM component from Python application code. We use the SEJITS technology (embodied in Asp) to perform the specialization and code generation processes to enable easy use of our GMM functions. Figure 5.3 shows an example usage of the GMM component in Python application code.

This code snippet shows the example use of the GMM component in an application that trains a Gaussian Mixutre Model on a training data set (stored as a Numpy array) and then computes the log likelihood of testing data using the trained model. The application first imports Numpy and the GMM component of PyCASP (lines 1 and 2). It then reads in the training and testing data sets (lines 5 and 8) and creates a GMM object (line 17). Then the GMM is trained on the training data set (line 20) and evaluated on the testing data set (line 23). When the training and evaluation calls are made on the GMM objects, the specializer triggers the set of steps that generate, compile, link and execute efficiency-level code on parallel hardware. To run the same application on a different GPU, the specializer automatically changes the values it uses to populate the templates when the GMM training function is called. This example illustrates how we are able to achieve productivity, efficiency and portability of applications using patterns and SEJITS for our design and implementation of PyCASP.

5.4 Linear Classification

5.4.1 Support Vector Machines

Support Vector Machines (SVMs) are one of the primary machine learning techniques used for content classification. Given a set of labeled training examples (consisting of [feature_vector, label] pairs) and a set of unlabeled target examples (consisting of [feature_vector] values), the task of an SVM is to produce a model based on the training data that can predict the labels of the target set [31]. More concretely, given a training set of feature-label pairs $(x_i, y_i), i = 1, ..., n$

```
1 import numpy as np
2 from gmm import *
3
4 # get numpy array of training data
5 training_data = get_training_data()
6
7 # get numpy array of testing data
8 testing_data = get_testing_data()
9
10 # set number of GMM components
11 M = 16
12
13 # get dimensionality of the data
14 D = training_data.shape[1]
15
16 # initialize the GMM, use diagonal covariance
17 gmm = GMM(M, D, cvtype='diag')
18
19 # train the GMM
20 gmm.train(training_data)
21
22 # evaluate the GMM to get log likelihoods on test data
23 log_lkld = gmm.score(testing_data)
```

Figure 5.3: Example usage of the GMM component.

where $x_i \in \mathcal{R}^n$ and $y \in 1, -1^n$, the SVM aims to find a solution of the following optimization problem:

$$\max_{\alpha} F(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\alpha^T Q \alpha \tag{5.7}$$

$$Q_{i,j} = y_i y_j K(x_i, x_j) \tag{5.8}$$

subject to

$$0 \leq \alpha_i \leq C, \forall i \in 1, ..., l; y^T \alpha = 0 \tag{5.9}$$

Where x_i are training feature vectors, y_i are the labels attached to x_i and α_i are a set of weights, one for each training example (which are the optimization parameters). C is an error parameter that trades off classifier accuracy with generality.

SVMs use a machine learning technique called basis expansion to project the feature vectors to a higher-dimensional space in order to better separate the training examples into two classes. The transformation is done using the function $\phi()$. Specifically, using a *kernel function*, $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$, this transformation

can be done efficiently without having to explicitly compute the values of $\phi()$ by taking advantage of the fact that only *dot products* of feature vectors are required for the optimization problem; this is commonly referred to as the *kernel trick*. There are several "standard" kernel functions:

- Linear. $K(x_i, x_j) = x_i^T x_j$
- Polynomial. $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$
- Gaussian. $K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$
- Sigmoid. $K(x_i, x_j) = tanh(\gamma x_i^T x_j + r)$

where r, d and γ are kernel parameters specific to each function.

The task of the SVM optimization is to find a linear separating hyperplane in this higher-dimensional space that has the maximum separating margin between examples from the two classes.

There are several ways to solve the SVM training optimization problem. We use the optimized implementation by Catanzaro et al. [16] as the baseline for our SVM component since it was shown to be the most efficient implementation of the SVM training and classification algorithm. This implementation uses the Sequential Minimal Optimization (SMO) algorithm [83] to solve the SVM training problem. The algorithm takes advantage of the sparse nature of the optimization problem and reduces the optimization steps to first, determining and then updating two weights α_i. The remainder set of weights is updated using the Karush-Kuhn-Tucker optimality conditions which are maintained using a vector $f_i = \sum_{j=1}^{l} \alpha_i y_i K(x_i, x_j) - y_i$.

The SMO algorithm we choose to use for our SVM component (from [16]) is as follows:

1. Initialize $b_{high} = -1$, $i_{high} = min\{i : y_i = 1\}$, $b_{low} = 1$, $i_{low} = min\{i : y_i = -1\}$
2. Initialize $\alpha_i = 0$, $f_i = -y_i, \forall i \in 1, ..., l$
3. Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
4. Update $f_i, \forall i \in 1, ..., l$
5. Compute b_{high}, b_{low}, i_{high}, i_{low}
6. Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
7. Repeat steps 5 - 7 until $b_{low} \leq b_{high} + 2\tau$, for a convergence parameter τ

The update functions are as follows:

$$\alpha'_{i_{low}} = \alpha_{i_{low}} + y_{i_{low}}(b_{high} - b_{low})/\eta \tag{5.10}$$

$$\alpha'_{i_{high}} = \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}}(a_{i_{low}} - a'_{i_{low}}) \tag{5.11}$$

where $\eta = K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) + 2K(x_{i_{high}}, x_{i_{low}})$.

$$f'_i = f_i + (\alpha'_{i_{high}} - \alpha_{i_{high}}) y_{i_{high}} K(x_{i_{high}}, x_i) + (\alpha'_{i_{low}} - \alpha_{i_{low}}) y_{i_{low}} K(x_{i_{low}}, x_i) \tag{5.12}$$

They define index sets as:

$$I_{high} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \cup \{i : y_i < 0, \alpha_i = C\} \tag{5.13}$$

$$I_{low} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \cup \{i : y_i < 0, \alpha_i = 0\} \tag{5.14}$$

This set is computed using a tolerance value ϵ, due to the approximate nature of the problem (i.e. $\{i : \epsilon < \alpha_i < (C - \epsilon)\}$). The weights $\alpha_{i_{low}}$ and $\alpha_{i_{high}}$ are clipped to the valid range $0 \leq \alpha_i \leq C$ to ensure validity of the updates. The algorithm chooses indices of the weights to be updated (i_{low} and i_{high}) using a combination two sets of heuristics: a first order heuristic from Keerthi et al. [61] and second-order heuristic from Fan et al.[37]. This *adaptive* heuristic technique is as follows. First, using the first order heuristic, i_{high} is chosen as follows:

$$i_{high} = argmin\{f_i : i \in I_{high}\} \tag{5.15}$$

Then, to compute i_{low}, two set of values are computed for all $i \in i, ..., l$:

$$\beta_i = f_{i_{high}} - f_i \tag{5.16}$$

$$\eta_i = K(x_{i_{high}}, x_{i_{high}}) + K(x_i, x_i) + 2K(x_{i_{high}}, x_i) \tag{5.17}$$

$$\Delta F_i(\alpha) = \beta_i^2/\eta_i \tag{5.18}$$

Then i_{low} is selected based on the maximum change in the objective function ΔF_i over all points $i \in I_{low}$, for which $f_{i_{high}} < f_i$, i.e. there is progress toward the optimum.

$$i_{low} = argmax\{\Delta F_i \alpha : i \in I_{low}, f_{i_{high}} < f_i\} \tag{5.19}$$

After solving the optimization problem using the SMO algorithm and the set of heuristics described above, we end up with a set of training data points for which $\alpha_i > 0$; these points are called *support vectors*. The decision surface is defined by the support vectors. Once the set of support vectors is determined during training, the SVM classification problem is states as follows:

$$l_z = sign\{b + \sum_{i=1}^{l} y_i \alpha_i K(x_i, z)\} \tag{5.20}$$

where $z \in \mathcal{R}^n$ is a point in the target set for which need to determine the label $l_z \in \{1, -1\}$.

5.4.2 SVM component overview

Our goal now is to design and implement the SVM component such that it enables efficient execution of the SVM SMO training and classification algorithm on a set of parallel platforms while enabling application developer to use the component productively, in a high-level language. Similar to the GMM component, we use the SEJITS methodology to enable efficiency, productivity and portability. We encapsulate the computations for this specific instance of an application pattern in a template-based specializer using Asp. The API for this component has to correspond to the standard way application developers use SVMs. We can again refer to the example API in the Scikit-learn [96] library. Our SVM component must provide the following parameters:

- `kernel_type`: type of kernel to use in the SVM model,
- `kernel_parameters`: the r, d and γ kernel parameters specific to the kernel function,
- `heuristic_method`: heuristic to use (first, second or adaptive),
- `epsilon`: the ϵ approximation parameter,
- `tolerance`: the τ convergence parameter

and support the following functions:

- `train(feature_vectors, labels, kernel_type, kernel_parameters, heuristic_method, epsilon, tolerance)`: train the SVM using the specified kernel functions and other parameters,
- `classify(feature_vectors)`: classify the given set of test feature vectors, optionally pass in labels to evaluate accuracy,

- `get_support_vectors()`: return the support vectors for model analysis.

Application programmer needs to be able to call each function from Python code and customize the computations using the set of parameters. We aim to enable portability of our SVM component to a variety of GPU platforms and enable applications to run on any NVIDIA GPU without code change. We leave enabling the SVM component to target Cilk+ backend to future work.

5.4.3 SVM component implementation

We aim to enable efficient and productive use of SVM instance of linear classification pattern in applications that use our framework. Since the SVM training function is by far the most compute-intensive function of this component, we focus the majority of our attention on it.

SVM training on the GPU

As mentioned above, for the SVM training and classification implementation, we start with the efficient implementation from Catanzaro et al. [16] and modifying it to use more modern CUDA functionality. This code contains implementations of the SMO training algorithm using both types of heuristics as well as the SVM classification code.

The GPU implementation of SVM training we use for our SVM component uses several parallel computation phase. The parallel tasks are based on the MapReduce structural pattern. As discussed in Chapter 4, the MapReduce structural pattern consists of map and reduce phases, where map phase maps independent computations onto a set of data points and the reduce phase summarizes the results. First, one GPU thread is assigned one data point in the training set to compute f' (equation 5.12), using map parallelism. Then, depending on the heuristic chosen, it launches another set of parallel tasks on the GPU. For the first order heuristic, it launches a set of reductions to compute b_{low}, b_{high}, i_{low} and i_{high}. For the second order heuristic it launches two reductions to compute b_{high} and i_{high}, it then launches a map to compute ΔF_i for all points and reduce phases for computing b_{low} and i_{low}. The implementation also caches the values of $K(x_i, x_j)$ on the fly, if the computation encounters a repeated set of x_i and x_j, those values are simply retrieved from the cache and not recomputed. After analyzing this implementation, we carefully went through the code and updated it to use more recent CUDA constructs (for example, for using shared memory on the GPU).

To implement the SVM component, we use Asp templates, following the process we used to implement the GMM component. We translate all CUDA code files of the SVM training and classification implementation to `.mako` template

files. We replace the parameters that will need to be filled in by the specializer with template parameters: the number of thread blocks, the number of threads per thread block, and shared memory size.

When the SVM training or classification functions are called from application Python code, our component can pull in the template code and populate the template parameters with values based on the particular GPU platform the code is running on. We can do this the same way we did in the GMM component implementation: by querying the GPU for its specs and filling in the template parameters with the corresponding values. After we populate the template parameters with the correct values, Asp can generate valid CUDA code from the templates, call the appropriate compile toolchain (which can be specified in our component logic), execute the function and return results back to the Python application.

Using SVM component in Python code

Figure 5.4 shows a sample Python code snippet that uses the PyCASP SVM component. First, the Numpy library and the SVM components are imported into the application (line 25-26). Then the training and testing data represented by Numpy arrays are read in (lines 29 and 32). The SVM object is instantiated on line 35 and trained using the training data feature vectors and labels and a linear kernel (line 39). Finally, the trained SVM is used to classify the testing data feature vectors and results are evaluated for accuracy against the ground truth labels (line 43). This sample application is written in Python but allows the application programmer to use the underlying GPU hardware by automatically specializing the SVM training and classification function calls to the GPU using SEJITS. Because of our specialization mechanisms, this code can run on any CUDA-programmable GPU without code change. This example illustrates how the pattern-based design and SEJITS-based implementations help us achieve productivity, efficiency and portability goals. The specific results on sample applications are discussed in Chapter 8.

5.5 Summary

This chapter steps through the process of realizing PyCASP's pattern-oriented design in software. Our goal is to realize patterns in software in a way that offers productivity to the application developers and efficiency and portability to the applications. Application patterns provide a great API for PyCASP as they are familiar to application developers and capture concise algorithms for efficiency programmers to implement. We design PyCASP to support two kinds of components that capture two levels of customizability: library and customizable components.

```
25 import numpy as np
26 from svm import *
27
28 # get numpy array of training data
29 training_data = get_training_data()
30
31 # get numpy array of testing data
32 testing_data = get_testing_data()
33
34 # create SVM object
35 svm = SVM()
36
37 # train the SVM on training data (features and labels)
38 # using a linear kernel and default eplison and tolerance values
39 svm.train(training_data.features, training_data.labels, "linear")
40
41 # classify testing data
42 # evaluate against ground truth labels to compute accuracy
43 accuracy = svm.classify(testing_data.features, testing_data.labels)
```

Figure 5.4: Example usage of the SVM component.

Previous work showed that the SEJITS methodology can bring productivity, flexibility, efficiency and portability to application frameworks. Happily, a specific implementation of SEJITS, called Asp, is a methodology that allows patterns captured in terms of a DSL to be matched up with a highly-efficient implementation created by an efficiency programmer. Asp fits our goals well in several ways. First, Asp provides two main mechanisms for efficiency programmers to create software implementations of a pattern: templates and AST manipulations, which map well onto the two kinds of components of PyCASP. Second, the Asp prototype uses Python, which is a good language choice for us for other reasons discussed in the chapter. We describe two detailed Asp template-based specializers for two PyCASP components: Gaussian Mixture Model (GMM) training and evaluation and Support Vector Machine (SVM) training and classification. We illustrate that even though library components don't allow for as much flexibility as customizable components, they still provide quite a bit of customization to the application developer. For example, library components can support customizations in the form of algorithm selection and in some cases parameter selection for the specific code variants of the algorithm.

Chapter 6

Composition with Structural Patterns

6.1 Composition Design Space

In previous chapters, we described the pattern-based design of PyCASP and the implementation of application patterns using Selected Embedded JIT Specialization. In addition to using these techniques to create a productive, efficient and portable software framework, we set out to create a software environment to enable *application-specific optimization opportunities* to further improve application performance. Using the pattern-mining process we determined that the three structural patterns, Pipe-and-Filter, Iterator, and MapReduce, are sufficient to express the types of compositions of computations in audio content analysis applications. By restricting the scope of our software environment to a specific application domain, we aim to gain an understanding of how the structural patterns are used to compose computations and what optimization opportunities they present. *The process of identifying and implementing composition optimizations is application-domain specific* as we must understand the types of compositions, the computations that are being composed and the data format they use. We now ask: how can the structural pattern-based composition mechanisms be realized in software and how can we use these to optimize component composition in PyCASP?

One option is to realize the structural pattern composition in software *explicitly*. Explicit composition is defined by having the application programmers explicitly use structural pattern classes in their application code. We can use explicit skeletal class structures [29] for each type of structural pattern. By having explicit class structures for the structural patterns, we can explicitly define composition points and available optimizations for each pattern. For example, we can provide a skeleton for the Pipe-and-Filter structural pattern by having

an abstract class that defines "pipe" and "filter" functions, where the pipes define data sharing optimizations. These functions can then be defined and implemented by the application writer. This approach requires application writers to identify what types of structural patterns they need to use in their application and rewrite the application code to use that logic.

An alternative approach is to have the application developers write application code without the explicit need to identify the structural patterns in their applications. This is an *implicit* composition strategy. It is defined by having the framework *automatically* detect composition points and perform composition optimizations. There are *domain-specific* ways of composing computational components in applications. Specifically, since we focus on the audio content analysis applications, we can infer the typical ways computations are *naturally composed* in such applications. We can then enable PyCASP to recognize these typical composition points and optimize them. In order to determine the composition points, we need to study example use cases to determine how PyCASP's components are typically used and composed in the example applications.

Figure 6.1 schematically illustrates how components can be used in PyCASP. Application programmers can import and use PyCASP components as black boxes in their application code (example A on the left). In some cases, there are typical ways to use components, for example in an Iterator pattern (example B). Figure 6.2 shows a more concrete code example of implicit composition. The figure shows a GMM-SVM composition using the Pipe-and-Filter structural pattern. The output of the GMM training call (specifically the GMM mean parameters) are passed to the SVM classification component. The GMM means are reused across the two component calls, which presents an opportunity for optimization. Since this is a natural way for application programmers to use the GMM and SVM components in an application, we can add logic in PyCASP to enable optimizations when such compositions occur in applications. Since we use Python to implement PyCASP, we can implement the composition logic in Python to implicitly optimize applications that use specific structural patterns. We focus on enabling application programer to productively develop applications. Thus, we choose to use the implicit composition strategy since it provides a natural way to use components in application, enables composition optimizations, and does not require rewriting applications to explicitly use structural patterns. We leave the evaluation of explicit composition strategy as future work.

The three structural patterns are used on many levels of software architectures of applications - from high-level computation across different audio analysis models, to low-level implementations of specific machine learning computations. In this work we focus on *composition of PyCASP's components*, and thus, are concerned with the high-level composition of computational building blocks. The use of the structural patterns at the low-level is captured in the component implementations (i.e. the low-level code emitted by the component specializer). In this

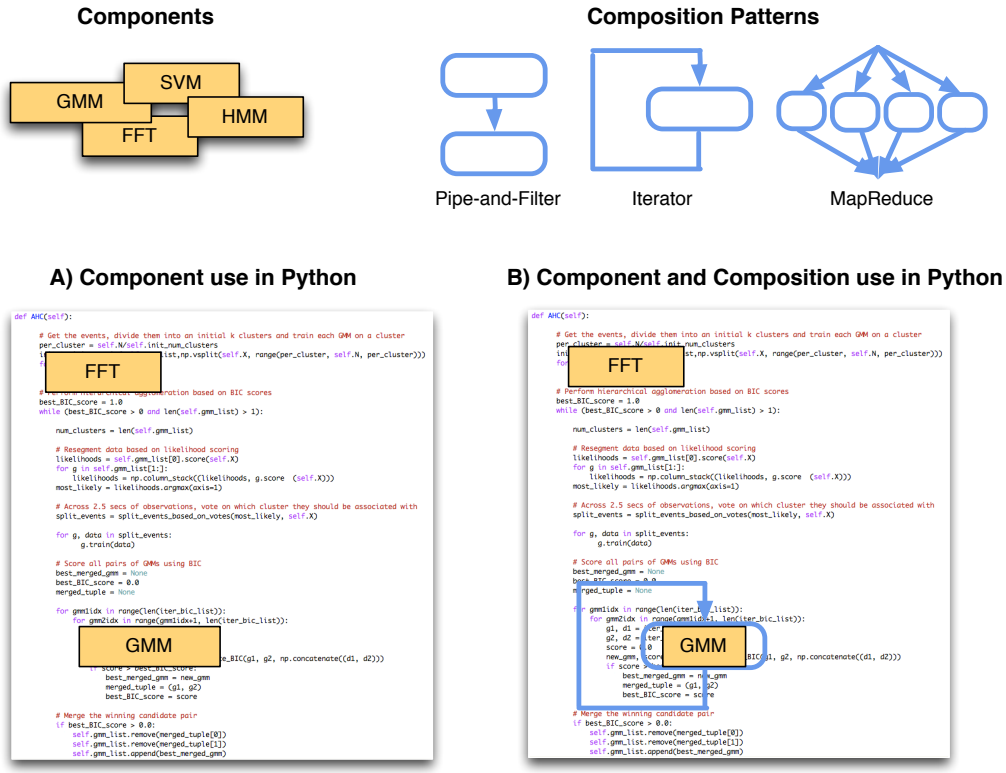

Figure 6.1: Schematic summary of components and composition patterns of PyCASP and sample usage in applications.

```
45 from pycasp import gmm
46 from pycasp import svm
47
48 input_data = get_input_data()
49
50 # Train
51 gmm = GMM(M, D)
52 gmm.train(input_data)
53
54 # Classify
55 class = SVM.classify(gmm.means)
```

Figure 6.2: Example implicit composition using Pipe-and-Filter.

work we are not concerned with composing computations with patterns at that low level. We also note that since we are focused on how PyCASP's components are composed at a high level, the underlying implementation of communication mechanisms (i.e. whether it is via message-passing or shared memory) is orthogonal to this analysis. Here, we are aiming to understand composition at the level of PyCASP components and implement composition optimizations based on the structural patterns.

We now analyze in detail how structural patterns are used to compose computations in applications. We start by describing each pattern and looking at the specific use cases and applications where the pattern in used to understand how the computations are naturally composed using the pattern. We then make decisions about what optimization opportunities are available for the specific pattern and how we can implement them.

6.2 Pipe-and-Filter

As described in Chapter 4, the Pipe-and-Filter pattern refers to applications that are structured with data flowing through modular phases of computation (filters) using data flow channels (pipes). This pattern is the most common structural pattern used in machine learning, and specifically, audio content analysis applications. These applications typically consist of a set of computations arranged in a particular, linear order. The structural control-flow of such applications uses the Pipe-and-Filter pattern to pass data and control from one phase of the computation to another; the control-flow is defined by the pipes of the Pipe-and-Filter pattern.

6.2.1 Composition using Pipe-and-Filter

We now look at specific instances of using the Pipe-and-Filter pattern to compose computations in audio content analysis applications. When learning optimal parameters of machine learning models in audio content analysis applications, application researchers typically need to iterate between training and testing of the model. First, they train a particular model instance on a set of training features. Then they evaluate the model performance on a set of testing examples and evaluate the model given the ground truth. If the model is not performing as well as needed, application researchers change certain parameters of the model that they think will improve performance, and repeat the training process until sufficient model prediction accuracy is achieved. This process heavily utilizes the Pipe-and-Filter pattern: the specific computations of the training and testing phases are the filters of the pattern and the data passed between computations flows through the pipes. We illustrate this approach with two concrete examples

of training machine learning models using the two PyCASP components described in Chapter 5: GMM training and prediction and SVM training and classification. We look at the details of using the Pipe-and-Filter structural pattern to compose computations in the model training examples.

Figure 6.3 shows a sample GMM training pipeline that can be used to learn a GMM-based acoustic model using short-term frequency Mel-Frequency Cepstral Coefficients (MFCC) features. In order to determine the right set of GMM parameters (the number of components, type of covariance matrix, etc.), the GMM training algorithm iterates between two phases - training and testing. In the training phase, the acoustic model represented by the GMM is trained on MFCC features. Then, using the MFCC features extracted from a *testing* dataset, the trained GMM is used to predict the likelihood of the features extracted from previously-unseen testing data. Finally, the model is evaluated against the ground truth to understand its predictive performance. As shown in Figure 6.3, computations in the training and testing phases are composed using the Pipe-and-Filter pattern. The filters are the computations of MFCC feature extraction and GMM training / prediction calls. The data flowing down the pipes of the Pipe-and-Filter are the MFCC features and the GMM parameter data.

One of the basic machine learning approaches used in audio analysis is to use GMMs for modeling acoustic properties of audio and use SVMs for classifying the content represented by the GMMs. The model learning procedure is employed in learning the GMM and SVM parameters in such systems. This approach can be used in many audio classification application such as speaker modeling and verification, speech recognition, music classification and audio concept detection.

We now look at the GMM-SVM training pipeline, shown in Figure 6.4. The SVM is trained on GMM means (referred to as supervectors). The SVM is tasked to classify GMM supervectors into two classes. The training data contains examples of audio from both classes and corresponding class labels. During the training phase, the MFCC features are extracted from raw audio and passed to the GMM training procedure. After training the GMM on each audio sample from both classes, GMM supervectors with the corresponding labels are passed to the SVM training computation. The SVM is trained using the two sets of features and labels. To evaluate the SVM, the support vectors (SVM parameters) are passed to the testing phase of the SVM learning process. A set of MFCC features is extracted from previously unseen testing examples and passed to the GMM training call to compute the supervector for each audio example. The trained SVM is then used to classify the new supervectors into one of the two classes. As shown in Figure 6.4, the Pipe-and-Filter pattern serves as the main structural pattern this application as well. The filters are the MFCC extraction, GMM training and SVM training and classification computations. The pipes are the data passed between these stages (MFCC features, GMM supervectors and SVM parameters).

The GMM-SVM model is also used in the speaker verification application,

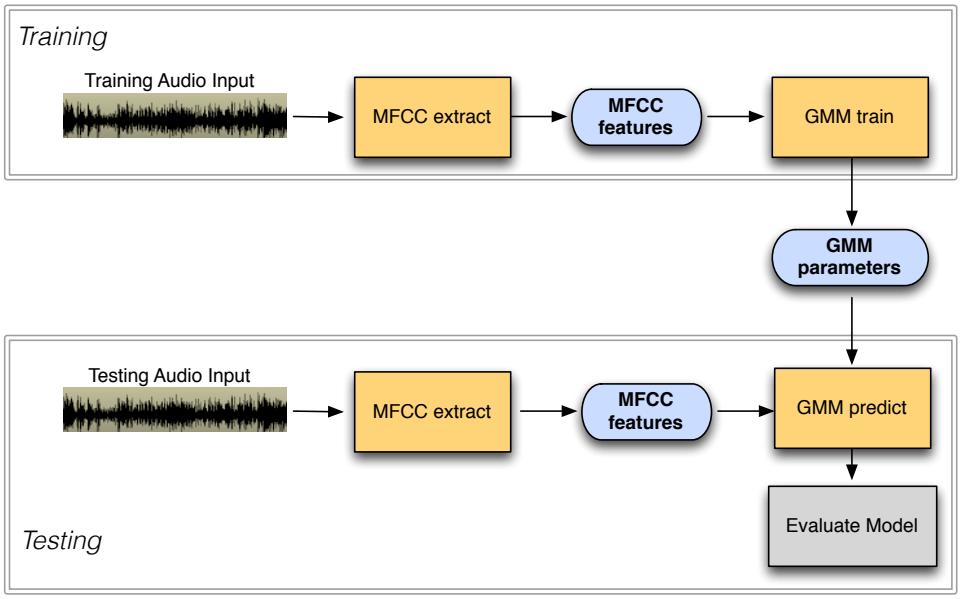

Figure 6.3: Composition using Pipe-and-Filter pattern for GMM training pipeline.

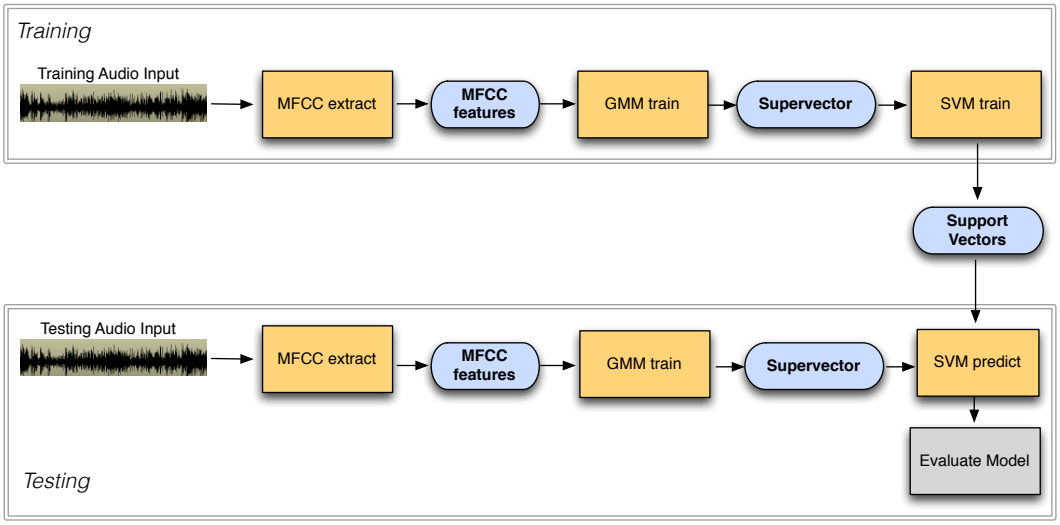

Figure 6.4: Composition using Pipe-and-Filter pattern for SVM training pipeline.

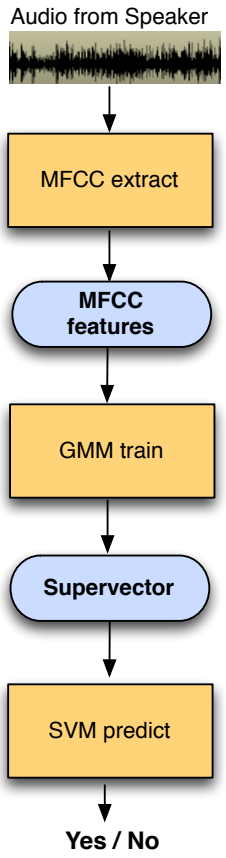

Figure 6.5: Composition using Pipe-and-Filter pattern for GMM-SVM speaker verification system.

whose algorithm we discussed in Chapter 3. In the speaker verification system, the SVM is used to classify previously unknown audio input to determine whether it belongs to a particular target speaker. Figure 6.5 shows the audio classification phase of the speaker verification application. Here, the Pipe-and-Filter pattern is also the main structural pattern used to compose computations. The MFCC features are extracted from raw audio recorded from the speaker, the GMM is trained on the features and then the GMM supervector is classified using the trained SVM. The filters are MFCC extraction, GMM training and SVM prediction computations and the pipes are the MFCC features and GMM supervectors passed between the computations.

Thus, during machine learning model training and evaluation as well as model prediction (exemplified by the speaker verification application), we discover the common ways that the Pipe-and-Filter structural pattern used to compose computations. The filters correspond to specific computations, in our case realized as PyCASP components. The pipes correspond to *specific data structures that are*

passed between the components. This gives us an insight into what types of compositions occur in our application domain and enables us to implement optimizations in our framework based on these insights.

6.2.2 Implementation of composition optimizations

To use the insights we gained by studying the Pipe-and-Filter-based compositions in audio content analysis applications, we need to determine application optimizations that are possible given these types of compositions. We have determined that composition using the Pipe-and-Filter pattern in our application domain corresponds to data structure sharing across computational components. We can use this knowledge to optimize applications that are developed using our software framework.

Data structure sharing

We can remove redundant data allocation, deallocation and deep copy calls when data structures are shared between computations. If one component uses the output of another component as its input, there is no need to deallocate and reallocate and copy that data. Thus, we need to implement a mechanism that simply passes the reference to already allocated data from one component to the next. For simplicity, we focus on the case of two components sharing one data structure; optimizing data structure sharing across several instances of computations becomes exponentially more difficult. When sharing a data structure across two computations, we must pay careful attention to the data format - both computations have to use the same format or know how to efficiently convert from one format to the other. This is the major contingency when composing applications using data structure sharing. Computational components need to agree on what data format they will use in a pre-defined "protocol". When new components are added to the framework, they must adhere to this protocol or define a new format if a new type of composition is introduced. This makes composition of computations a challenging task: there are numerous ways to represent data structures, for example graphs can be represented using adjacency matrix or adjacency lists, sparse matrix representations can also use a variety of formats. Each representation implies different level of efficiency of the algorithm, depending on the type of computation that is performed on it. Thus, we are faced with the question - how do we define a data structure format protocol for the components of our software framework?

To address the above question, we look back at the scope of our framework. Since each type of component can introduce their own data structure format, we focus on the exact computations that our framework aims to support. We use the design patterns to define and restrict the scope of our framework. This enables us

to reason through *specific computations and data structures* that are used in our application domain. We discover that in audio applications the data structures are used to represent:

- Audio input data, for example MFCC features,
- Alternative or transformed representations of audio, for example the DCT or FFT of the input data,
- Machine learning model parameters, for example GMM means and covariance matrices, and
- Auxiliary evaluation data, for example likelihood of an observation computed using a pre-trained GMM.

These data structures are usually represented using *dense matrices and vectors*. Thus, we restrict our composition problem to the following:

- Data structures that are shared across components are represented using dense matrix and vectors,
- We optimize one-to-one component data sharing,
- We know exactly the types of computations that will be composed (given by application patterns).

By focusing on a more restricted composition problem, we hope to enable component composition and cross-computation optimizations in our applications.

Implementation details

We focus on the restricted set of compositions given our application domain - allowing for efficient sharing of dense matrix and vector data structures across computational components of PyCASP. As described in Chapter 5, we use the Asp framework to develop specializers for each specific instance of application patterns. When a component is called for the first time, it allocates the data structures it needs to perform its computations. However, if a component operates on data that has previously been allocated by another component, instead of reallocating the data structures in this component call, we can pass the *reference* to the data structure.

The components must agree on the data format of the shared data structures - in our case, the data is dense matrix and vectors, thus the components must agree on whether the data is stored in row-major or column-major format. By default, each component in PyCASP assumes the data is in row-major format. A

component is allowed to internally transpose data to make specific computations more efficient (for example the GMM component stores two copies of the input data, one in row-major and one in column-major format). The conversion between formats is left to the component writer (i.e the SEJITS specializer developer).

In Python, objects are passed by reference by default - when data structures (for example Numpy arrays or Python dictionaries) are passed from one computation to the other, they are passed the value of the object's reference, without reallocation. It is assumed that the programmer knows this default feature and expects each function to potentially modify the data structures passed to it as parameters (unless they are of immutable type, such as numbers or strings). Thus, when running Python applications on CPU platforms, Python by default already implements the data structure sharing optimization.

As mentioned in Chapter 2, when working with the GPU backends, data allocation and memory management must be handled explicitly by the programmer. Each data structure needs to be explicitly allocated, copied and deallocated in CUDA code. Thus, we must explicitly implement the pass-by-reference mechanism for the GPU backend of our PyCASP components. For example, Figure 6.6 shows the two alternative data sharing implementations in the speaker verification application. In the first scenario (Figure 6.6(A)), without implementing the composition optimization, the supervector data will be deallocated, copied back to the CPU, reallocated, and copied back to the GPU. This is clearly wasteful. Instead, by recognizing that data is shared across these two component calls, PyCASP can remove the redundant data reallocation calls and pass the reference to an already allocated data structure to the SVM component (shown in the second scenario, Figure 6.6(B)).

The pass-by-reference mechanism can be implemented by creating a Python dictionary that maps data structures to their locations in GPU memory. When a component invokes a data allocation call, PyCASP first checks if the data has been allocated on the GPU by referencing this memory map data structure. If there is a "hit", PyCASP passes the reference to the data in memory and skips the allocation call. This way, data structures can be shared across GPU function calls. We use the data structure map to keep track of which data structures have been allocated on the GPU. Specifically, since Python already uses references to pass data structures between function calls, we store the pair `[CPU reference, GPU reference]` in our dictionary. The next task is to insert the read and write calls to this data structure in the PyCASP component implementation logic. Every time a component calls a GPU allocation function, we look up the CPU reference of that data in the dictionary. If we don't find the reference in the dictionary, we allocate and copy the data to the GPU and store the new pair `[CPU reference, GPU reference]` in the dictionary. If we do get a hit in our dictionary, we simply set the GPU data reference in the component to the value returned by the dictionary and skip the allocation and copy calls.

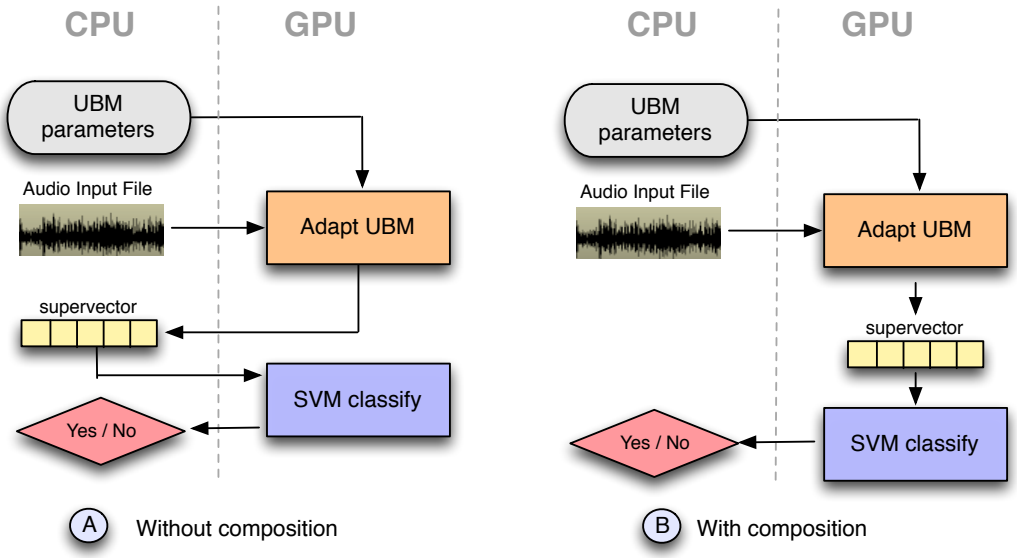

Figure 6.6: Two alternative data sharing implementations in the speaker verification system.

When the PyCASP component objects are no longer used by the application, their destructors are automatically invoked by Python's garbage collector. Thus, we add deallocation calls to the destructors of our components that deallocate data structures on the GPU. This way, when the same object is reused many times in an application, its destructor will not be called, and thus, the data structures will stay in the GPU memory. When the destructor is finally called, the GPU data structures are deallocated and we remove the corresponding entry from the memory map dictionary. Using Python objects allows us to add the memory management logic to the objects constructors and destructors and thus, implicitly control data structure allocation in the PyCASP components.

Now that we have determined our optimization strategy and its implementation, we mark the specific locations to insert these optimizations in our component's code, i.e. the specific functions that will require these optimizations. Referring back to Figures 6.3, 6.4 and 6.5, we identified the set of data structure transfers between specific component calls. They are:

- MFCC-extract - **MFCC features** - GMM train
- MFCC-extract - **MFCC features** - GMM predict
- GMM train - **GMM parameters** - GMM predict
- GMM train - **GMM supervector** - SVM train

- GMM train - **GMM supervector** - SVM predict
- SVM train - **SVM parameters** - SVM predict

Thus, we need to insert the memory optimizations in the following function calls:

- Feature extractor data allocator
- GMM training input data allocator
- GMM prediction input data allocator
- GMM training parameter allocator
- SVM training input data allocator
- SVM prediction input data allocator
- SVM training parameter allocator

We insert the dictionary write and read logic into these functions in the socializer Python code. This process then becomes a way to tie together the sets of components that we know are composed into applications using the Pipe-and-Filter pattern specializer. We discuss the performance results of these optimizations in Chapter 8.

6.3 Iterator

As we discussed in Chapter 4, the Iterator pattern refers to applications whose computation is repeated many times in a loop until a termination condition is met. The Iterator pattern is used extensively in audio content analysis, specifically when learning model parameters. During speaker diarization, for example, the learning algorithm iterates through each speaker model, updating its parameters based on likelihood of the meeting segmentation. During acoustic or classification model training, model parameters are updated until a particular error function is minimized (as in SVM training) or a certain number of iterations is reached (as in GMM training).

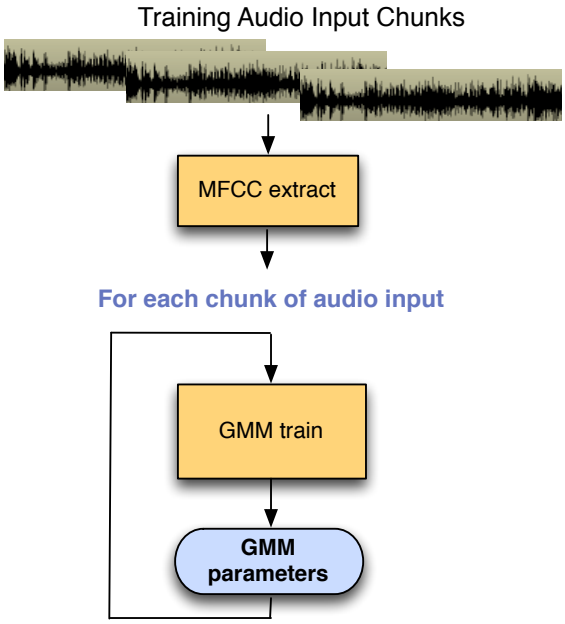

Figure 6.7: Composition using Iterator pattern in UBM training application.

6.3.1 Composition using Iterator

We now look at specific instances of composition of computational building blocks using the Iterator pattern. Figure 6.7 shows the Iterator pattern used for training a UBM model. UBM is represented by a GMM and is trained on a set of features randomly sampled from the problem dataset. We divide the dataset into chunks of audio and train the GMM on the set of features in each chunk, updating the GMM parameters in each training call. The GMM parameters are *shared across* the calls of the GMM training function. We can optimize this composition by recognizing that the same model is trained on different chunks of audio features and the model parameters are *reused* across training calls.

Figure 6.8 shows how the Iterator pattern is used in the speaker diarization application to segment a meeting recording. At the top level, the algorithm iterates over the speaker model training and data segmentation phases, merging speaker models (represented by GMMs) until the optimal BIC score is reached (see Chapter 3 for the algorithm details). In each iteration, each speaker model is retrained on data that has been previously assigned to it during the segmentation phase. Then each speaker model is evaluated using the new trained parameters to compute the likelihood of each feature in the meeting recording. After the likelihoods are computed for each model, they are passed to the segmentation phase to re-assign chunks of audio to the speaker models.

In this Iterator composition, each speaker model is retrained, using a sepa-

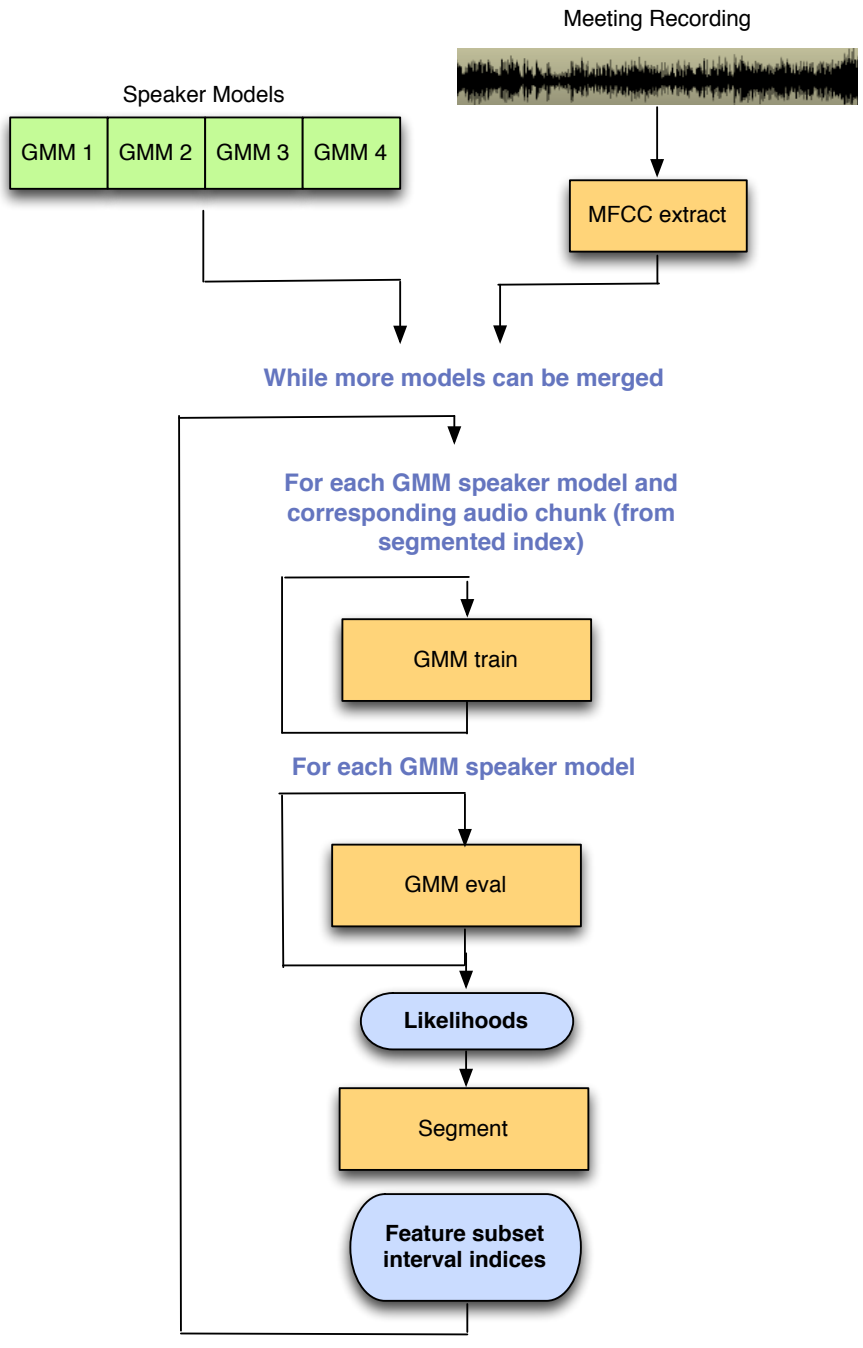

Figure 6.8: Composition using Iterator pattern in speaker diarization application.

rate call to the GMM training function. Thus, we are forced to reallocate the GMM parameters each time when the GMM training function is invoked. We can, however, optimize the way data is passed to the GMM training call. Since the meeting audio data is reused throughout the segmentation process, we can allocate it once for the entire speaker diarization application. In addition, after segmentation, instead of passing subsets of feature vectors to each GMM training call, we can instead pass the *indices* of the feature subset intervals, reducing the amount of data that is passed between function calls.

Note that the overall structure of this application uses the Pipe-and-Filter pattern - the GMMs are trained and evaluated and then the audio is segmented using the output of the GMM evaluation computation. Accessing the input data using the interval indices can be done at the Pipe-and-Filter level - i.e. we can always access the meeting data using the intervals instead of passing in audio features directly. However, since in the Iterator pattern the audio data is reused repeatedly across component calls, this optimization becomes much more important.

As shown by the examples, the Iterator pattern is used to compose computations that are repeated many times. The opportunity for optimization is presented by the *data structure reuse* across component calls. As shown in our example applications, these data structures can be model parameters that are updated over several training calls, or read-only audio data that is reused across multiple model training calls. We now discuss the implementation details of how we can use these insights and implement optimizations for compositions that use the Iterator pattern.

6.3.2 Implementation of composition optimizations

To implement the optimizations for composition of computations using the Iterator pattern, we use a *lazy allocation* technique to remove redundant data allocation and copy calls: for both CPU and GPU backends, we skip the data structure allocations if the data is reused from a previous object allocation. Python object allocation on CPU is automatically managed by Python. Python extensions such as PyUBLAS [85] also allow data structures to be shared by reference between C++ and Python. Thus, similar to the Pipe-and-Filter pattern, we focus on the CPU-GPU implementations, as we must manage data allocation logic manually. Instead of reallocating the GMM parameters for every call to the GMM training function, we can lazily reallocate GMM parameters by recognizing when the same GMM object is reused across multiple training calls. If the same GMM is trained using new input training data, we do not reallocate the GMM parameters but reuse the ones that are already in memory by passing in a reference to the parameters.

To implement the feature interval indices optimization, we implement another GMM training function variant that operates on interval indices instead of feature

vector training data as input. This function (`train_on_subset()`) takes a set of index intervals as parameters and gathers the subsets of audio features corresponding to the intervals into a buffer that is then passed to the training function. The meeting audio data resides in main GPU memory and is not reallocated across GMM training calls (unless, of course, the application programmer changes the input data to be a new set of audio features). We discuss the trade-offs of the results of these optimizations in Chapter 8.

6.4 MapReduce

The MapReduce pattern (as discussed in Chapter 4) describes programs that implement specific map and reduce operations on key-value pairs to enable application parallelism. Using the two simple data operators, many loosely coupled tasks can be readily mapped to clusters [87]. MapReduce jobs are typically used to distribute a computation across a set of nodes in a cluster. A programmer can write arbitrarily complex applications using a set of map and reduce functions to extract information and analyze large datasets. The computation model is based on streaming - data flows through the map and reduce stages, each stage processing the data, performing computation, processing or summarizing tasks. The output of a MapReduce job is typically an analysis result or a summary of the input data.

MapReduce is also used extensively on the lower-levels of application software architectures. For example, in GMM EM training algorithm, MapReduce is used to map computations to the GPU cores. However, as mentioned before, we are concerned with the composition of PyCASP's components and thus, analyze the high-level use of the MapReduce pattern.

6.4.1 Composition using MapReduce

In audio analysis applications, the MapReduce pattern is typically used when applications process large datasets. By using the MapReduce pattern and distributing the computation across many nodes of a computer cluster, large datasets can be processed much more efficiently. The simplest MapReduce job consists of launching a set of map tasks, each executing a particular set of computations on a subset of data and there are no dependencies among the tasks. This is the simplest use of MapReduce yet it is powerful enough to provide scalability to applications that process large datasets. There has been extensive research done on implementing MapReduce frameworks and optimizing more complex MapReduce applications [51, 76, 82, 112]. In this work, we focus on integrating the MapReduce functionality into PyCASP to enable application writers to easily scale their applications to clusters of machines.

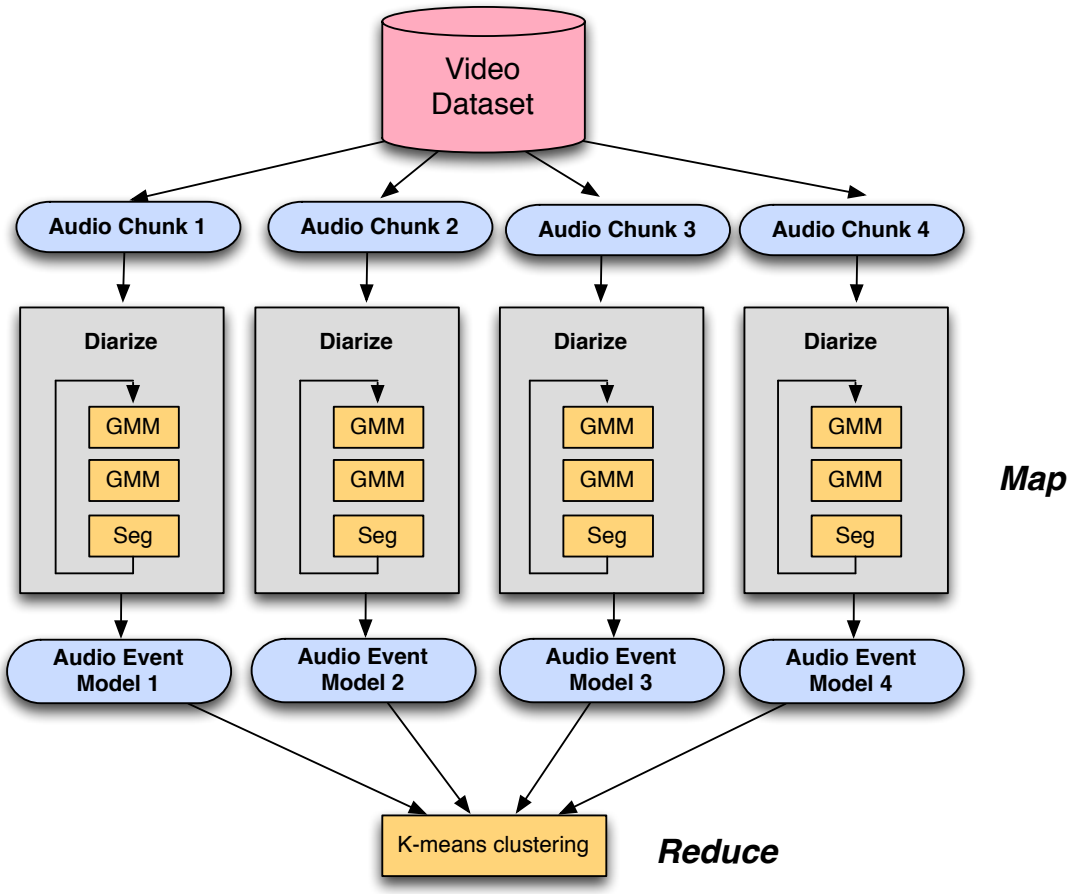

Figure 6.9: Composition using MapReduce pattern in video event detection application.

We use the video event detection application as the driving example for the MapReduce-based composition. The application processes thousands of videos, extracting information from the soundtrack of each video. MapReduce is required to make this application scalable to large datasets. Figure 6.9 shows the use of MapReduce pattern in the video event detection application. The video dataset is partitioned across a set of nodes in a cluster, each cluster node is assigned a particular subset of videos. Each node performs the diarization computations on its assigned subset of data to segment the video soundtrack and compute event models (see Chapter 3 for algorithm details). The result is then returned to one node, which runs the K-means clustering to distill the global set of audio events present in all videos. Since diarization of each video is independent, we are able to process the entire video dataset in parallel using the MapReduce pattern.

The computations in each map function can themselves consist of multiple

compute stages, composed using Pipe-and-Filter and Iterator patterns. We can use the optimizations for each pattern described earlier in this chapter. Thus, we can then *compose MapReduce workloads with Pipe-and-Filter and Iterator workloads* in applications, each of which uses a set of PyCASP components and composition optimizations. For example, in the video event detection application, we compose the MapReduce pattern with the diarization computation, which itself consists of an Iterator and Pipe-and-Filter patterns and uses the segmentation and data sharing optimizations described earlier. The component computations are run on a multi-core CPU or a GPU, utilizing parallel hardware on the compute node. Thus, we use our optimization strategies for composition of components across several levels of structural patterns. This enables full-functioning, optimized applications that scale to large datasets and utilize several levels of parallel hardware.

6.4.2 Implementation of composition optimizations

From our analysis above, an application of the MapReduce pattern is to distribute data and independent computations across cluster nodes. In this work we choose to focus on this basic use of MapReduce - enabling functionality to distribute data to cluster nodes and exposing the map and reduce functions in PyCASP.

To expose the MapReduce functionality in PyCASP, we use the popular, open source Hadoop MapReduce framework [107]. Specifically we choose to use the `mrjob` abstraction for writing MapReduce programs in Python [1]. The application programmers can then use the map and reduce functions in Python to implement their applications and scale them to computer clusters. Map and reduce functions take arbitrary Python functions and thus can be passed complex functions that use other PyCASP components. This enables composition of PyCASP components and other composition optimizations with the MapReduce pattern. The programmer is tasked with transferring the data to the Hadoop distributed file system (HDFS) before invoking the application code. This is the standard practice in using MapReduce frameworks: the data needs to be transferred to the distributed file system to enable locality in MapReduce computations. The data transfer can be automated by the framework with specific input from the application developer; we leave this as future work. We discuss the specific examples of using the MapReduce pattern in applications in Chapter 7.

Our choice of MapReduce and Hadoop as the means of parallelization has several added benefits: Hadoop has automatic load balancing and fault tolerance, and it provides an array of tuning parameters that PyCASP can use to optimize performance. Furthermore, it comes bundled with a distributed filesystem

[1] http://packages.python.org/mrjob/

that provides high read bandwidth, data durability, and the ability to schedule computation close to its data.

6.5 Summary

In this chapter, we have determined the set of composition points based on three structural patterns: Pipe-and-Filter, Iterator and MapReduce. We analyzed specific use cases of each pattern in our sample applications and we have determined the most natural way of composing components in audio analysis applications and available optimizations for the composition points. Table 6.1 summarizes our findings.

Pattern	Optimization
Pipe-and-Filter	Data structures shared between components
Iterator	Data structures reused in a loop
MapReduce	Data structures distributed across compute nodes

Table 6.1: Summary of composition optimizations based on structural patterns.

After identifying the specifics of composition mechanisms for each of the structural patterns, we determined how we can implement optimizations based on these insights. For data structure sharing and reuse, we use a reference-tracking mechanism to remove redundant allocation and copy calls. Since this functionality is already supported by default in Python when using CPU platforms, we focus our attention on the GPU backend, where we must explicitly manage memory use and data structure allocations. For data structure sharing using Pipe-and-Filter and reuse using Iterator, we implement a pass-by-reference mechanism and lazy allocation respectively to remove redundant data allocation on the GPU. For MapReduce we task the application programmer to distribute the data across cluster nodes before invoking the application code. For each pattern, we study each composition point explicitly and insert the optimization logic into each relevant function. We discuss the results of these optimizations in Chapter 8 and discuss advantages and disadvantages of our approach in Chapter 9

Chapter 7

Implementing Applications with PyCASP

Chapters 4, 5 and 6 described the pattern-oriented design of PyCASP, the implementation of the specific instances of application patterns using Selected Embedded JIT Specialization (SEJITS), and the composition methodology using structural patterns. In Chapter 3, we discussed the algorithmic details of the four sample applications that we use as application case studies in this work. We now tie things together by discussing how PyCASP can be used to implement these sample applications. We discuss how PyCASP components are used and how the composition mechanisms based on structural patterns are utilized to implement the four full-functioning audio content analysis applications.

Figure 6.1 in Chapter 6 summarizes the pattern-based structure of PyCASP and schematically shows the use of PyCASP in Python applications. We have designed PyCASP to contain a set of components that implement specific instances of application patterns (top left of Figure 6.1) and employ a set of optimizations for composing the components based on the three structural patterns (top right of Figure 6.1). The target audience of PyCASP are the audio analysis application developers and domain experts. To enable programmer productivity and application efficiency, we implement PyCASP in Python and use SEJITS to implement the PyCASP components. Thus, in order to use PyCASP in an application, the application developer can import PyCASP's components directly into the application code, instantiate the objects and then call the specialized functions, as shown in the schematic code example A in Figure 6.1. Importing PyCASP's component can be done by using the standard Python `import()` call to import the specialized component objects into the application. Because of our design decision to implicitly handle composition, when there is a particular structural pattern employed in an application (as shown in schematic code example B in Figure 6.1), PyCASP *automatically* identifies the composition points and uses its internal knowledge of data structure sharing and reuse to invoke the optimizations corresponding to the

particular structural pattern.

Using the SEJITS technology, after the application Python code is written, the application writer can run the code as a regular Python program. When the specialized component functions are called, PyCASP triggers the Asp SEJITS mechanisms and generates, compiles, links, caches, and executes the code. Thus, programmers are tasked with identifying the components of PyCASP that they can use in their applications, and PyCASP generates parallel optimized versions of the components as well as identifies composition points and performs further optimizations of the application by removing redundant data allocation and copy calls. This enables the application programmer to stay productive and focus on the algorithmic details of the application. PyCASP handles the rest of the implementation stack, i.e. the parallelization strategy, efficiency, code generation, and composition. We now describe the detailed implementations of the four sample applications using PyCASP.

7.1 Speaker Verification

The speaker verification system determines whether a piece of recorded audio belongs to a particular target speaker. Figure 7.1 shows the Python code of the classification step of the speaker verification system. The PyCASP components, which are executed on parallel hardware, are highlighted in light grey. The algorithm for this application is described in Chapter 3 and the software architecture is shown in Figure 4.2 in Chapter 4. To implement this application in Python, we use PyCASP's GMM component for adapting the UBM (i.e. performing EM iterations) and the SVM linear classification component to classify whether the supervector of the adapted UBM belongs to the target speaker. We now step through the code for the classification phase of the application shown in Figure 7.1.

1. **Initialize**. First, we import the two PyCASP components, GMM and SVM (lines 57 and 58). The function `verify_speaker()` is passed the parameters of the speaker models (M - the number of components and D - the dimensionality of the data), the UBM and SVM models trained in the training phase, and the audio data that needs to be classified.

2. **Adapt UBM**. We first create a new GMM object (line 62) and train it using the UBM parameters on the audio data (line 63), to adapt the UBM to the new speaker data.

3. **Classify using SVM**. We use the SVM to classify the GMM mean vectors (i.e. the supervector) to one of the two classes (target speaker or other speaker) and return the corresponding label.

```
57 from pycasp import gmm
58 from pycasp import svm
59
60 def verify_speaker(M, D, UBM, SVM, speaker_data):
61   # Adapt UBM
62   gmm = GMM(M, D, UBM.params)
63   gmm.train(speaker_data)
64
65   # Classify
66   class = SVM.classify(gmm.means)
67
68   return class
69
70 # Main Speaker Verification Code
71
72 M = 256    # number of components in GMM
73 D = 19     # dimensionality of GMM
74
75 UBM, SVM = train_speaker_verifier(M, D)    # code omitted
76 new_data = get_new_speaker_data()  # read in new speaker data to classify
77 decision = verify_speaker(M, D, UBM, SVM, new_data)
```

Figure 7.1: Speaker verification in Python. Components that are executed on the GPU are highlighted in light-gray. Code for the training phase omitted.

Lines 72 - 77 show the use of the `verify_speaker()` function, we set the parameters of the speaker model, M and D (lines 72 and 73), and train the UBM and SVM models (details omitted for brevity) on line 75. We then read in the new speaker data that needs to be classified and pass it to the `verify_speaker()` function for classification. When the GMM training and SVM classification functions are called, PyCASP invokes the specialization pipeline for each component. It calls the code generation logic, fills in the template values for each function, compiles, links and executes the function on underlying parallel hardware (either multi-core CPU or GPU), all transparent to the application writer.

In Chapter 6, we analyzed this application when studying the Pipe-and-Filter composition pattern. Without any additional logic, the result of the UBM adaptation (i.e. the supervector) would be copied back to the CPU after the GMM training computation was complete (line 63) and that memory on the GPU would be deallocated. After the copy back and deallocation, the SVM component would immediately reallocate the space in the GPU memory for its input data (i.e. the same supervector) and copy its values from the CPU to the GPU (line 66). This data structure reallocation is redundant, since the output of one operation is the immediate input of the other. Instead, using the logic and implementation of im-

plicit composition optimizations described in Chapter 6, PyCASP identifies this particular composition pattern and uses the data structure reuse logic to keep the data on the GPU. This is done by reassigning the reference to the input data for the SVM component to point to the output of the GMM component. This allows us to bypass the copy/deallocate/allocate/copy calls and simply continue onto the SVM computation.

This sample application illustrates how we can implement a sample speaker verification system in Python code. We can use the PyCASP components to automatically utilize parallel hardware and use composition optimizations to remove redundant data reallocation calls.

7.2 Speaker Diarization

The speaker diarization application segments a recording of a meeting into speaker-homogenious regions to determine what speaker spoke when in the meeting. Figures 7.2 and 7.3, show the implementation of the speaker diarization system in Python. As in previous example of speaker verification, the PyCASP components that are executed on the parallel platform (either GPU or multi-core CPU) are highlighted in light-gray.

Based on the algorithm description from Chapter 3 we now step through the Python code shown in Figure 7.2 and 7.3:

1. **Initialize**. First we import the Numpy [5] library and the GMM component of PyCASP (lines 79 and 80). We then uniformly initialize a list of K GMMs (in our case 16 5-component GMMs) on line 90. After creating the list of GMMs, we perform initial training on equal subsets of feature vectors (lines 91-92). The training computation is executed on the underlying parallel platform. Next, we implement the agglomerative clustering loop based on the Bayesian Information Criterion (BIC) score [91] (line 95-96).

2. **Re-segment**. In each iteration of the agglomeration, we compute the likelihood of the feature vectors given the current GMM parameters and re-segment the feature vectors into subsets using majority vote segmentation (lines 100-103). PyCASP GMM component automatically use the underlying parallel platform to compute the log-likelihoods (`gmm.score()` method), which calls the E-step of the GMM training algorithm.

3. **Re-train**. After re-segmentation we re-train the Gaussian Mixtures on the parallel platform on the corresponding subsets of frames (lines 105-106).

4. **Agglomerate**. After re-training, we decide which GMMs to merge by first computing the unscented-transform based KL-divergence of all GMMs (line

```
79 import numpy as np
80 from gmm import *
81
82 # Main diarization function. Parameters:
83 # M: number of GMM components, D: data dimensionality,
84 # K: starting number of segments
85
86 def diarize(self, M, D, K, data):
87     gmm_list = new_gmm_list(M,D,K)
88     N = data.shape[0]
89     per_cluster = N/K
90     init = uniform_init(gmm_list, data, per_cluster, N)
91     for gmm, data in init:
92         gmm.train(data)
93
94     # Perform hierarchical agglomeration
95     best_BIC_score = 1.0
96     while (best_BIC_score > 0 and len(gmm_list) > 1):
97
98       # Resegment data based on likelihood scoring
99       L = gmm_list[0].score(data)
100      for gmm in gmm_list[1:]:
101          L = np.column_stack((L, gmm.score(data)))
102      most_likely = L.argmax()
103      split_data = split_obs_data_L(most_likely, data)
104
105      for gmm, data in split_data:          # retrain on new segments
106          gmm.train(data)
```

Figure 7.2: Speaker diarization in Python part 1. Components that are executed on the parallel platform are highlighted in light-gray

114). We then compute the BIC score of the top k pairs of GMMs (in our case $k = 3$) by re-training merged GMMs as described in Chapter 3 (lines 116-120) and keeping track of the highest BIC score (lines 121-124). Finally we merge two GMMs with the highest BIC score (lines 127-128) and repeat the iteration until no more GMMs can be merged. The result of running the application are the meeting segmentation and the speaker models represented by the GMMs.

The code for speaker diarization shows how we can capture the core algorithm in under 100 lines of Python code. PyCASP components automatically utilize underlying parallel hardware by using the SEJITS methodology. In this application, the GMM training and likelihood computations are performed on parallel hardware, with parallel code automatically generated by PyCASP and Asp. Using

```
107     # Score all pairs of GMMs using BIC
108     best_merged_gmm = None
109     best_BIC_score = 0.0
110     m_pair = None
111
112     # Find most likely merge candidates using KL
113     gmm_pairs = get_top_K_GMMs(gmm_list, 3)
114
115     for pair in gmm_pairs:
116         gmm1, d1 = pair[0] #get gmm1 and its data
117         gmm2, d2 = pair[1] # get gmm2 and its data
118         new_gmm, score =
119             compute_BIC(gmm1, gmm2, concat((d1, d2)))
120         if score > best_BIC_score:
121           best_merged_gmm = new_gmm
122           m_pair = (gmm1, gmm2)
123           best_BIC_score = score
124
125     # Merge the winning candidate pair
126     if best_BIC_score > 0.0:
127       merge_gmms(gmm_list, m_pair[0], m_pair[1])
```

Figure 7.3: Speaker diarization in Python, part 2. Components that are executed on the parallel platform are highlighted in light-gray

the structural-pattern-based composition, PyCASP can further optimize this application by recognizing the opportunity for data structure reuse across iterations based on the Iterator pattern. Namely, as we discussed in Chapter 6, this application reuses the meeting audio across many iterations. Thus, instead of reallocating the meeting data for every call to the GMM training and prediction function, we can use the lazy allocation optimization to remove redundant data allocation and copy calls. In addition, we can use the interval indices for meeting segmentation to pass the segmentation information to the GMM training call. To switch to interval-indices-based implementation, we can replace the GMM training calls in lines 91-92 and 105-106 with the call to `train_on_subset(data, intervals)` that takes the reference to the meeting recording data (passed by reference using PyCASP's internal reference-tracking dictionary), and the intervals corresponding to the particular speaker model we're re-training. Given the interval indices, we can implement the logic to gather corresponding feature vectors either in Python or in CUDA. We discuss the results of these optimizations and implementation decisions in Chapter 8

7.3 Music Recommendation

The music recommendation system suggests musically-similar songs based on a query set of songs. Based on the algorithm described in Chapter 3, we now describe the usage of PyCASP in the application. First, in the offline data preparation phase, we train the UBM, adapt it on each song's data, one by one, and create the hash table of nearest neighbors. In the online phase, we receive a query from the user, adapt the UBM to the features of the songs returned by the query and find the list of closest songs.

Figure 7.4 shows the Python code for the relevant steps of the offline data preparation and online recommendation phase of the music recommendation system. We describe the use of PyCASP GMM component in the application and discuss the implementation details in the next two subsections.

Offline Data Preparation Phase

1. **Train UBM**. We use the GMM training component of PyCASP to train the UBM. We randomly sample the features of all songs in the database and train the UBM on 7 million timbre feature vectors. To train the UBM, we setup a GMM object and invoke the train() method on it. The Python logic for this step is captured by the function `train_music_UBM()` on lines 157-168 in Figure 7.4

2. **Adapt the UBM to all songs**. To adapt the UBM we also use the GMM training component (not shown in the sample code), but set the number of EM iterations to 1. For every song in the database, we retrieve the features and call the GMM training component to adapt the UBM to each song using the features.

3. **Hash song supervectors**. This step (also, skipped in the code snippet for brevity) does not use PyCASP but is implemented in Python using the Numpy library.

Online Recommendation Phase

1. **Get the query from the user**. We get the query from the user (in our case, all songs by Elton John). This step requires parsing the user query and obtaining the song features from the SQLite database using Python tools (details implemented in the function on line 134). No specialization is used in this step.

2. **Adapt the UBM to the query**. After parsing the query, we retrieve the feature vectors of all the songs that match the query and store them in a

```
128 from pycasp import gmm
129 import sqlite3 as sqlite
130
131 def recommend_songs(M, D, UBM, artist):
132   # get query song data and features
133   query_songs = DB.get_song_data(artist)
134
135   features = []
136   for song in query_songs:
137     features.append(song.features)
138
139   # adapt UBM
140   gmm = GMM(M, D, UBM.params)
141   gmm.train(features)
142
143   # get a list of near neighbors from the hash table
144   nn = hash_table.get_near_neighbors(artist, title)
145
146   # compute the distance between each neighbor
147   distances = []
148   for neighbor in nn:
149     song_info = db.get_song_data(neighbor.artist, neighbor.title)
150     distances.append(compbute_distance(song_info.sv, gmm.means))
151
152    # find the 10 closes songs
153    indices = argsort(distances)
154    return nn[indices[:10]]
155
156 def train_music_UBM(M, D, DB):
157   # get sets of features, randomly sampled from the DB
158   feature_sets = get_feature_sets(DB)
159
160   # create a UBM
161   ubm = GMM(M, D)
162
163   # train UBM on subsets of features
164   for fs in features_sets:
165     ubm.train(fs.data)
166
167   return ubm
168
169 M = 64
170 D = 12
171
172 UBM = train_music_UBM(M, D, DB)
173 closest_songs = recommend_songs(M, D, UBM, "Elton␣John")
```

Figure 7.4: Python code for the online phase of the music recommendation application.

list (lines 136-138). We then use the PyCASP GMM training component to adapt the UBM to the query features by doing one EM iteration of the train() function (lines 141-142).

3. **Get approximate nearest-neighbors for the query supervector**. We retrieve the approximate nearest neighbors from the hash tables (line 145). This is done in Python using the Numpy library.

4. **Compute the closest C songs**. We then compute the distance between the query and all the nearest neighbors returned in the previous step and return top C (in our example $C = 10$) songs. This is done in Python using the Numpy library (lines 148-155).

The main code invokes the UBM training and calls a sample query for "Elton John" songs. We are able to rapidly prototype the entire system in about 400 lines of Python code (excluding the SQLite database and LSH setup) and use PyCASP to remove the UBM training bottleneck by automatically offloading the computation onto underlying parallel processors using the GMM component of PyCASP. We use the composition optimization based on the Iterator structural pattern when training the UBM. Instead of reallocating the UBM parameters every time the GMM training function gets called (line 166 in Figure 7.4), we keep the UBM components on the GPU and iterate through chunks of feature vectors randomly sampled from our entire database of songs. This removes redundant data allocation and copy calls and improves the performance of the offline phase of the system. We discuss the performance results of this application in the next chapter.

7.4 Video Event Detection

The video event detection application uses the speaker diarization algorithm (and thus the GMM training component of PyCASP) to segment video soundtracks into audio events. The algorithm is described in Chapter 3. The application invokes the MapReduce job in Python for analyzing the set of videos. Each map job runs the speaker diarization algorithm on its assigned set of videos (see algorithm description in Chapter 3).

As mentioned in Chapter 6, the MapReduce component of PyCASP provides a high-level map function for applying the same diarization operation to a set of data segments in parallel. We use this function to perform the diarization computation on each video. Figures 7.5 and 7.6 show a portion of the video event detection code before and after it was refactored to use the MapReduce component. From the application point of view, this is the entire diarization phase of the video event detection system, consisting of 4 lines of code for the MapReduce component in

```
175 import sys
176 from speaker_diarizer import diarize
177
178 input_filenames = parse_input_filenames( sys.argv ):
179 for filename in input_filenames:
180   diarize(components=5, dim=60, clusters=16, data=filename )
```

Figure 7.5: A for-loop that applies the "diarize" operation to every filename. Python runs each iteration sequentially.

```
181 input_filenames = parse_input_filenames( sys.argv ):
182 map(diarize, components=5, dim=60, clusters=16, data=input_filenames )
```

Figure 7.6: The same operation (in light-gray) expressed without an implied order, allowing each iteration to be executed in parallel.

addition to the diarization code (shown in the speaker diarziation example). With this modification, the application is now able to scale to a computer cluster. The same change would require at least 20× as many lines of Java code.

In the video event detection application, we compose the MapReduce component to map the event detection system to a cluster of parallel processors using a two-line code change in the application code. We compose the MapReduce structural pattern with the speaker diarization algorithm, which uses the GMM training component and the Iterator structural pattern. We have discussed in Section 7.2 how the speaker diarization implementation composes the GMM component calls using the Iterator structural pattern. The composition optimizations corresponding to the Iterator pattern are also used in the video event detection system, since it uses the speaker diarization algorithm to segment video soundtracks. In addition to the Iterator pattern and the GMM component, we use the MapReduce-based composition optimizations to scale the application to a cluster of parallel processors.

When invoked, the MapReduce composition mechanism generates an input record for each content file to be processed. The record is passed to a worker node, which calls the speaker diarization algorithm to efficiently diarize the soundtrack. The reduction step is skipped, and the call returns when all input records (video soundtrack files) have been successfully processed. Since our MapReduce structural pattern implementation uses Hadoop, it automatically handles load balancing by assigning records to under-utilized nodes and enables fault tolerance by reassigning records from nodes that have failed to healthy ones. Thus, when using PyCASP and its MapReduce-based composition mechanisms, a two line code change scales the application from a single parallel processor to a cluster of parallel processors within the same software environment. We discuss the results

```
183 GMM:
184     autotune: False
185     name_of_backend_to_use: "cilk"
186     template_path: "/disk/home/egonina/pycasp/specializers/gmm/"
187     cuda_device_id: 0
188 SVM:
189     autotune: False
190     name_of_backend_to_use: "cuda"
191     template_path: "/disk/home/egonina/pycasp/specializers/svm/"
192     cuda_device_id: 0
```

Figure 7.7: Example config file for PyCASP specializers.

of using the MapReduce structural pattern in this application in Chapter 8.

7.5 Porting Applications to Different Platforms

In Chapter 2, we discussed the different hardware architectures that we focus on in this work. Figures 2.3, 2.4 and 2.5 in Chapter 2, show block diagrams for a multi-core CPU, GPU and a compute cluster backends. These backends differ significantly in their processor architectures, memory hierarchies and programming environment, and thus, it is extremely difficult for one application programmer to enable portability to these platforms. Thus, before we discuss the specific results for both productivity and performance in the next chapter, we illustrate the process of porting applications to different platforms using PyCASP. As discussed in Chapter 5, the specializer writers are responsible for enabling portability of their specializers. The specializer enables portability by specifying the code generation logic for each specific class of hardware platforms (i.e. multi-core CPUs, CUDA GPUs, etc.). The interface of the specializer remains unchanged, and thus, application code that uses the specializers does not change when application is ported to a different platform. In order to specify what platform the application will be run on as well as other runtime parameters, we use a configuration file

Figure 7.7 shows a sample config file. The application writer needs to specify which platform to run the application on ("cilk" for Cilk+ backend and "cuda" for CUDA backend) as well as the CUDA device ID if the CUDA backend is selected and there are multiple GPUs in the machine. The application writer can also specify whether or not to use auto-tuning (automatically selecting the most efficient code variant of the generated implementation) for each specializer. We separate the application code from the configuration of the specific hardware platform it will be run on. The specializer logic of PyCASP's components understands the different platforms it can target. Thus, the application code needs to be written

once and then the application writer can use the configuration file to specify the particular platform he/she wants to run their application on.

7.6 Summary

In this Chapter, we have described the implementation details of the four example applications using Python and PyCASP. Using the example applications, we illustrated that we can enable application programmer productivity by embedding PyCASP in Python. We enable high performance of the applications by offloading the compute-intensive parts of the applications to parallel hardware. Specifically, we use the SEJITS methodology to implement the components of PyCASP, each component corresponding to a specific instance of an application pattern. Using SEJITS, PyCASP components automatically generate parallel implementation of the algorithms for the specific application pattern, and compile and execute it on the parallel hardware. By decoupling the application logic from the efficiency code and enabling portability with SEJITS, we enable application developer productivity and portability of the application code.

Chapter 8

Results

In this chapter we first present the results on the productivity, efficiency and portability of individual PyCASP components. We then analyze the performance of component composition based on the structural patterns. We then put things together and analyze the productivity, efficiency and portability of each example application written in Python and using the PyCASP components and the component composition mechanisms. For the hardware backends, we use two Intel CPU platforms and two NVIDIA GPU platforms as well as a cluster of 16 Tesla M1060 GPUs. The specs of the platforms are summarized in Table 8.1.

Processor	Intel Core i7	Intel Wesmere	GTX285	GTX480
Cores	4 cores (SMT)	2×6 (SMT)	30 cores	15 cores
SIMD Width	4 lanes	4 lanes	8 lanes	16 lanes
Clock Speed	2.66 GHz	3.33 GHz	1.51 GHz	1.45 GHz
Memory Capacity	6GB	24GB	2GB	2.6GB
Memory BW	32.0 GB/s	32 GB/s	141.7 GB/s	177.4GB/s
Compiler	icc 12.1.2	icc 12.1.2	nvcc 5.0	nvcc 5.0

Table 8.1: Parameters for the experimental platforms

8.1 Components

First, we discuss the standalone performance of the two PyCASP components, the GMM training and prediction and SVM training and classification. We described the design principles and implementation of the components using SEJITS in Chapters 4 and 5, and we now evaluate the productivity, efficiency and portability of each component individually.

8.1.1 Productivity

First, we focus on the complexity of the components (i.e. the SEJIT specializers) and the productivity of specializer writers. Using Asp, we implement the specific instances of application patterns as specializes. PyCASP's GMM and SVM components are template-based specializers, they include the template CUDA and Cilk+ code as well as Python code of the specializer logic. Table 8.2 shows the number of lines of code used to implement both components of PyCASP. The first column shows the number of lines of Python code used to implement the specializer logic (i.e. the code variant selection mechanism, templating and compilation toolchain). The second column shows the number of lines of efficiency code in the specializer templates. The development of the specializer is done once after that, the code is reusable in all applications that use the component. This allows for *specializer writer productivity*, i.e. in $600 - 800$ lines of Python code we encapsulate the efficient CUDA code into a reusable component of PyCASP. Thus, implementing specializers is itself made productive by using the Asp framework. We discuss the productivity results of implementing the example applications that use the PyCASP components later in this chapter.

8.1.2 Efficiency

Second, we look at the standalone efficiency of each component. Table 8.3 shows the GMM and SVM component standalone efficiency on a NVIDIA GTX480 and an Intel Westmere CPU (3.33GHz). We use a C++/Pthread-based implementation of the GMM training algorithm to evaluate the performance of the GMM component. We use the widely-used LIBSVM [19] threaded CPU implementation of the SVM training component to evaluate the performance of the SVM component. Both components achieve significant speedups compared to the state-of-the-art implementations. The GMM component achieves $5 - 20\times$ speedup using CUDA and $4 - 12\times$ using the Cilk+ backend compared to the state-of-the-art threaded CPU implementation running on the Intel Westmere CPU. The SVM training achieves $13 - 79\times$ speedup on GTX480 compared to the LIBSVM implementation. Speedup numbers range for a variable size of the models and training data. Thus, the efficiency of PyCASP's components is not only on-par, but actually beats the hand-tuned state-of-the art implementations by one or two orders of magnitude.

8.1.3 Portability

Finally, we look at the portability of PyCASP's components. Specifically, since we focused on the GMM component as an example of a portable component with code variants, we look at the portability of GMM training to the two types of par-

Component	Specializer Python Code	Efficiency Code
GMM	800	2700 (CUDA) + 700 (Cilk+)
SVM	600	3500 (CUDA)

Table 8.2: Number of lines of code for both components' specializer Python and template code.

Component	CUDA Speedup	Cilk+ Speedup
GMM	$5 - 20\times$	$4 - 12\times$
SVM	$13 - 79\times$	N/A

Table 8.3: Speedup of each component on NVIDIA GTX480 GPU and Intel Westmere CPU compared to state-of-the-art threaded implementations running on the Intel Westmere CPU.

allel platforms - multi-core Intel CPUs (running the CIlk+ backend) and NVIDIA GPUs (running CUDA). Without application code change (but by changing the config file described in Chapter 7) applications that use the GMM component can run on any CUDA-programmable GPU and Intel multi-core CPU. Figure 8.1 shows the performance of PyCASP's GMM training component on NVIDIA GPU (GTX480) and Intel multi-core CPU (Intel X 5680 Westmere). Both versions outperform the baseline Pthread implementation of the GMM training computation. Furthermore, the CUDA backend of the GMM component can beat even the hand-coded CUDA implementation [80] by selecting the best-performing algorithmic variant at runtime based on the problem size and hardware parameters (for more detail, see [30]). Thus, by targeting two types of platforms in the PyCASP GMM component, we can enable application portability to a variety of parallel hardware with no required application code change. We note that the SVM component is also portable across CUDA-programmable GPUs; the application programmer can specify the selected GPU backend by modifying the PyCASP configuration file.

8.2 Composition

We now analyze the performance improvements that are possible using the optimizations for composing PyCASP's components. In Chapter 6, we discussed how structural patterns can help us identify and understand the ways PyCASP components are composed together in applications. To recap our findings, by analyzing specific use cases of each pattern in our sample applications and we have determined that:

- Pipe-and-Filter composition corresponds to data structures being *shared*

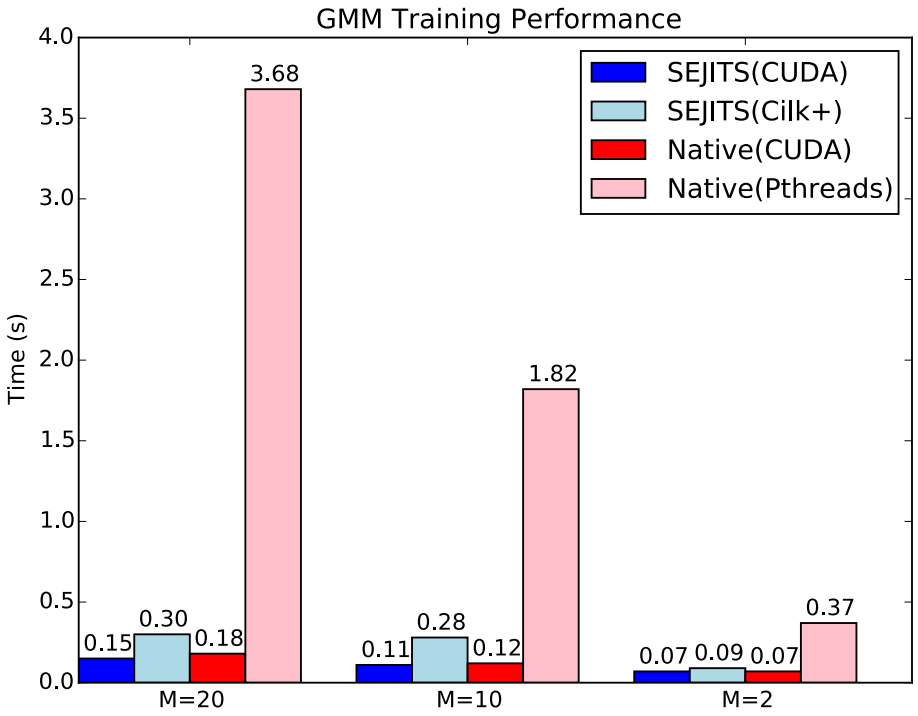

Figure 8.1: GMM training time (in seconds) given number of mixture-model components M using the CUDA backend and a native CUDA version (both on NVIDIA GTX480), and the Cilk+ backend and a C++/Pthreads version (both on dual-socket Intel X5680 Westmere 3.33GHz).

between computational components,

- Iterator composition corresponds to data structures being *reused* by computational components in an iterator loop,
- MapReduce composition corresponds to data structures being *distributed* across compute nodes in a computer cluster.

Thus, in order to optimize composition of PyCASP components, we can identify and remove redundant data structure allocation and copy calls. Specifically, we focus on running our applications on the GPU backend, since we must manage data structure allocation explicitly (as discussed in Chapter 6, Section 6.2.2) and removing redundant data allocation and copy calls to the GPU memory. Figure 8.2 shows the time required to allocate data structures of varying sizes (varied on the x-axis) and copy the data to and from the GPU memory (for NVIDIA GTX480

GPU). The overhead becomes much more significant as we move to larger data structures. In this section, we aim to understand how much of this overhead we can save removing redundant allocation and copy calls.

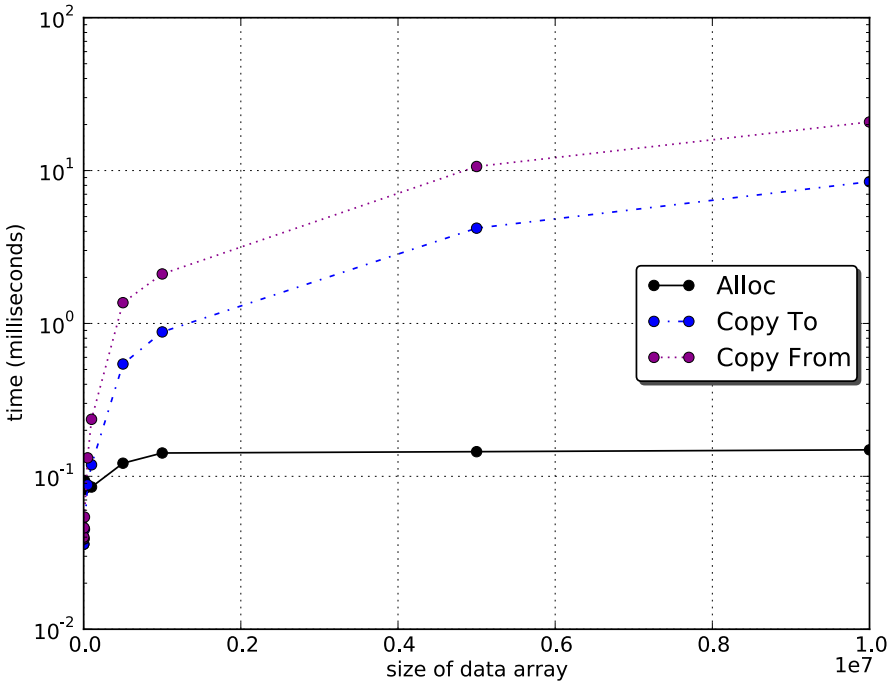

Figure 8.2: GPU memory management overhead.

We evaluate the optimizations in composing PyCASP's components using the three structural patterns by comparing execution time with and without data structure reuse. We use data sizes that are typical in audio content analysis applications. For example, we range the number of GMM components from $M = 16$ to $M = 1024$, as those are the number of components that are used to for acoustic modeling using GMMs. We consider the dimensionality of feature vectors of $D = 19$ and $D = 39$ corresponding to MFCC features with delta and delta-delta coefficients respectively.

8.2.1 Pipe-and-Filter

As discussed in Chapter 6, the Pipe-and-Filter-based composition corresponds to data structures shared between calls to PyCASP components. We can optimize the composition of components using Pipe-and-Filter by realizing what specific data structures are shared across component calls and removing redundant data

allocation and copy calls. In Chapter 6, we discussed three specific examples of composing PyCASP components using the Pipe-and-Filter pattern:

1. GMM training - GMM prediction (Figure 6.3). GMM parameters (weights, means, covariances) are shared between the two component calls.

2. SVM training - SVM classification (Figure 6.4). SVM parameters (support vectors) are shared between the two component calls.

3. GMM training - SVM classification (Figure 6.5) in the speaker verification application. GMM means (i.e. the supervector) is passed between the two component calls.

In each application example, we optimized the Pipe-and-Filter-based composition by removing the data structure reallocation and copy in the second component call and instead *sharing* the data structure by *passing a reference* to the allocated data structure from the first component call to the second. To evaluate the benefit of this optimization, we measured the time that it takes to execute the second component call in each example application (including the data allocation and copy), and the time spent allocating and copying data structures. We then analyzed the fraction of the total execution time of the second component call spent in data allocation. With our optimization, we are able to remove the data allocation calls and reduce the execution time by that amount.

Tables 8.4 and 8.5, show the performance data for the GMM train - GMM prediction application example for different number of GMM components, M, and data dimensionality D, $D = 19$ and 39 respectively. The tables show that for small models ($M = 16$, $M = 32$), data allocation time for GMM prediction takes $2.0 - 3.8\%$ of the total GMM prediction time. This fraction becomes even less significant as the model sizes grow, down to 0.5% and 0.9% for $M = 1024$ ($D = 19$ and 39 respectively). Thus, we are able to reduce the runtime by $0.5 - 3.8\%$ by removing redundant data allocation and copy calls by recognizing this Pipe-and-Filter composition of components in our applications. While this reduction in runtime is not as significant for individual small problems, it can become significant when we scale the application to run many (hundreds to thousands) of instances of this application.

For SVM training - SVM classification pipeline, the optimization yields even less significant results as the support vector data structures are typically small (10-200KB). We measured the allocation and copy calls for support vectors (data structure size of 152KB) in a sample application to take 0.7 milliseconds. Thus, the optimization of removing redundant data allocation and copy calls for this specific example usage does not present an opportunity for significant performance improvement.

M	GMM predict time (ms)	GMM param alloc time (ms)	Alloc fraction
16	11.55	0.44	3.8%
32	16.19	0.47	2.9%
64	25.37	0.49	1.9%
128	44.27	0.52	1.2%
256	77.58	0.66	0.9%
512	147.22	1.12	0.8%
1024	283.10	1.36	0.5%

Table 8.4: D = 19. Total time (in ms) for GMM prediction, parameter allocation time and fraction of total time due to allocation. The allocation procedure allocates and copies $D \times M$ means, $D \times D \times M$ covariance and M weights on the GPU.

M	GMM predict time (ms)	GMM param alloc time (ms)	Alloc fraction
16	15.70	0.38	2.4%
32	21.44	0.44	2.0%
64	32.60	0.65	2.0%
128	54.07	1.04	1.9%
256	97.20	1.35	1.4%
512	169.02	1.89	1.1%
1024	322.10	3.02	0.9%

Table 8.5: D = 39. Total time (in ms) for GMM prediction, parameter allocation time and fraction of total time due to allocation. The allocation procedure allocates and copies $D \times M$ means, $D \times D \times M$ covariance and M weights on the GPU.

Finally, we study the last example of composition using Pipe-and-Filter, the GMM train - SVM classify composition in the speaker verification application. The data structure that is shared between the two component calls is the GMM means (supervector) data structure, represented by a $D \times M$ dense matrix. The SVM input data allocation function copies the $D \times M$ input matrix from the CPU to the GPU, as well as aligns the matrix on the CPU, transposes it on the CPU, and allocates and copies it to the GPU. Tables 8.6 and 8.7 show the execution time of the SVM classification component for different input data sizes ($D \times M$), for varying M and $D = 19$ and 39 respectively. The tables show the total classification time of SVM classification (including data allocation), the execution time of the data allocation calls and the fraction of the execution time spent in the data allocation function. Since SVM classification computation itself is very efficient and the SVM input data allocation function makes several allocation and

copy calls, the fraction of the SVM classification call due to data allocation is quite significant, from $19.8 - 21.2\%$ for smaller input to $23.7 - 30.4\%$ for larger input. Thus, this example illustrates that there are indeed opportunities for significant performance improvement by enabling the composition optimizations in our framework. We are able to significantly reduce the SVM classification performance by recognizing the data structure sharing across the two component calls and removing redundant data allocation and copy calls.

M	SVM classify time (ms)	Data alloc time (ms)	Alloc fraction
16	1.80	0.36	20.0%
32	1.95	0.39	20.0%
64	2.44	0.49	20.1%
128	2.91	0.69	23.7%
256	5.11	1.34	26.2%
512	9.15	2.48	27.1%
1024	16.7	4.83	28.9%

Table 8.6: D = 19. Total time (in ms) for SVM classification, allocation time and fraction of total time due to allocation. The allocation procedure allocates and copies $D \times M$ GMM means (i.e. the supervector) on GPU, aligns data on the CPU, allocates and copies transposed data on CPU and GPU.

M	SVM classify time (ms)	Data alloc time (ms)	Alloc fraction
16	1.97	0.39	19.8%
32	2.21	0.45	20.4%
64	3.40	0.72	21.2%
128	5.20	1.35	26.0%
256	9.23	2.54	27.5%
512	16.50	4.94	29.9%
1024	30.80	9.36	30.4%

Table 8.7: D = 39. Total time (in ms) for SVM classification, allocation time and fraction of total time due to allocation. The allocation procedure allocates and copies $D \times M$ GMM means (i.e. the supervector) on GPU, aligns data on the CPU, allocates and copies transposed data on CPU and GPU.

8.2.2 Iterator

In Chapter 6, we identified that when PyCASP's components are composed using the Iterator structural pattern, certain data structures are reused across iterations.

This insight gives us an opportunity for optimizing composition of components - we can identify which data structures are reused and remove redundant allocation and copy calls by using lazy allocation. When the same data structure is reused across component calls, we can skip the data allocation and copy calls and just reuse the previously-allocated data structures. As before, we focus on the GPU backend. In addition to data structure reuse opportunities, we also identified an opportunity for optimization during audio segmentation. This audio feature gather operation can be performed either in the Python code of the PyCASP component, or in the low-level CUDA code. We now evaluate the performance gains we obtain by reusing data structures across component calls in an Iterator pattern as well as study the performance of the two different gather mechanisms in the audio segmentation task.

We have identified three examples of composition of components and opportunities for optimization using the Iterator pattern:

1. GMM training on multiple sets of input data points. GMM parameters are reused across subsequent calls to the GMM training function.

2. Audio segmentation in the speaker diarization application. Input audio features are reused by speaker models and audio segmentation calls.

3. Audio segmentation in the speaker diarization application. Audio features can be gathered in Python or in CUDA code with varying efficiency.

First, we look at the performance gains of reusing GMM parameters when training the same GMM on multiple sets of feature vectors. Tables 8.8 and 8.9 show the time to train one GMM of varying sizes ($M = 16$ to $M = 1024$, $D = 19$ and $D = 39$ respectively) on six (6) datasets (two of each, $N = 10,000$, $N = 50,000$ and $N = 100,000$, D-dimensional features). The leftmost columns show the execution time (in milliseconds) of training the model on the six datasets without reusing the GMM parameters. The second column shows the time to train the model reusing the model parameters. The third column shows the difference between the two execution times, i.e. the time that we save by removing redundant parameter allocation calls. The last column shows the fraction of runtime due to data allocation. The tables illustrate that the GMM parameter allocation time is a very small fraction of the total execution of GMM training when iterating over several detasets, in our examples it was less than 0.25%. Thus, while the absolute reduction in time is $2.63 - 53.24$ milliseconds, the relative reduction in time is not very significant.

When we look at the reuse of the input data (i.e. the audio feature vectors) across GMM training calls, removing redundant data allocation and copy calls presents a much greater opportunity for optimization. Tables 8.10 and 8.11 show the allocation time (in seconds) for the input audio features of varying sizes

M	w/o composition (ms)	with composition (ms)	Diff (ms)	% reduction
16	1,074.43	1,071.80	2.63	0.24%
32	1,511.69	1,508.25	3.44	0.23%
64	2,403.20	2,397.31	5.89	0.25%
128	4,182.97	4,175.59	7.38	0.18%
256	7,767.70	7,751.56	16.14	0.21%
512	14,870.70	14,844.04	26.66	0.18%
1024	29,134.30	29,100.882	33.48	0.11%

Table 8.8: D = 19. Time to train one GMM 6 times on varying datasets (two sets of $N = 10,000$, $N = 50,000$, $N = 100,000$ features) and different model sizes ($M = 16 - 1024$). Time (in ms) with and without composition (i.e. GMM parameter reuse), absolute and % reduction due to data reuse optimization.

M	w/o composition (ms)	with composition (ms)	Diff (ms)	% reduction
16	2,652.60	2,646.73	5.87	0.22%
32	5,439.29	5,429.26	10.03	0.18%
64	10,216.17	10,204.90	11.27	0.11%
128	19,793.47	19,779.90	13.57	0.07%
256	38,901.64	38,886.73	14.91	0.04%
512	77,231.10	77,196.59	34.51	0.04%
1024	153,762.80	153,709.57	53.24	0.03%

Table 8.9: D = 39. Time to train one GMM 6 times on varying datasets (two sets of $N = 10,000$, $N = 50,000$, $N = 100,000$ features) and different model sizes ($M = 16 - 1024$). Time (in ms) with and without composition (i.e. GMM parameter reuse), absolute and % reduction due to data reuse optimization.

($N = 10,000$ to $N = 1,000,000$ $D = 19$-dimensional features) and execution time (in seconds) of training an $M = 16$ and $M = 64$ component GMM respectively ($D = 19$ in both cases). The right-most columns show the fraction of GMM training execution time due to input data allocation. For both model sizes, the input data allocation time takes up from 2.28% to 10.01% of the total GMM training runtime. The larger the input data, the less significant the data allocation call becomes in the overall computation, as the GMM training execution time increases. This presents a significant optimization opportunity for applications that reuse the input data over many GMM training calls (such as the speaker diarization application). By identifying input data reuse, we can reduce the execution time of GMM training calls by up to 10.3%.

Finally, we look at the two different gather mechanisms in the audio segmentation example. Figure 8.3 shows the total time for training a GMM using the

N	Allocation time (sec)	Total GMM train time (sec)	Alloc fraction
10,000	0.06	0.54	10.26%
50,000	0.06	0.62	9.33%
100,000	0.07	0.73	9.05%
500,000	0.08	1.52	5.34%
1,000,000	0.10	2.53	3.87%

Table 8.10: M = 16. Time (in seconds) to train a GMM on input data of different sizes (N $D = 19$-dimensional features). Input data allocation time, total training time and the fraction of the total time due to data allocation.

N	Allocation time (sec)	Total GMM train time (sec)	Alloc fraction
10,000	0.06	0.59	10.01%
50,000	0.06	0.74	8.63%
100,000	0.07	0.91	7.37%
500,000	0.08	2.37	3.29%
1,000,000	0.10	4.23	2.28%

Table 8.11: M = 64. Time (in seconds) to train a GMM on input data of different sizes (N $D = 19$-dimensional features). Input data allocation time, total training time and the fraction of the total time due to data allocation.

two types of gather mechanisms on varying subset sizes. For each data point in the graphs, we varied the size of the subset of features that is gathered in either Python code or CUDA code. The three plots show the execution time (in seconds) of three GMMs with $M = 5$ and $M = 15$ components (corresponding to the GMM model sizes in the speaker diarization application). The plots show that the crossover points vary for different model sizes, with the CUDA-gather implementation becoming more optimal for larger models. Overall, we see that the trade-off point for using Python-gather vs CUDA-gather lies around the subset size of $50,000 - 65,000$. After this point, the CUDA-gather based implementation is more efficient than the Python-gather implementation. Thus, the most efficient gather mechanism depends on the size of the subset of input data that is being gathered. In our case, we use subset sizes of only 250 feature vectors, and thus, the Python-based gather is most efficient for the example speaker diarization application.

8.2.3 MapReduce

The MapReduce structural pattern composition corresponds to data structures being distributed across compute nodes for parallel computation. When we map an application from a single node to a cluster of compute nodes, we need to

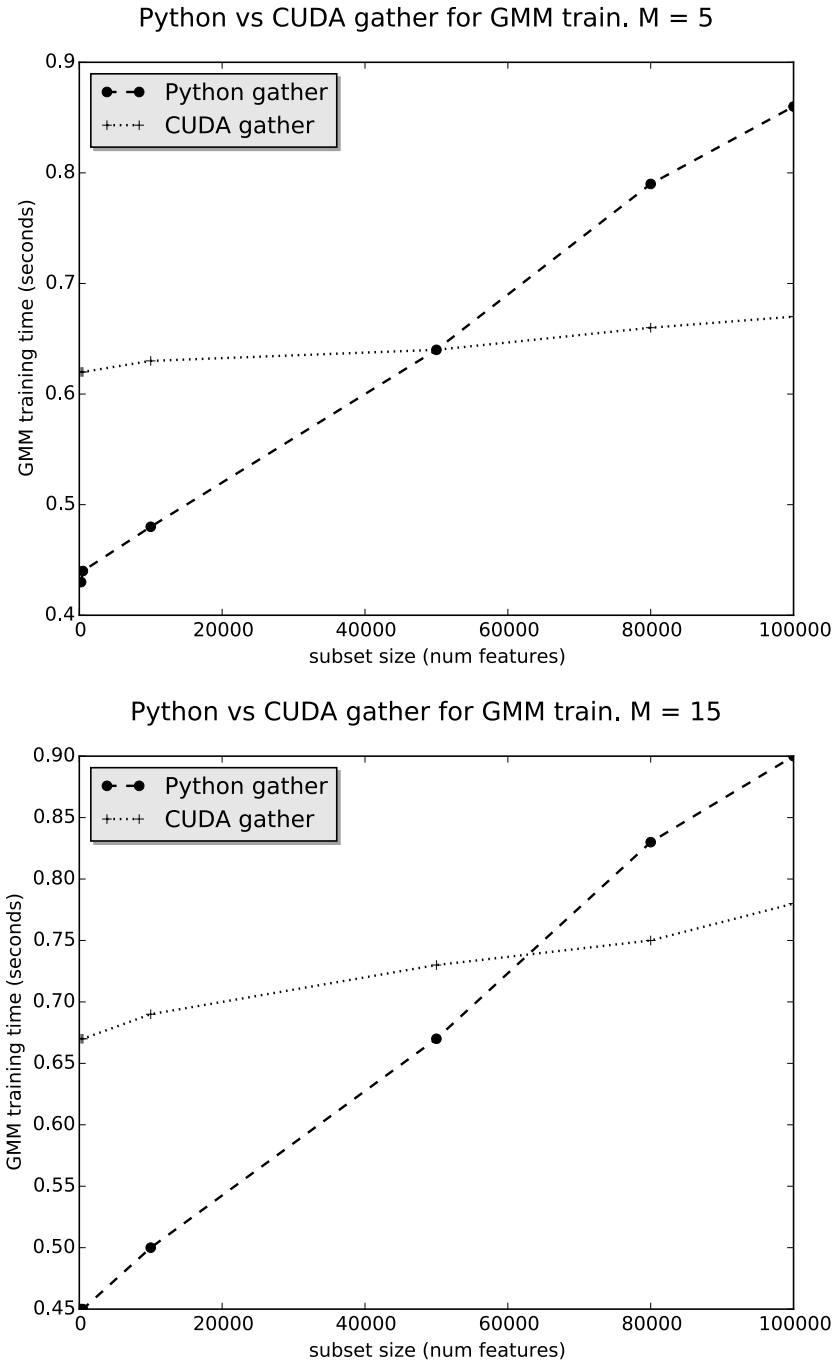

Figure 8.3: GMM training time with Python and CUDA gather mechanisms for $M = 5$ and $M = 15$ GMM components.

	Speaker Verification	Speaker Diarization	Music Recommendation	Video Event Detection
LOC	260	700	600	720
Performance remarks	Train: 13 min speech / 2.4 sec Classify: 5 min speech / 0.3 sec	$71 - 115\times$ Faster-than-real-time	Under 1 second recommendation time for all queries	$15.5\times$ speedup on 16 nodes

Table 8.12: Number of lines of Python code and performance remarks for the four example applications.

distribute the data across these compute nodes to maximize data locality. In this work, we focus on the basic use of MapReduce to map computations to a cluster of nodes by distributing a dataset across the compute nodes and executing a `map()` function on each subset of data in parallel. Each map computation can be a Python function or call other PyCASP components. In our example video event detection application, the map function is the diarization algorithm that gets executed on each video file.

The videos are distributed across the compute nodes in the beginning of the computation. Before calling MapReduce applications, the application programmer has to explicitly transfer data to the computer cluster. In our case, data needs to be transferred to HDFS. To transfer the video dataset across a network with a 200-300 MB/s bandwidth takes about 5-7 seconds. This is the standard first step in using MapReduce to enable data locality. The data transfer has to be done once for the entire application before invoking the application algorithm. Thus, we compose the data distribution procedure with the other optimizations implemented in the speaker diarization application (such as input data and GMM parameter reuse across GMM training calls). We discussed the implementation details in Chapter 6 and optimization results earlier in this chapter.

8.3 Applications

We now describe the results on productivity, efficiency and portability based on the four example applications: speaker verification, speaker diarization, music recommendation and video event detection using the efficient PyCASP components and composition optimizations. We use the four different hardware platforms shown in Table 8.1 for our evaluation. Each application takes full advantage of

the efficiency of the PyCASP components in its implementation as well as the composition optimizations based on structural patterns.

We evaluate productivity in terms of how productive the *application developer* can be when developing the application. As discussed in Chapter 2, we base our evaluations on one of the common evaluation metrics - the lines of application code (LOC) as well as code reuse. We note that in order to fully understand productivity of our approach, we need to look at both application programmer productivity and the specializer developer productivity. The application lines-of-code consist of both the Python application code *and* the specializer-generated code. It is important to note that the Python lines of code that implement the application does not include all of the code that gets executed when the application is run. Specifically, the specializer logic and generated code, captured outside of the application also gets called when an application is running on a particular backend. We discussed the specializer developer productivity and the lines of code needed to implement each specializer earlier in this chapter. We emphasizes that the specializer code needs written *once*, and then is *reused* across many applications. In this section, we describe the the number of lines of Python code that is required for an application developer to prototype an application (re)using PyCASP.

We evaluate the efficiency of each application based on the application-specific evaluation criteria. Table 8.12 summarizes the results on productivity and efficiency for each application. Each application has a specific target when it comes to performance. The baseline for the evaluation is the execution time; we can compare the execution time with other implementations of the same application to get an idea of the efficiency of our applications. The faster we can perform a particular computation, the better. Another metric, typically used in interactive applications is the Real-Time-Factor (×RT), which is computed by dividing the meeting time by the processing time. For example, a 100× real-time factor means we can process a ten minute meeting in six seconds. We use the Real-Time-Factor (×RT) to evaluate the speaker verification and speaker diarization applications since these applications rely on fast, interactive processing of audio. For large-scale applications that process lots of data (such as our video event detection example application), it is also important to measure scalability - i.e. how well does the application scale to and utilize underlying hardware.

Finally, we measure portability by looking at the efficiency of executing each application on a variety of available parallel hardware and analyze the results. We use two Intel CPU platforms and two NVIDIA GPU platforms for our experiments. The specs of the platforms are summarized in Table 8.1. We describe the specific results for each application using the subset of the hardware platforms in the sections below.

8.3.1 Speaker verification

Productivity

The speaker verification system is implemented in about 260 lines of Python code. The most compute-intensive phases of the application are the GMM and SVM training. As discussed in Chapter 7, we use the GMM and SVM components of PyCASP to offload the compute-intensive phases to parallel hardware. The rest of the application consists of reading in and parsing the training and testing datasets and setting up the UBM adaptation and SVM classification procedures. This application illustrates the rapid prototyping capabilities of PyCASP - we developed the full-functioning speaker verification application in about one week of engineering time. We do not have a comparable speaker verification application written in a low-level language; however, looking at the productivity advantage of Python over low-level languages such as C/C++, the same application would require at least $10 - 60\times$ more lines of code to implement the same functionality.

Efficiency

To evaluate the efficiency of the speaker verification application, we look at the time that it takes for the application to train the GMM and SVM models and the time that it takes to classify a piece of previously-unsceen audio. Since speaker verification application requires quick, real-time feedback to the user, it is critical for the application to run as quickly as possible, much faster than real-time. We measure efficiency in two ways - the total time it takes to train the speaker and classification models, and the real-time-factor based on the length of the input audio.

As discussed in Chapter 7, the speaker verification application uses two components of PyCASP - the GMM and SVM components for learning speaker models and performing classification. The GMM component can generate CUDA and Cilk+ code and thus can run on either NVIIDA GPU backends or Intel multi-core CPU backends. The SVM component generates CUDA code and can run on NVIDIA GPU backends. Thus, for our experiments, we use two NVIDIA GPUs, the GTX480 (Fermi) GPU and the GTX285 GPU to run the CUDA backend and a 4-core Intel Core i7 CPU for the Cilk+ backend. We can either run both components on a GPU (and thus utilize the composition optimizations), or run the GMM training on the multi-core CPU and the SVM component on a GPU.

Figure 8.4 shows the performance of the two phases of the speaker verification application: training and testing, on the four hardware configurations. The blue bar shows the time (in seconds) to train the UBM, adapt it to two types of speakers and train the SVM. The magenta bar shows the time to classify a piece of audio to one of the two classes ("target speaker" or "intruder"). Training dataset consists

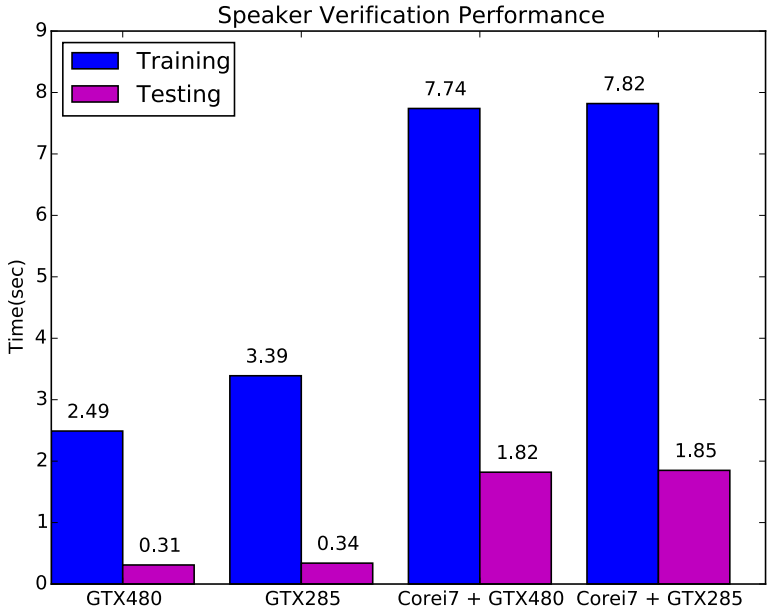

Figure 8.4: Performance of the speaker verification application on a variety of parallel hardware. Training dataset consists of 825.21 seconds (13.75 minutes) of speech and testing dataset consists of 344.63 seconds (5.74 minutes) of speech.

of 825.21 seconds (13.75 minutes) of speech and testing dataset consists of 344.63 seconds (5.74 minutes) of speech.

Table 8.13 shows the performance of the speaker verification in terms of the real-time-factor. The performance numbers indicate, that we can train and classify speaker audio within seconds and orders of magnitude faster than real-time. Thus, the application can be run in real-time which is the primary requirement of a speaker verification system. We do not have a full-functioning speaker verification application implemented in a low-level language to compare our efficiency results to; however, based on the performance of both components (GMM and SVM, discussed earlier in Section 8.1), we can conclude that our implementation of the speaker verification application achieves state-of-the-art efficiency.

Portability

Figure 8.4 shows the performance of the speaker verification training and classification phases on four different hardware configurations: GTX480 GPU, GTX285 GPU, Intel Core i7 CPU + GTX480 GPU and Intel Core i7 CPU + GTX285 GPU. As noted earlier, we can run the GMM training component on either a

Platform	Training ×RT	Testing ×RT
GTX480	331	1111
GTX285	243	1013
Core i7 + GTX480	106	189
Core i7 + GTX285	105	186

Table 8.13: Speaker verification Real-Time-Factor (×RT).

GPU or a CPU and the SVM component on any NVIDIA GPU. By using a configuration file (described in Chapter 7), we can specify which backend to run each component on. As shown in Figure 8.4, the best-performing platform for this application is the NVIDIA GTX480 GPU. The GTX285 GPU (which has less cores and lower DRAM bandwidth), gets similar performance for classification phase, but is almost 1 second slower for the training phase. Using the Cilk+ backend for the GMM slows the application down significantly regardless of the GPU that is used to run the SVM component. This indicates that the GMM training during both model learning and classification phases is impacted significantly by the shift to the multi-core CPU from a GPU. In addition, we cannot take advantage of the composition optimizations because the two components use different device memories. Using the SEJITS-based implementations of PyCASP components and enabling backend selection using the configuration file, we are able to run this application on a variety of hardware targets without application code change. This illustrates how easily an application writer can experiment with different backends for his/her application to select the platform that performs best. In the case of speaker verification, the GTX480 is the "winning" platform.

8.3.2 Speaker diarization

Productivity

The entire speaker diarization application is implemented is about 700 lines of Python code, which includes the core algorithm, feature file reading, parsing and speech-non-speech pruning, output formatting and storage, printing of usage instructions and parsing of the configuration input file. The core algorithm of the speaker diarization application is captured in 100 lines of Python code as shown in Chapter 7, Figures 7.2 and 7.3. The application uses the GMM component of PyCASP to train speaker models and segment the meeting audio based on the models. The code uses Numpy arrays for manipulating the feature and parameter data and Python for file I/O and application "glue" code. A comparable speaker diarization application written in C++ requires about 30× more lines of code [39]. Thus, PyCASP enables application programmers to capture an equivalent,

complex state-of-the-art application in an order of magnitude less lines of code, enabling high programmer productivity.

Efficiency

We evaluate the performance of the speaker diarization system using the real-time-factor metric to measure speed as well as the Diarization Error Rate (DER) to measure accuracy. We studied this particular application in extensive detail to show that not only can we achieve productivity and efficiency with our framework, but we can ensure that the application performs as well as other state-of-the-art speaker diarization applications in terms of accuracy. We discuss the details implementation and the analysis of accuracy and performance of our speaker diarization application in [43]; this work was well-received by the speech recognition community, whose domain experts struggle with productive utilization of parallel processors. Here, we summarize the performance results of this application.

We use a popular subset of 12 meetings (5.4 hours) from the Augmented Multi-Party Interaction (AMI) corpus [13] to analyze the performance in terms of speed and accuracy and also compare our speaker diarization application to an equivalent system written in C++. The AMI corpus consists of audio-visual data captured from four to six participants in a natural meeting scenario. For the experiments described here, we used the beamformed far-field (FF) and near-field (NF) array microphone signals.

FF DER	FF ×RT	NF DER	NF ×RT
35.49 %	71.02×	24.76 %	115.40×

Table 8.14: Average Diarization Error Rate (DER) of the diarization system and the faster than real-time performance factor (×RT) for far-field (FF) and the near-field (NF) microphone array setup for the AMI corpus.

Table 8.14 shows the average performance numbers in terms of accuracy and speed of diarization across all the meetings in the AMI dataset. Columns "FF DER" and "NF DER" show the average accuracy (Diarization Error Rate (% DER) [1]). The accuracy of our system is consistent with the state-of-the art system described in [39]. Columns "FF ×RT" and "NF ×RT" in Table 8.14 show corresponding performance for far-field and near-field microphone setup in terms of faster than real-time factor (×RT) using PyCASP on NVIDIA GTX480 GPU. The real-time performance varies by each meeting from about 50-250×RT, depending on the length of the audio as well as the number of clustering iterations computed before convergence (for details see [43]). Our reference C++ application achieves about real-time performance. Thus, our speaker diarization application performs orders

[1] National institute of standards and technologies: Rich transcription spring 2004 evaluation.

Mic Array	Orig. C++/pthreads	Py+Cilk+	Py+CUDA
	Westmere	Westmere	GTX285/GTX480
Near field	20×	56×	101 × / 115×
Far field	11×	32×	68 × / 71×

Table 8.15: The Python application using CUDA and Cilk+ outperforms the native C++/Pthreads implementation by a factor of $3-6\times$.

of magnitude better than the reference C++ implementation in terms of speed, and just as well in terms of accuracy.

Portability

The speaker diarization application uses the GMM component of PyCASP. As described in Chapters 5 and 7, we designed the GMM component to be portable to CUDA GPUs and Intel multi-core CPUs. Table 8.15 shows the speaker diarization performance on three different backends: Intel Westmere CPU and on two generations of NVIDIA GPUs, GTX285 and GTX480. We compare the performance of the PyCASP-based implementation with the original implementation running C++ and Pthreads on the Intel Westmere CPU. On all platforms, the implementation that uses PyCASP achieves higher performance than even the reference sequential C++ implementation in terms of the real-time-factor (and as noted above, with 30× fewer lines of code). The best-performing platform is again, the NVIDIA GTX480 GPU. Since performance of this application is highly-dependent on the performance of the GMM training component, the platform that can execute GMM training computations is the most efficient platform for the entire application. Thus, we enable application portability as well as high-performance on a variety of parallel hardware by using SEJITS to implement the components of PyCASP.

8.3.3 Music recommendation

Productivity

The music recommendation system is written in about 800 lines of Python code. Out of the 800 lines, 600 lines capture the recommendation algorithm and additional 200 lines implement the music server interface, database querying other auxiliary functions. The application uses the GMM component of PyCASP to train and adapt the UBM for music similarity computation. We use the SQLite Python library [79] for setting up the music server and for manipulating and accessing the song database. We use the Numpy library [5] for auxiliary numerical

computations, such as the distance computation to find the set of closest songs to the query songs.

Figure 7.4 in Chapter 7 captures the core Python code for the online phase of the application. Using the high-level Python language as well as offloading the compute-intensive GMM training computation to parallel platforms using PyCASP, we were able to prototype this application in about three weeks of engineering time. Python enables easy integration of a variety of tools, for example, it is quite straightforward to import and use the SQLite package for database manipulation and the Numpy library for numerical computations. In addition to using PyCASP, the availability and the ability to easily integrate a variety of Python tools into our application code aids in productive application development. We do not have a comparable low-level implementation of the music recommendation system, but from our experience, we can estimate that a similar implementation would take an order of magnitude more lines of code, and thus, significantly more development time.

Efficiency

To evaluate the efficiency of our music recommendation system, we use two datasets: the $10,000$ song subset of the Million Song Dataset [8] and the entire set of one million songs in the Million Song Dataset on one NVIDIA GTX480 GPU desktop. Figure 8.5 shows the time it takes to adapt the UBM and the total recommendation time (excluding the database accesses) for various query sizes for the two datasets respectively. The total time contains UBM adaptation time, approximate nearest-neighbor computation using LSH and computing the C closes songs returned by LSH (algorithm steps 2-4 discussed in Chapter 3). We used artist names to query our recommendation system to come up with the list of query songs; for example, we ran queries like "Radiohead" and "Elton John or Eric Clapton" to find songs that are musically-similar to all songs by Radiohead and to both Elton John and Eric Clapton. In the 10K system, typical queries returned $1 - 17$ songs, up to $23,000$ feature vectors. The number of songs returned by our queries using the 1M song dataset ranged from 30 to 500, up to $560,000$ feature vectors total. Figure 8.5 shows the scalability of PyCASP's GMM training component as we increase the number of features returned by the query. For the largest query our system is able complete the online recommendation phase from the one million song dataset in under 1 second. The GMM training component in PyCASP can process up to 7 million features to train $M = 64$-component GMMs on NVIDIA GTX480 GPU. Thus, we believe that the online phase of the application will scale to even larger queries.

The offline phase takes a significantly longer time to execute than the online phase. The most significant component of the offline phase of the recommendation system is the UBM feature gather, the SQLite database setup and the LSH hash

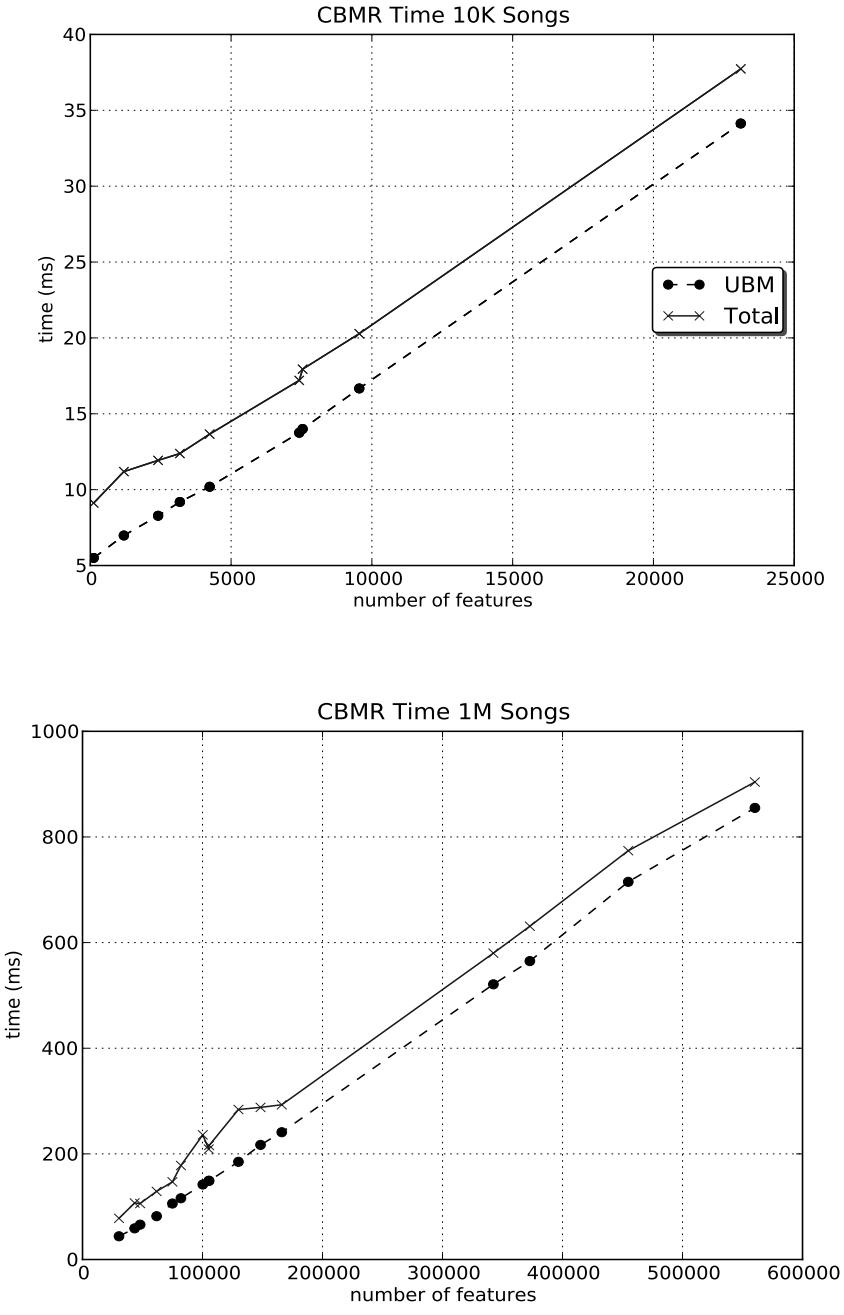

Figure 8.5: Scaling of the [Content-Based] Music Recommendation (CBMR) system on the 10,000 song subset of the Million Song Dataset (top) and on the full dataset (bottom).

table setup. Since these operations require frequent disk access, the offline phase of our recommendation system is disk I/O bound and takes about 24-36 hours to complete on the 120GB database. The Sqlite database query time can range quite significantly, depending on whether the data is in memory or on disk, ranging anywhere from 1 to 200 seconds. The UBM adaptation itself on 7M randomly-sampled features takes about 50 seconds on the NVIDIA GTX480 GPU platform, and is not the bottleneck. Thus, due to these factors and the fact that the offline phase of the system needs to be run once for the entire appellation, we don't include it in the detailed analysis of performance.

Portability

Our music recommendation system makes heavy use of the GMM component of PyCASP, and thus, music recommendation system can run on a multi-core CPU or GPU without code change. In the offline phase of the recommendation system, we train a UBM on a large set of randomly-sampled features. This is is the compute-intensive part of the offline phase of the application (other time-intensive phases are database setup and file parsing, which are I/O bound). To train large GMM models, the most efficient platform to execute UBM training is the NVIDIA GTX480 GPU. Thus, although we are able to use other NVIDIA GPUs and Intel multi-core CPUs, we choose to use the GTX480 platform for the offline phase of the recommendation system.

For the online phase of the recommendation system, it is less clear what platform will perform best. The two NVIDIA GPU platforms we use for our evaluation have separate device memories; running an application on an NVIDIA GPU requires explicit data structure transfer from the CPU memory to the GPU memory. Although GPUs are more efficient at executing GMM training (as we have seen in speaker verification application in Section 8.3.1), the overhead of data reallocation can impede the application performance on a GPU backend. Indeed, when we run a set of sample queries through our recommendation system, we see that using the Intel CPU Cilk+ backend (in this case we used the Core i7 4-core processor), is about $10 - 20\%$ faster than the NVIDIA GTX480 GPU backend (the total recommendation time on the Intel Core i7 is 3.2 seconds and on the NVIDIA GTX480 is about 3.6 seconds). Thus, when we train a GMM on a small set of features (as in the online phase of the music recommendation system), the data transfer overhead becomes more significant and impedes the performance of the GPU backend making the Intel CPU backend more efficient. We are able to perform these experiments by modifying the configuration file without changing the application code.

8.3.4 Video event detection

Productivity

The video event detection application is written in 720 lines of Python code. 100 lines correspond to the core functionality of the speaker diarizaiton algorithm. The remainder 600 lines correspond to file I/O and parsing, output formatting and storage, printing of usage instructions and parsing of the configuration input file. Finally 20 lines of code, implement the MapReduce code for scaling the the application to run on a computer cluster. The same change would require at least 20× as many lines of Java code. Thus, we are able to rapidly prototype the application by reusing the speaker diarization algorithm and adding the MapReduce code to scale it to the large dataset of videos.

Efficiency

We evaluate the efficiency of our system using the TRECVid Med 2011 dataset. This dataset is comprised of consumer-produced videos collected from social networking sites, or "found videos." The data is broken down into 15 categories or "event-kits", with 5 of those categories available in the test set. The event categories available in the test set are "attempting a board trick", "feeding an animal", "landing a fish", "wedding ceremony", and "working on a woodworking project". Of the test set, 496 videos are from these 5 categories, and the remaining $3,755$ videos are random videos not belonging to any event category. The system uses 60-dimensional features: C0-C19 Linear Frequency Cepstral Coefficients (LFCC) features with 25ms windows and 10ms step size, along with deltas and delta-deltas.

We performed the experiments on a cluster of 8 nodes with two NVIDIA Tesla M1060 GPUs each, total of 16 GPUs, using CUDA 3.2. We analyzed the speedup of the video event detection system using the MapReduce composition of PyCASP compared to running the diarization on each video sequentially on one machine with one GPU. We map an increasing number of videos from different event categories to nodes in our cluster to investigate the scalability of our implementation. Figure 8.6 shows that once we process enough video files, we obtain nearly perfect speedup - 15.5× on 500 and 1000 videos compared to running the same computation on one GPU node. This result shows that the video event detection application achieves nearly optimal scaling on the GPU cluster. We performed this experiment on Amazon EC2 cloud compute platform, yielding the same results. The efficiency of the diarization algorithm was shown in analysis of the speaker diarization application earlier, in Section 8.3.2. Thus, this example illustrates a near-perfect scaling and efficiency, of our video event detection application that uses PyCASP's MapReduce composition and GMM component to run on a cluster of GPUs.

Portability

Using the MapReduce functionality of PyCASP, we are able to easily port the video event detection application from running on one GPU node to a cluster of GPUs. By changing the configuration file we can also run the video event detection system on a cluster of Intel CPU machines. With this example application, we illustrate the portability of PyCASP application from desktop GPUs and CPUs to computer clusters. This also serves as an example of how PyCASP enables application writers to develop and prototype algorithms using a sample subset of data on one machine (utilizing underlying parallel hardware if using PyCASP components), and then easily scale their application to a larger dataset and utilize a whole cluster of machines.

8.4 Comparison to prior approaches

In Chapter 2, we discussed several main alternative approaches to bridging the implementation gap: efficiency languages and libraries, productivity language and libraries, and efficiency and productivity frameworks. The alternative approaches allowed for varying amount of programmer productivity and flexibility, and appli-

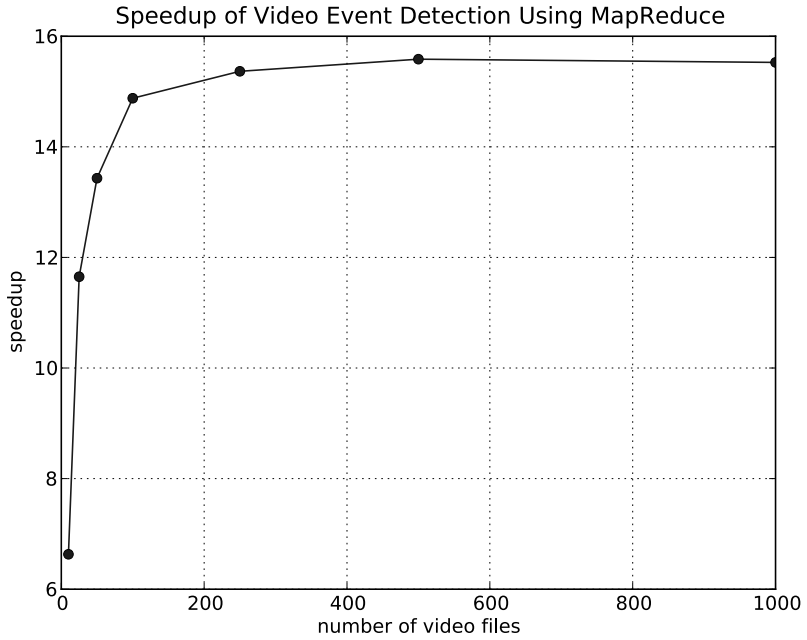

Figure 8.6: Scaling of the video event detection system using MapReduce for varying number of video files diarized.

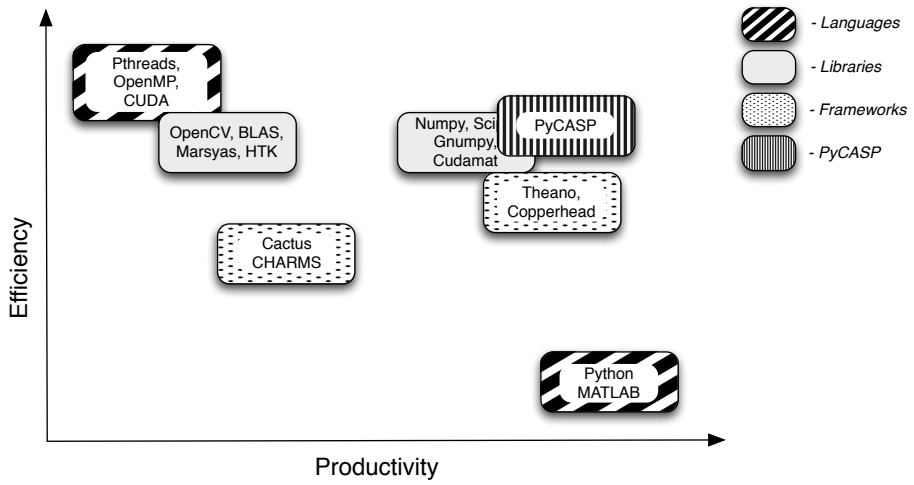

Figure 8.7: Comparing efficiency and productivity of alternative approaches and PyCASP.

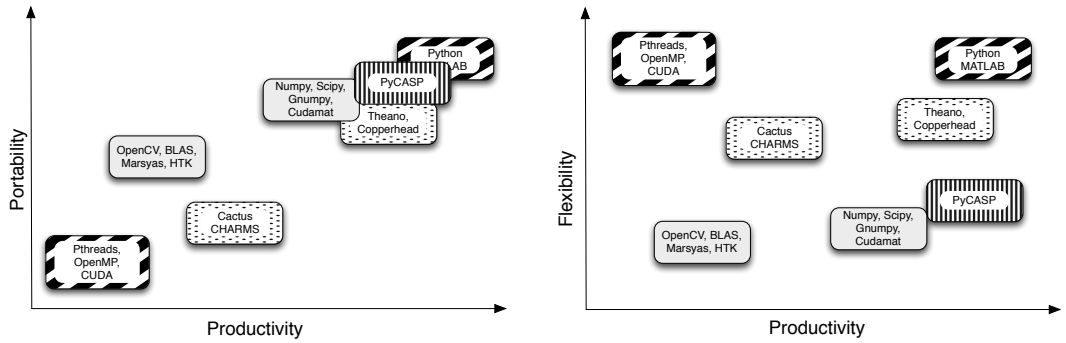

Figure 8.8: Comparing portability and productivity (left) and flexibility and productivity (right) of alternative approaches and PyCASP.

cation efficiency and portability, shown in Figures 2.6 and 2.7. Now that we have illustrated the productivity, efficiency and portability results of both PyCASP components, composition optimizations and applications written using PyCASP, we can discuss how PyCASP fits into this global picture.

Figures 8.7 and 8.8 show the relative positioning of PyCASP and each type of the alternative approaches on the Productivity-Efficiency scale (Figure 8.7) and the Productivity-Portability and Productivity-Flexibility scales (Figure 8.8). As a reminder, the Productivity scale denotes the *productivity of application developers* when using the particular approach. The Efficiency scale denotes the *efficiency of applications* that is possible when they are written using these approaches. The Portability scale denotes *how many different platforms* applications that are written using these approaches can run on. The Flexibility scale denotes the

application developer flexibility in specifying application functionality when using these approaches. While the positioning of each approach doesn't change on the horizontal (productivity) axis, they show quite different placement on the vertical (efficiency, productivity and flexibility) axes. Again, we note that the placement of each approach does not capture the exact measured efficiency and productivity, but is rather meant to show relative placement of PyCASP and each alternative approach on the global spectrum.

When looking at the Productivity-Efficiency scale, our component and application examples illustrate that PyCASP can be quite competitive with efficiency languages and libraries in terms of efficiency it provides to the applications. In addition, PyCASP is much more productive than the former. PyCASP provides productivity that is close to the productivity languages but allows for much more efficiency. On the Portability-Productivity scale, PyCASP enables high portability by targeting multiple types of platforms (GPUs, multi-core CPUs and clusters) while maintaining productivity. PyCASP provides higher portability than productivity libraries as it targets not only multiple platforms, but also multiple versions of those platforms by tuning the code that it generates to the underlying hardware. Finally, on the Productivity-Flexibility scale, PyCASP gives more flexibility to applications programmers than productivity and efficiency libraries by allowing application programmers to modify the specific details of computations in their applications. PyCASP is not as flexible as code-lowering frameworks such as Theano or Copperhead or programming languages; however, PyCASP is designed to close this gap with the customizable components, discussed more in Chapter 9.

We conclude that PyCASP is a *productivity* framework that presents application programmers with a single *application-specific* software environment that brings efficiency, productivity and portability to applications and provides more flexibility than traditional library approaches.

8.5 Summary

In this chapter, we presented results on productivity, efficiency and portability of PyCASP's components and applications written using PyCASP. We show that we are able to achieve productivity, efficiency and portability of both PyCASP components and the example applications. We also analyzed the performance gains achieved by utilizing optimizations when composing PyCASP's components in applications. We evaluated the performance gains possible when sharing (using the Pipe-and-Filter pattern), reusing (using the Iterator pattern) and distributing (using the MapReduce pattern) data structures. While some composition points, such as the GMM training - GMM prediction composition, did not yield significant performance improvement from removing redundant data allocation calls,

other composition points did. Specifically, we showed that we can reduce the runtime of the speaker verification classification phase by almost 30% by sharing data structures across the GMM training and SVM evaluation calls. Thus, we have illustrated that by restricting the scope of our framework to one application domain and using structural patterns to analyze composition of components in applications, we can optimize composition of computations and enable higher application efficiency. In the next chapter, we summarize and evaluate this work against our goals and discuss advantages and drawbacks of using our approach to create application-specific frameworks for productive parallel computing.

Chapter 9

Conclusion

9.1 Thesis Summary

With the commoditization of parallel processors, porting applications to utilize parallel processors is essential to enable application scalability to new generations of hardware. Unfortunately, the programming world is still divided into two types of programmers: application developers and researchers who focus on designing and prototyping algorithms, and efficiency programmers who focus on extracting the most performance out of a particular compute kernel. The gap between these two types of programmers is referred to as "the implementation gap". In order to enable applications to utilize parallel hardware, one way to bridge the implementation gap is to create a software environment that allows the two types of programmers to communicate with each other. In this dissertation, we set out to answer the following research question:

How can we build a software environment to bridge the implementation gap and enable application writers to productively utilize parallel hardware and develop efficient, scalable, and portable applications?.

In this thesis, we focus on developing a software framework that bridges the implementation gap between the two types of programmers. We aim for this software framework to enable application programmer productivity and flexibility in developing applications, as well as allow for application efficiency and portability to a variety of parallel hardware. We propose three main mechanisms to answer our thesis hypothesis:

1. We propose using a *pattern-oriented design* for our framework to enable productivity, modularity and comprehensiveness of our software solution. We use Our Pattern Language, originally presented in [62], for the design of our framework. The pattern-oriented approach allows us to define a

clear scope and enable modular and comprehensive coverage of the specific application domain. Patterns also provide for a common vocabulary for application domain experts and efficiency programmers.

2. We propose using *Selected Embedded Just-in-Time Specialization (SEJITS)* to realize the framework's design in software to enable productivity, flexibility, efficiency and portability. We use the SEJITS technology to realize the pattern-based design of our framework in software. SEJITS allows for the separation of concerns between the application developers and efficiency programmers enabling productivity, as well as efficiency and portability of resulting applications.

3. We propose *restricting the scope of the framework to one application domain* to enable efficient composition of computations. We restrict the scope of our framework to one application domain, in our case, the audio content analysis applications. Audio applications present a compelling set of applications due to the specific latency and throughput requirements and the ever-increasing amount of audio data that is available for analysis and machine learning. By restricting the scope of our framework to one application domain, we aim to understand the types of *compositions of computations* that occur in the applications and implement composition optimizations.

In this thesis, we present an example framework called PyCASP ("Python-based Content Analysis using SPecialization") whose design and implementation combines the three approaches described above. Using patterns, we aim to provide the programmers with a software environment that has a familiar vocabulary and a concise scope, allowing for productivity in application development. Using the SEJITS approach and the separation of programmer concerns it enables, PyCASP is able to provide high application efficiency and portability to a variety of parallel platforms, in addition to high-level language productivity.

By restricting the scope of the framework to one application domain, we are able to *optimize composition of computations*. We analyze the variety of ways PyCASP components are composed together into applications using structural patterns. We show that the three structural patterns, Pipe-and-Filter, Iterator and MapReduce, are sufficient to describe the types of compositions that occur in majority of audio content analysis applications. We verify the hypothesis by analyzing the types of computation compositions in a variety of audio content analysis applications. We then identify the optimization opportunities for specific compositions based on the structural patterns. Specifically, we discover that:

- Pipe-and-Filter composition corresponds to data structures being *shared* between computational components,

- Iterator composition corresponds to data structures being *reused* by computational components in an iterator loop,
- MapReduce composition corresponds to data structures being *distributed* across compute nodes in a computer cluster.

The opportunities for optimization of composing computations correspond to removing redundant data structure allocation and copy calls. We focus on the CPU-GPU backend, as we must manage data allocation and memory utilization explicitly. By identifying the specific ways computations are composed, we are able to remove redundant data allocation and thus, further optimize application performance. We show that, in addition to specialization, adding the optimizations for composing components in these applications can give up to 30% performance improvement.

To illustrate the applicability of PyCASP we analyze the implementation of four full-functioning audio content analysis applications using PyCASP: a speaker verification system, a speaker diarization application, a content-based music recommendation system, and a video event detection system. We use two NVIDIA GPUs, two Intel multi-core CPUs and a cluster of NVIDIA GPUs to run and analyze the performance of the four applications. We show that across this wide range of applications and parallel platforms, PyCASP is able to give high productivity to application writers and efficiency and portability of applications. By using PyCASP, application developers can capture a full-functioning application in 10-60× less lines-of-code compared to implementing the same application in a low-level language such as C++. In addition, using PyCASP's components, they are able to reuse significant amount of code, further improving productivity. Applications written using PyCASP are able to achieve, and often even exceed, the performance of the same application written in a low-level language.

Finally, we demonstrate that using SEJITS, we are able to port the applications to run on a variety of parallel hardware, ranging from Intel multi-core CPUs, NVIDIA GPUs and compute clusters. The application porting from CPU to GPU can be done without any application code change, but my modifying a configuration file. Porting an application to a cluster of machines requires an additional 20 lines of Python code. Applications written using PyCASP can scale up to large sets of audio data enabling the programmer to easily go from single-desktop, small subset experimentation to running the application on a cluster of parallel nodes using the full dataset. Combining all of the techniques and optimizations, our example applications are able to automatically achieve 50-1000× faster-than-real-time performance on both multi-core CPU and GPU platforms and 15.5× speedup on 16-node cluster of GPUs showing near-optimal scaling.

We now evaluate our results against the goals we set in the beginning of this work:

1. **Productivity**: We aimed for $10 - 100\times$ lines-of-code reduction compared to applications written in a low-level efficiency language. While it is difficult to evaluate the exact lines-of-code reduction, using our best estimates of the implementation the example applications, we conclude that we can achieve at least a $10-60\times$ reduction in the number of lines of code for the application implementation using Python and PyCASP compared to using low-level languages such as C++.

2. **Efficiency** We aimed to obtain within $30-50\%$ of hand-coded performance and within an order of magnitude faster performance than pure Python code. For PyCASP's components, we showed that we can not only obtain efficiency within the $30 - 50\%$ factor of hand-coded performance, but we can often beat the hand-coded implementation by choosing the best code variant for a particular computation (illustrated with the GMM training component). When evaluating example applications, we showed that using PyCASP we can achieve state-of-the-art performance and enable applications to run orders of magnitude faster than real-time, which can also beat the performance of equivalent hand-coded applications.

3. **Portability** We aimed to demonstrate the same application running on multi-core CPU, a GPU and a computer cluster without significant application code change. We have shown that our applications can run on a variety of GPU and multi-core CPU platforms without application code change but by using the configuration file to specify the hardware backends. We have also shown, using the video event detection application, that by adding 20 lines of Python code, we were able to scale our applications to run on a cluster of compute nodes, enabling portability from a single node to a compute cluster.

4. **Scalability** We intended to show application scaling from one node to a cluster of nodes, using our same framework, without significant programming effort. We have demonstrated this capability using the video event detection example application. We showed that by adding 20 lines of Python code, we enabled the application to scale to a larger dataset and run on a computer cluster achieving a speedup of $15.5\times$ on a 16-node GPU cluster.

Thus, we have shown that our approach achieves the goals of bridging the implementation gap, application programmer productivity and application efficiency and portability; however, there are some drawbacks and limitations of our approach that we need to analyze in order to fully understand the applicability of this technology.

9.2 Discussion

9.2.1 Components & scope

Software frameworks have to trade-off efficiency of the generated code with the generality of the computations they support. It is difficult to enable both generality and efficiency, and thus, most frameworks fall closer to one of the ends of this spectrum. We use the pattern-oriented approach to define the scope of PyCASP. Specifically, we use application patterns to understand the types of computations and algorithms our framework will support, and structural patterns for understanding how the computations are composed together into applications. Thus, PyCASP presents a *tall-skinny* framework that focuses deeply on one application domain and specializes the specific algorithms used in this particular domain. PyCASP cannot support new, "undiscovered" algorithms by definition - we design PyCASP's components to support existing algorithms and allow for their modification within particular constraints. By focusing on specializing entire instances of application patterns (i.e. the algorithms in the particular application domain), PyCASP is able to achieve high levels of efficiency of the generated code, at the expense of generality.

In contrast, frameworks such as Copperhead [14] and Theano [7], implement and specialize general, lower-level constructs, such as data-parallel operations or loop-nest optimizations. Domain experts can construct new algorithms using the lower-level constructs of such frameworks, which then get specialized and executed on parallel hardware. These frameworks have a *wider* scope, i.e. they support a subset of general computations that can be specialized to run on parallel hardware from application code. At the expense of this generality, applications developed using these frameworks can achieve limited efficiency (see [14] for more details).

9.2.2 Flexibility & customization

As discussed in Chapter 5, PyCASP is designed to support customizable components in addition to library components. Currently, PyCASP contains library components, which specialize specific algorithms and allow for a set of pre-defined customization points. The application developer chooses particular components of PyCASP to use and sets the customization points based on the application requirements. For example, the application programmer can choose to use Support Vector Machines (SVMs) for content classification, and then choose between several kernel functions for the SVM component. These customizations give some flexibility to application programmers, but still restrict the capabilities of the framework to support only the pre-defined computations and customizations. Thus, this restriction *limits the flexibility of PyCASP* and restricts its use cases to only the ones supported by the library components.

In some scenarios, the framework users may want to specify the functionality of components themselves instead of relying on fixed functionality or pre-existing components. To enable higher application developer flexibility, PyCASP is designed to support customizable components. Customizable components allow for more user input and customization of algorithms and computations. Customizable components can be implemented as embedded domain-specific-languages (eDSLs) using the Asp framework to lower high-level (Python) code to low-level efficiency code. Application programmers can then specify the exact functionality of the component by defining the internals of specific functions. Asp and SEJITS will then "translate" these functions into low-level code and plug that code in with the remainder of the specializer-generated code. Customizable components can enable much higher *flexibility* for PyCASP, as the application writer can specify the exact algorithm for each application, without relying on previously-implemented approaches. Currently, PyCASP does not contain customizable components, and thus does not allow for significant application programmer flexibility. In order to enable PyCASP to be a fully-flexible software environment that allows application developers to experiment with algorithms, customizable components are a requirement and the natural next step in continuing development of PyCASP.

9.2.3 Composition

In order to enable efficient composition of computations in applications, we restrict the scope of PyCASP to a specific application domain. We restrict our focus on composing PyCASP's components, i.e. a high-level composition of machine learning and signal processing algorithms. These compositions can be analyzed using structural patterns. Namely, in this work, we discover that we can use the three structural patterns, Pipe-and-Filter, Iterator and MapReduce to express the ways components are composed together in audio content analysis applications. Based on our structural pattern analysis, we discover that composing PyCASP's component at this high level corresponds to data structure sharing and reuse. Thus, we can optimize application performance by removing redundant data structure allocation and copy calls from our generated code. This optimization can result in up to 30% reduction in execution time (see Chapter 8 for experimental results).

Our approach for analyzing and optimizing composition of computations gives a relatively small insight into the general composition problem. In order to enable efficient composition of PyCASP's components, we 1) restricted the scope of our framework to one application domain, 2) restricted our composition optimizations to only apply to composing two components, and 3) pre-defined the data structure format that will be used by both components (in our case, the data that is shared across components are row-major dense matrices). Application writers must call two components that they want to be composed one right after the other, passing the data structures directly between the two components. If there

are any modifications to the shared data structure between the two component calls, the composition is not possible as we must deallocate and reallocate the new updated data structure between the two component calls. Thus, we setup a strict composition protocol in order to optimize compositions in our framework. The composition problem becomes much harder once we move to composing general computations, composing multiple computation calls instead of just two, as well as when we move to more complex data structures (for example graphs or sparse matrices) that have many alternative formats. Thus, it is not clear whether the methodology we describe here is applicable to more general composition scenarios than the ones we focus on in this work.

9.2.4 Uses of PyCASP

While PyCASP is still in its initial phases of development, the framework is already being used by speech recognition researchers at the International Computer Science Institute (ICSI) to perform fast speaker diarization and video event detection tasks. The speaker diarization application, written using PyCASP, is also being used for meeting diarization by the Intel Corporation. The GMM training component is employed in other multimedia and audio research projects by PhD students at UC Berkeley and researchers at audio analysis companies such as Gracenote [1]. By collaborating with researchers, we have iterated on a better API for PyCASP's components as well as included detailed documentation of specific functionality of the framework. In order to develop PyCASP further, researcher and application domain expert collaboration is essential to identify new use cases, fix bugs and develop good documentation. Thus, we hope to continue the collaboration with application domain experts and encourage more researchers to use our software.

9.3 Future Work

PyCASP illustrates an initial effort in a larger project of using selected embedded specialization to create productive and efficient software environments for a variety of application areas [2]. Continuation of this work is supported by a full NSF grant,

[1] http://www.gracenote.com/

[2] Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Partially supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20066. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or

and thus, we expect a significant increase in the magnitude and development efforts of this project or other projects based on PyCASP. PyCASP illustrates the use of key elements such as pattern-oriented design, SEJITS for component specialization and customization, and component composition, that can be reused in designing productive programming environments for other application areas.

Future work for PyCASP includes completing the PyCASP framework with more components, corresponding to the application patterns we mined in the design phase of this work. Other machine learning components such as neural-network training and classification are especially important to add as well as feature extraction signal processing components. Another important trajectory is to add *customizable* components to PyCASP to enable higher application programmer flexibility and customization of algorithms. By adding more components, we can further explore the breadth and efficiency of PyCASP as well as analyze new composition points and optimizations.

Other exciting directions for future work include adding more cluster computing functionality to PyCASP and integrating more auto-tuning technology. Currently, PyCASP supports basic calls to map and reduce functions by exposing Hadoop through its Python interface. Enabling more complex MapReduce applications and their integration with PyCASP's components and composition mechanisms can provide productive software solutions to large-scale data analysis applications (not only in audio content analysis). Adding auto-tuning and machine learning to perform automatic code-tuning and optimizations can further improve performance of applications and enable higher application portability.

Finally, we hope that more domain experts and researchers use, extend contribute to the PyCASP project, and we would like to encourage the research community to do so. PyCASP is available at: https://github.com/egonina/pycasp.

---

endorsement, either expressed or implied, of IARPA, DOI/NBC, or the U.S. Government.

Bibliography

[1] Xavier Amatriain, Maarten De Boer, and Enrique Robledo. *CLAM: An OO Framework for Developing Audio and Music Applications*. 2002.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. "LAPACK: a portable linear algebra library for high-performance computers". In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 2–11. ISBN: 0-89791-412-0. URL: http://dl.acm.org/citation.cfm?id=110382.110385.

[3] Alexandr Andoni. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In: *In FOCSÕ06*. IEEE Computer Society, 2006, pp. 459–468.

[4] X Anguera, Simon Bozonnet, Nicholas W D Evans, Corinne Fredouille, G Friedland, and O Vinyals. "Speaker diarization : A review of recent research". In: *IEEE Transactions On Acoustics Speech and Language Processing" (TASLP), special issue on "New Frontiers in Rich Transcription"* 20 (2 2012), pp. 356–370.

[5] David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant. *Numerical Python*. UCRL-MA-128569. Lawrence Livermore National Laboratory. Livermore, CA, 1999.

[6] Eric Battenberg and David Wessel. "Accelerating Non-Negative Matrix Factorization for Audio Source Separation on Multi-Core and Many-Core Architectures." In: *ISMIR*. Ed. by Keiji Hirata, George Tzanetakis, and Kazuyoshi Yoshii. International Society for Music Information Retrieval, 2009, pp. 501–506. ISBN: 978-0-9813537-0-8. URL: http://dblp.uni-trier.de/db/conf/ismir/ismir2009.html#BattenbergW09.

[7] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the Python for Scientific Computing Conference*

(SciPy). Oral. Austin, TX, June 2010. URL: http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf.

[8] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. "The Million Song Dataset". In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. 2011.

[9] C. M. Bishop. *Neural Networks for Pattern Recognition*. New York: Oxford Univ. Press, 1995.

[10] 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)". In: *ACM Transactions on Mathematical Software* 28 (2001), pp. 135–151.

[11] B. Boehm. "Managing software productivity and reuse". In: *Computer* 32.9 (1999), pp. 111–113. ISSN: 0018-9162. DOI: 10.1109/2.789755.

[12] W.M. Campbell, D.E. Sturim, D.A. Reynolds, and A. Solomonoff. "SVM Based Speaker Verification using a GMM Supervector Kernel and NAP Variability Compensation". In: *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*. Vol. 1. 2006, p. I. DOI: 10.1109/ICASSP.2006.1659966.

[13] Jean Carletta. "Unleashing the killer corpus: experiences in creating the multi-everything AMI Meeting Corpus." In: *Language Resources and Evaluation* 41.2 (May 11, 2010), pp. 181–190. URL: http://dblp.uni-trier.de/db/journals/lre/lre41.html#Carletta07.

[14] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. *Copperhead: Compiling an Embedded Data Parallel Language*. Tech. rep. UCB/EECS-2010-124. EECS Department, University of California, Berkeley, 2010. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-124.html.

[15] Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, and Kurt Keutzer. "Efficient, high-quality image contour detection". In: *Computer Vision, 2009 IEEE 12th International Conference on*. 2009, pp. 2381 –2388. DOI: 10.1109/ICCV.2009.5459410.

[16] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. "Fast support vector machine training and classification on graphics processors". In: *Proceedings of the 25th international conference on Machine learning*. ICML '08. Helsinki, Finland: ACM, 2008, pp. 104–111. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390170. URL: http://doi.acm.org/10.1145/1390156.1390170.

[17] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. "SEJITS: Getting Productivity And Performance With Selective Embedded JIT Specialization". In: *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*. Raleigh, NC, 2009.

[18] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. "A domain-specific approach to heterogeneous parallelism". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 35–46. ISBN: 978-1-4503-0119-0. DOI: http://doi.acm.org/10.1145/1941553.1941561. URL: http://doi.acm.org/10.1145/1941553.1941561.

[19] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, 27:1–27:27.

[20] Christophe Charbuillet, Damien Tardieu, and Geoffroy Peeters. "GMM Supervector for Content Based Music Similarity". In: *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx 2011)*. 2011.

[21] Sourish Chaudhuri, Mark Harvilla, and Bhiksha Raj. "Unsupervised Learning of Acoustic Unit Descriptors for Audio Content Representation and Classification". In: *11th Annual Conference of the International Speech Communication Association (InterSpeech)*. 2011.

[22] J.C. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi. "Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation". In: *HPCMP Users Group Conference, 2006*. 2006, pp. 429 – 434. DOI: 10.1109/HPCMP-UGC.2006.55.

[23] J. Chong, E. Gonina, Y. Yi, and K. Keutzer. "A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit". In: *10th Annual Conference of the International Speech Communication Association (InterSpeech)*. 2009.

[24] J. Chong, Y. Yi, N. R. Satish A. Faria, and K. Keutzer. "Data-parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors". In: *Proc. Intl. Workshop on Emerging Applications and Manycore Architectures*. 2008.

[25] Jike Chong. "Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors". PhD thesis. EECS Department, University of California, Berkeley, 2010. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-158.html.

[26] Jike Chong, Gerald Friedland, Adam Janin, Nelson Morgan, and Chris Oei. "Opportunities and challenges of parallelizing speech recognition". In: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. HotPar'10. Berkeley, CA: USENIX Association, 2010, pp. 2–2. URL: http://portal.acm.org/citation.cfm?id=1863086.1863088.

[27] *Cilk 5.4.6 Reference Manual*. Version 5.4.6. Intel. URL: http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

[28] *CodePy: a C/C++ metaprogramming toolkit for Python*. http://mathema.tician.de/software/codepy.

[29] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-53086-4.

[30] Henry Cook, Ekaterina Gonina, Shoaib Kamil, Gerald Friedland, David Patterson, and Armando Fox. "Cuda-level Performance with Python-level Productivity for Gaussian Mixture Model Applications". In: *USENIX Workshop on Hot Topics in Parallelism (HotPar)*. 2011.

[31] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411. URL: http://dx.doi.org/10.1023/A:1022627411411.

[32] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[33] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[34] P. R. Dixon, T. Oonishi, and S. Furui. "Fast Acoustic Computations Using Graphics Processors". In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*. Taipei, Taiwan, 2009.

[35] Phaklen EhKan, Timothy Allen, and Steven F. Quigley. "FPGA implementation for GMM-based speaker identification". In: *Int. J. Reconfig. Comput.* 2011 (2011), 3:1–3:8. ISSN: 1687-7195. DOI: http://dx.doi.org/10.1155/2011/420369. URL: http://dx.doi.org/10.1155/2011/420369.

[36] B. Elizalde, G. Friedland, H. Lei, and A. Divakaran. "There is No Data Like Less Data: Percepts for Video Concept Detection on Consumer-Produced Media". In: *1st ACM Workshop on Audio and Multimedia Methods for Large-Scale Video Analysis, ACM Multimedia*. Nara, Japan, 2012.

[37] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. "Working Set Selection Using Second Order Information for Training Support Vector Machines". In: *J. Mach. Learn. Res.* 6 (Dec. 2005), pp. 1889–1918. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1046920.1194907.

[38] Pascal Ferraro, Pierre Hanna, Laurent Imbert, and Thomas Izard. "Accelerating Query-by-Humming on GPU." In: *ISMIR*. Ed. by Keiji Hirata, George Tzanetakis, and Kazuyoshi Yoshii. International Society for Music Information Retrieval, 2009, pp. 279–284. ISBN: 978-0-9813537-0-8. URL: http://dblp.uni-trier.de/db/conf/ismir/ismir2009.html#FerraroHII09.

[39] Gerald Friedland, Chuohao Yeo, and Hayley Hung. "Dialocalization: Acoustic speaker diarization and visual localization as joint optimization problem". In: *ACM Trans. Multimedia Comput. Commun. Appl.* 6 (4 2010), 27:1–27:18. ISSN: 1551-6857. DOI: http://doi.acm.org/10.1145/1865106.1865111. URL: http://doi.acm.org/10.1145/1865106.1865111.

[40] Gerald Friedland and Oriol Vinyals. "Live speaker identification in conversations". In: *Proceeding of the 16th ACM international conference on Multimedia*. MM '08. Vancouver, British Columbia, Canada: ACM, 2008, pp. 1017–1018. ISBN: 978-1-60558-303-7. DOI: http://doi.acm.org/10.1145/1459359.1459558. URL: http://doi.acm.org/10.1145/1459359.1459558.

[41] Gerald Friedland, Jike Chong, and Adam Janin. "Parallelizing Speaker-Attributed Speech Recognition for Meeting Browsing". In: *Proceedings of the 2010 IEEE International Symposium on Multimedia*. ISM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 121–128. ISBN: 978-0-7695-4217-1. DOI: http://dx.doi.org/10.1109/ISM.2010.26. URL: http://dx.doi.org/10.1109/ISM.2010.26.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "Design patterns: Abstraction and reuse of object-oriented design". In: *ECOOPÕ93ÑObject-Oriented Programming* (1993), pp. 406–431.

[43] E. Gonina, G. Friedland, H. Cook, and K. Keutzer. "Fast speaker diarization using a high-level scripting language". In: *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*. 2011, pp. 553 –558. DOI: 10.1109/ASRU.2011.6163887.

[44] Ekaterina Gonina, Anitha Kannan, John Shafer, and Mihai Budiu. "Parallelizing large-scale data processing applications with data skew: a case study in product-offer matching". In: *Proceedings of the second international workshop on MapReduce and its applications*. MapReduce '11. San Jose, California, USA: ACM, 2011, pp. 35–42. ISBN: 978-1-4503-0700-0. DOI: 10.1145/1996092.1996101. URL: http://doi.acm.org/10.1145/1996092.1996101.

[45] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan MassŮ, Edward Seidel, and John Shalf. "The Cactus framework and toolkit: Design and applications". In: *In High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference*. Springer, pp. 26–28.

[46] V. W. Gregory. *Programmers tool chest: the OpenCV library*. Dr. Dobbs Journal. 2000.

[47] Eitan Grinspun, Petr Krysl, and Peter SchrŽder. "CHARMS: A Simple Framework for Adaptive Simulation". In: *ACM Transactions on Graphics*. ACM Press, 2002, pp. 281–290.

[48] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1994, pp. xx + 307. ISBN: 0-262-57104-8.

[49] *HMM Toolkit Web Page*.

[50] *http://rubygems.org/gems/hadoop-papyrus*. URL: http://rubygems.org/gems/hadoop-papyrus.

[51] *http://www.cascading.org*. URL: http://www.cascading.org.

[52] P. Hudak and M.P. Jones. *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*. Research Report YALEU/DCS/RR-1049. New Haven, CT: Department of Computer Science, Yale University, 1994.

[53] David Imseng and Gerald Friedland. "Robust Speaker Diarization for Short Speech Recordings". In: *Proceedings of the IEEE workshop on Automatic Speech Recognition and Understanding*. Merano, Italy, Dec. 2009, pp. 432–437.

[54] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura. "Parallel LVCSR Algorithm for Cellphone-oriented Multicore Processors". In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*. Toulouse, France, 2006.

[55] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[56] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '93. Washington, D.C., USA: ACM, 1993, pp. 91–108. ISBN: 0-89791-587-9. DOI: 10.1145/165854.165874. URL: http://doi.acm.org/10.1145/165854.165874.

[57] S Kamil, D Coetzee, and A Fox. "Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-in-Time Specialization". In: *Python for Scientific Computing Conference (SciPy)*. 2011.

[58] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. "Portable parallel performance from sequential, productive, embedded domain-specific languages". In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 303–304. ISSN: 0362-1340. DOI: 10.1145/2370036.2145865. URL: http://doi.acm.org/10.1145/2370036.2145865.

[59] Shoaib Ashraf Kamil. "Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages". PhD thesis. EECS Department, University of California, Berkeley, 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html.

[60] Thomas Karcher and Victor Pankratius. "Run-time automatic performance tuning for multicore applications". In: *Proceedings of the 17th international conference on Parallel processing - Volume Part I*. Euro-Par'11. Bordeaux, France: Springer-Verlag, 2011, pp. 3–14. ISBN: 978-3-642-23399-9. URL: http://dl.acm.org/citation.cfm?id=2033345.2033348.

[61] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. "Improvements to Platt's SMO Algorithm for SVM Classifier Design". In: *Neural Comput.* 13.3 (Mar. 2001), pp. 637–649. ISSN: 0899-7667. DOI: 10.1162/089976601300014493. URL: http://dx.doi.org/10.1162/089976601300014493.

[62] Kurt Keutzer and Tim Mattson. "A design pattern language for engineering (parallel) software". In: *Intel Technology Journal* 13 (2010), p. 4.

[63] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. "A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects". In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ParaPLoP'10. Carefree, Arizona: ACM, 2010, 9:1–9:8. ISBN: 978-1-4503-0127-5. DOI: 10.1145/1953611.1953620. URL: http://doi.acm.org/10.1145/1953611.1953620.

[64] SADANORI KONISHI and GENSHIRO KITAGAWA. "Generalised information criteria in model selection". In: *Biometrika* 83.4 (1996), pp. 875–890. DOI: 10.1093/biomet/83.4.875. eprint: http://biomet.oxfordjournals.org/content/83/4/875.full.pdf+html. URL: http://biomet.oxfordjournals.org/content/83/4/875.abstract.

[65] Anthony Kosner. *YouTube Turns Seven Today, Now Uploads 72 Hours of Video Per Minute*. May 2012. URL: http://www.forbes.com/sites/anthonykosner/2012/05/21/youtube-turns-seven-now-uploads-72-hours-of-video-per-minute/.

[66] NSL Kumar, S. Satoor, and I. Buck. "Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA". In: *11th IEEE International Conference on High Performance Computing and Communications, 2009. HPCC'09*. 2009, pp. 103–109.

[67] Bil Lewis and Daniel J. Berg. *Multithreaded programming with Pthreads*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-680729-1.

[68] Lie Lu and A. Hanjalic. "Audio Keywords Discovery for Text-Like Audio Content Analysis and Retrieval". In: *Multimedia, IEEE Transactions on* 10.1 (2008), pp. 74 –85. ISSN: 1520-9210. DOI: 10.1109/TMM.2007.911304.

[69] *Mako Templates for Python*. http://www.makotemplates.org/.

[70] Timothy G Mattson, Beverly A Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2005.

[71] Christophe Rhodes Michael Casey and Malcolm Slaney. "Analysis of Minimum Distances in High-Dimensional Musical Spaces". In: *IEEE Trans. Audio Speech Lang. Process* 16 (2008), 1015Ð1028.

[72] Volodymyr Mnih. *Cudamat: a CUDA-based Matrix Class for Python*. Tech. rep. 2009.

[73] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[74] Mark Murphy, Kurt Keutzer, Shreyas Vasanawala, and Michael Lustig. "Clinically feasible reconstruction time for L1-SPIRiT parallel imaging and compressed sensing MRI". In: *Proceedings of the ISMRM Scientific Meeting & Exhibition*. 2010, p. 4854.

[75] *NVIDIA CUDA Programming Guide*. Version 3.2. NVIDIA Corporation. 2010. URL: http://www.nvidia.com/CUDA.

[76] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1099–1110. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376726. URL: http://doi.acm.org/10.1145/1376616.1376726.

[77] *OpenCL 1.1 Specification*. Version 1.1. Khronos Group. 2010. URL: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[78] *OpenMP Application Programming Interface*. Version 3.0. OpenMP. 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[79] Mike Owens. *The Definitive Guide to SQLite (Definitive Guide)*. Berkely, CA, USA: Apress, 2006. ISBN: 1590596730.

[80] Andrew D. Pangborn. "Scalable Data Clustering using GPUs". MA thesis. Rochester Institute of Technology, 2010, pp. 1–124.

[81] S. Phillips and A. Rogers. "Parallel Speech Recognition". In: *Intl. Journal of Parallel Programming* 27.4 (1999), pp. 257–288.

[82] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, and Google Inc. "Interpreting the Data: Parallel Analysis with Sawzall". In: *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, pp. 227–298.

[83] John C. Platt. "Advances in kernel methods". In: ed. by Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola. Cambridge, MA, USA: MIT Press, 1999. Chap. Fast training of support vector machines using sequential minimal optimization, pp. 185–208. ISBN: 0-262-19416-3. URL: http://dl.acm.org/citation.cfm?id=299094.299105.

[84] L. Prechelt. "An empirical comparison of seven programming languages". In: *Computer* 33.10 (2000), pp. 23 –29. ISSN: 0018-9162. DOI: 10.1109/2.876288.

[85] *PyUBLAS: a seamless glue layer between Numpy and Boost.Ublas*. http://mathema.tician.de/software/pyublas.

[86] L.R. Rabiner. "A tutorial on hidden Markov models and selected applications in speech recognition". In: *Proceedings of the IEEE* 77.2 (Feb 1989), pp. 257–286. ISSN: 0018-9219. DOI: 10.1109/5.18626.

[87] Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, and Anping Liu. "Magellan: experiences from a science cloud". In: *Proceedings of the 2nd international workshop on Scientific cloud computing*. ScienceCloud '11. San Jose, California, USA: ACM, 2011, pp. 49–58. ISBN: 978-1-4503-0699-7. DOI: 10.1145/1996109.1996119. URL: http://doi.acm.org/10.1145/1996109.1996119.

[88] K.R. Rao and N. Ahmed. "Orthogonal transforms for digital signal processing". In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '76*. Vol. 1. 1976, pp. 136–140. DOI: 10.1109/ICASSP.1976.1170121.

[89] M. Ravishankar. *Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition*. 1993.

[90] James Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[91] D.A. Reynolds and P. Torres-Carrasquillo. "Approaches and applications of audio diarization". In: *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*. Vol. 5. 2005, v/953 –v/956 Vol. 5. DOI: 10.1109/ICASSP.2005.1416463.

[92] Douglas A. Reynolds, Thomas F. Quatieri, and Robert B. Dunn. "Speaker verification using Adapted Gaussian mixture models". In: *Digital Signal Processing*. 2000, p. 2000.

[93] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations - Version 2*. 1994.

[94] Lauri Savioja, Vesa Valimaki, and Julius O. Smith. "Audio Signal Processing Using Graphics Processing Units". In: *J. Audio Eng. Soc* 59.1/2 (2011), pp. 3–19. URL: http://www.aes.org/e-lib/browse.cfm?elib=15772.

[95] Walt Scacchi. *Understanding Software Productivity*. 1994.

[96] *scikits.learn: machine learning in Python*. http://scikit-learn.sourceforge.net/index.html.

[97] Mary Shaw and David Garlan. "Software architecture: perspectives on an emerging discipline". In: (1996).

[98] R. Strebelow, M. Tribastone, and C. Prehofer. "Performance Modeling of Design Patterns for Distributed Computation". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. Aug. Pp. 251–258. DOI: 10.1109/MASCOTS.2012.37.

[99] Bor-Yiing Su. "Parallel Application Library for Object Recognition". PhD thesis. EECS Department, University of California, Berkeley, 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-199.html.

[100] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. "OptiML: an implicitly parallel domainspecific language for machine learning". In: *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*. 2011.

[101] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. "Scalable Collaborative Filtering Approaches for Large Recommender Systems". In: *J. Mach. Learn. Res.* 10 (June 2009), pp. 623–656. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1577069.1577091.

[102] Tijmen Tieleman and Tijmen Tieleman. *Gnumpy: an easy way to use GPU boards in Python*. 2010.

[103] Nicolas Tsingos. "Using Programmable Graphics Hardware for Acoustics and Audio Rendering". In: *Audio Engineering Society Convention 127*. Oct. 2009. URL: http://www.aes.org/e-lib/browse.cfm?elib=15045.

[104] G. Tzanetakis. *MARSYAS SUBMISSIONS TO MIREX 2007*. URL: http://marsyas.info/.

[105] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels". In: *Journal of Physics: Conference Series* 16.1 (2005), pp. 521+. ISSN: 1742-6596. DOI: 10.1088/1742-6596/16/1/071. URL: http://dx.doi.org/10.1088/1742-6596/16/1/071.

[106] R. Clint Whaley and Jack Dongarra. "Automatically Tuned Linear Algebra Software". In: *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. CD-ROM Proceedings. 1999.

[107] Tom White. *Hadoop: The Definitive Guide*. Ed. by Mike Loukides. first edition. O'Reilly, 2009. URL: http://oreilly.com/catalog/9780596521981.

[108] C. Wooters and M. Huijbregts. "The ICSI RT07s Speaker Diarization System". In: Baltimore, Maryland, 2007, pp. 509–519.

[109] Richard Xia, Tayfun Elmas, Shoaib Ashraf Kamil, Armando Fox, and Koushik Sen. *Multi-level Debugging for Multi-stage, Parallelizing Compilers*. Tech. rep. UCB/EECS-2012-227. EECS Department, University of California, Berkeley, 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-227.html.

[110] K. You, Y. Lee, and W. Sung. "OpenMP-based Parallel Implementation of a Continous Speech Recognizer on a Multi-core System". In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*. Taipei, Taiwan, 2009.

[111] 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*. 6. 2009, pp. 124–135.

[112] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, òlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In: *OSDI*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, Jan. 7, 2009, pp. 1–14. ISBN: 978-1-931971-65-2. URL: `http://dblp.uni-trier.de/db/conf/osdi/osdi2008.html#YuIFBEGC08`.

[113] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

[114] Mingjian Zhang. "Blind Source Separation Using Generalized Singular Value Decomposition". In: *Information Science and Engineering (ICISE), 2009 1st International Conference on*. 2009, pp. 509–511. DOI: `10.1109/ICISE.2009.364`.