Parallel Application Library for Object Recognition

Bor-Yiing Su

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-199
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-199.html

September 27, 2012

Parallel Application Library for Object Recognition

by

Bor-Yiing Su

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt W. Keutzer, Chair
Professor Jitendra Malik
Professor Sara McMains

Fall 2012

Parallel Application Library for Object Recognition

Abstract

Parallel Application Library for Object Recognition

by

Bor-Yiing Su

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt W. Keutzer, Chair

Computer vision research enables machines to understand the world. Humans usually interpret and analyze the world through what they see – the objects they capture with their eyes. Similarly, machines can better understand the world by recognizing objects in images. Object recognition is therefore a major branch of computer vision. To achieve the highest accuracy, state-of-the-art object recognition systems must extract features from hundreds to millions of images, train models with enormous data samples, and deploy those models on query images. As a result, these systems are computationally-intensive. In order to make such complicated algorithms practical to apply in real life, we must accelerate them on modern massively-parallel platforms.

However, parallel programming is complicated and challenging, and takes years to master. In order to help object recognition researchers employ parallel platforms more productively, we propose a parallel application library for object recognition. Researchers can simply call the library functions, and need not understand the technical details of parallelization and optimization. To pave the way for such a library, we perform pattern mining on 31 important object recognition systems, and conclude that 15 application patterns are necessary to cover the computations in these systems. In other words, if we support these 15 application patterns in our library, we can parallelize all 31 object recognition systems. In order to optimize any given application pattern in a systematic way, we propose using patterns and software architectures to explore the design space of algorithms, parallelization strategies, and platform parameters.

In this dissertation, we exhaustively examine the design space for three application patterns, and achieve significant speedups on these patterns – 280× speedup on the eigensolver application pattern, 12-33× speedup on the breadth-first-search graph traversal application pattern, and 5-30× speedup on the contour histogram application pattern. To improve the portability and flexibility of the proposed library, we also initiate the OpenCL for OpenCV project. This project aims to provide a collection of autotuners that optimize the performance of application patterns on many different parallel platforms. We have developed two autotuners in this project. *clSpMV* is an autotuner for sparse matrix vector multiplication (SpMV) computation – it tunes the representation of a sparse matrix and the corresponding SpMV kernel, and is 40% faster than the vendor-optimized parallel implementation. *clPaDi*

is an autotuner for the pair-wise distance computation application pattern – it allows users to customize their own distance functions, and finds the best blocking size for each function. *clPaDi* performs 320-650 giga floating point operations per second on modern GPU platforms. By employing these optimized functions in a state-of-the-art object recognition system, we have achieved 110-120× speedup compared to the original serial implementation. Now it takes only three seconds to identify objects in a query image – a much more practical and useful processing time. Our research makes it possible to deploy complicated object recognition algorithms in real applications.

With these encouraging results, we are confident that the methodology we illustrate in this dissertation is applicable to optimizing all application patterns. If we expand the parallel application library to support all 15 application patterns, the library will be a key toolkit for both existing and future object recognition systems.

To Wei-Hsuan

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BCSR	Blocked Compressed Sparse Row (sparse matrix format)
BDIA	Banded Diagonal (sparse matrix format)
BELL	Blocked ELLPACK/ITPACK (sparse matrix format)
BFS	Breadth-First-Search
CMP	Cocktail Matrix Partitioning
COO	Coordinate (sparse matrix format)
CSB	Compressed Sparse Block
CSR	Compressed Sparse Row (sparse matrix format)
CUDA	Compute Unified Device Architecture
DIA	Diagonal (sparse matrix format)
ELL	ELLPACK/ITPACK (sparse matrix format)
FFT	Fast Fourier Transform
FLOP	FLoating point OPeration
FLOPS	FLoating point OPerations per Second
FPPI	False Positive Per Image
GB	Giga Bytes
GFLOPS	Giga FLoating point OPerations per Second
gPb	Global Probability
HOG	Histogram of Oriented Gradients
HPC	High Performance Computing

ILSVRC	ImageNet Large Scale Visual Recognition Challenge
KCCA	Kernel Canonical Correlation Analysis
KKT	Karush-Kuhn-Tucker
KNN	K-Nearest Neighbor
MRI	Magnetic Resonance Imaging
NUMA	Non-Uniform Memory Access
OLOV	OpenCL for OpenCV
OPL	Our Pattern Language
OSKI	Optimized Sparse Kernel Interface
PaDi	Pair-wise Distance
PCA	Principal Component Analysis
SBELL	Sliced Blocked ELLPACK/ITPACK (sparse matrix format)
SELL	Sliced ELLPACK/ITPACK (sparse matrix format)
SIFT	Scale-Invariant Feature Transform
SIMD	Single Instruction Multiple Data
SMO	Sequential Minimal Optimization
SPMD	Single Program Multiple Data
SpMV	Sparse Matrix Vector multiplication
SVM	Support Vector Machine
TBB	Threading Building Blocks
TLB	Translation Lookaside Buffer
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word

Acknowledgments

I would like to acknowledge my sincere appreciation to my advisor, Kurt Keutzer, for his guidance and support throughout my time in Berkeley. Kurt is always passionate about demonstrating how parallelism can change the computing industry. Without his insight, I cannot tightly bind my research with practical problems. I am also grateful for his patience on coaching me in my presentations. It is Kurt who makes my graduate research successful. I would also like to thank Prof. Jitendra Malik and Prof. Sara McMains for serving on my dissertation committee and giving me important feedback.

I had four summer internships that helped me sharpen my parallel programming skills on a variety of different applications. I would like to thank Tom Spyrou, Tasneem Brutch, Calin Cascaval and Pradeep Dubey for being my mentors, bringing me challenging problems, and broadening my experience on different areas.

It is a privilege working with all the members in the PALLAS group. Thanks to Matt Moskewicz, Nadathar Satish, Jike Chong, Bryan Catanzaro, Narayanan Sundaram, Mark Murphy, Ekaterina Gonina, Chao-Yue Lai, Michael Anderson, David Sheffield and Patrick Li for making my research collaborative and interesting. Special thanks to Prof. James Demmel for his inputs in linear algebra related research. I would also like to thank Pablo Arbeláez, Michael Maire, Lubomir Bourdev, Subhransu Maji and Chunhui Gu for introducing me state-of-the-art computer vision applications.

Most importantly, I would like to thank my wife Wei-Hsuan Hsiung for her full support and understanding when I am pursuing my Ph.D. degree. I cannot concentrate on my research without her sacrifice and tolerance. Finally, I would like to thank my family: my parents Mu-Huan Su and Su-Cin Gao, and my sisters Ying-Ke Su and Huei-Ke Su. Their confidence and encouragement motivated me to study hard and work hard.

Chapter 1

Introduction

Computer vision is a research field that enables machines to see, understand, and analyze the real world. One straightforward usage of this technology is in robotics – if the robots have the same views and the same interpretations as human beings, then they can behave more like human beings. Another important application for this research is to search the world. As long as the machines can understand visual data, we can collect data about the world and ask machines to analyze and find objects in which we are interested. The easiest way for machines to get "snapshots" of the real world is by collecting images and videos. Therefore, most computer vision research focuses on image and video analysis.

When given an image or video, one way that human beings understand the context is by identifying objects inside the scenes. Similarly, if machines are able to recognize objects in scenes, they might be able to understand the contexts as well. Therefore, object recognition is a major branch of computer vision research.

Human beings are not born with the ability to recognize objects – we establish the ability to recognize objects by learning. Similarly, machines need to learn before they can identify objects. The object recognition problem is therefore solved in two stages: a training stage that trains machines to be familiar with objects, and a deployment stage that asks machines to recognize objects from images or videos. In the training stage, for each object, a collection of example images or videos is provided containing instances of the object. Special features are extracted from the example images or videos to represent the characteristics of the object instances. Based on the features of the example object instances, a model is built to recognize the object. In the deployment stage, the features are extracted and plugged into that model. The model can then decide whether instances of the object are found in a given image or video.

A sense of the computational challenge at the training stage can be conveyed by considering the following example based on some reasonable assumptions: we are interested in only 1000 objects; 1000 instances of example images are enough to build an accurate model of each object; each example image is only 1 mega-pixel in size; we need only 1000 floating point operations (FLOPs) to extract the features on each pixel that summarizes the contents of the image; the dimension of the feature vector is 100; and 10000 FLOPs per feature vector element are enough to build the recognition model of the example in-

stances of an object. The computational cost for the feature extraction for this example is thus 1000(objects)×1000(examples)×1000000(pixels)×1000(FLOPs)= 10^{15} FLOPs. The computational cost for the model building is 1000(objects)×1000(examples)×100(vector dimension)×10000(FLOPs)= 10^{12} FLOPs. The total computational cost for the training stage is therefore about 10^{15} FLOPs. On a 1GHz CPU, this computation takes at least 10^6 seconds – 11.5 days.

Regarding the computational challenge at the deployment stage, object recognition technology is deployed in two different scenarios: the service provider (the cloud), and the end user (the client). The cost of feature extraction is similar to that in the training stage. Assume the classification computation takes 1000 FLOPs per feature vector element. For the first scenario, the YouTube team (for example) wants to recognize whether the videos uploaded in one minute contain any of the 1000 objects of interest. According to official YouTube statistics [109], 60 hours of videos are uploaded every minute. Assuming a frame-rate per second of 30, the cost of feature extraction is 60×60×60(seconds)×30(frames) ×1000000(pixels)×1000(FLOPs) = 6.48×10^{15} FLOPs. The cost of feature extraction is 60×60×60(seconds)×30(frames)×100(vector dimension)×1000(FLOPs)= 6.48×10^{11} FLOPs. The total computational cost is therefore about 6×10^{15} FLOPs. On a 1GHz CPU, this would take at least 70 days. For the second scenario, an end user might deploy the technology on a single image. The computational cost is 1000000(pixels)×1000(FLOPs)+100(vector dimension)×1000(FLOPs) = 1.0001×10^9. On a 1GHz CPU, this takes at least 1 second. On a less powerful mobile platform, it might take more than 20 seconds.

Clearly, the computational costs of both the training stage and the deployment stage are enormous. With the desire to improve recognition quality, analyze more images and videos, or process images and videos with higher resolution, the computational challenge becomes even larger.

Historically, application developers generally expect that computational challenges can be solved by CPU frequency scaling. According to Moore's Law [93], the number of transistors on a single chip doubles every 18 months. This prediction is based on the idea of reducing the dimension of transistors by 30% every generation. This decrease in transistor dimensions results in reducing the delay by 30%. Thus, the operating frequency can be 1.4× higher, and the number of computations performed in the same amount of time increases. This steady improvement is the low-hanging fruit that application developers can get without any coding effort. According to Figure 1.1, this expectation has been quite true from 1985 to 2004. During this period, CPU frequency scaled at an exponential rate. However, in 2004 the frequency scaling line stopped at around 3GHz, and since then has not exceeded that upper-bound. Moore's Law [93] still scales as expected, but we cannot find enough power to operate all transistors at a higher frequency. This is called the `Power Wall` [9].

The power wall puts a hard limit on uniprocessor performance. This restriction has driven the computer industry towards the parallel era. Moore's Law [93] still acts as the highest principle for processor manufacturers; however, instead of reducing the latency of the processor, hardware architects put multiple processors on a single chip to increase overall throughput. Although the frequency scaling trend comes to an end, the parallelism scaling trend begins.

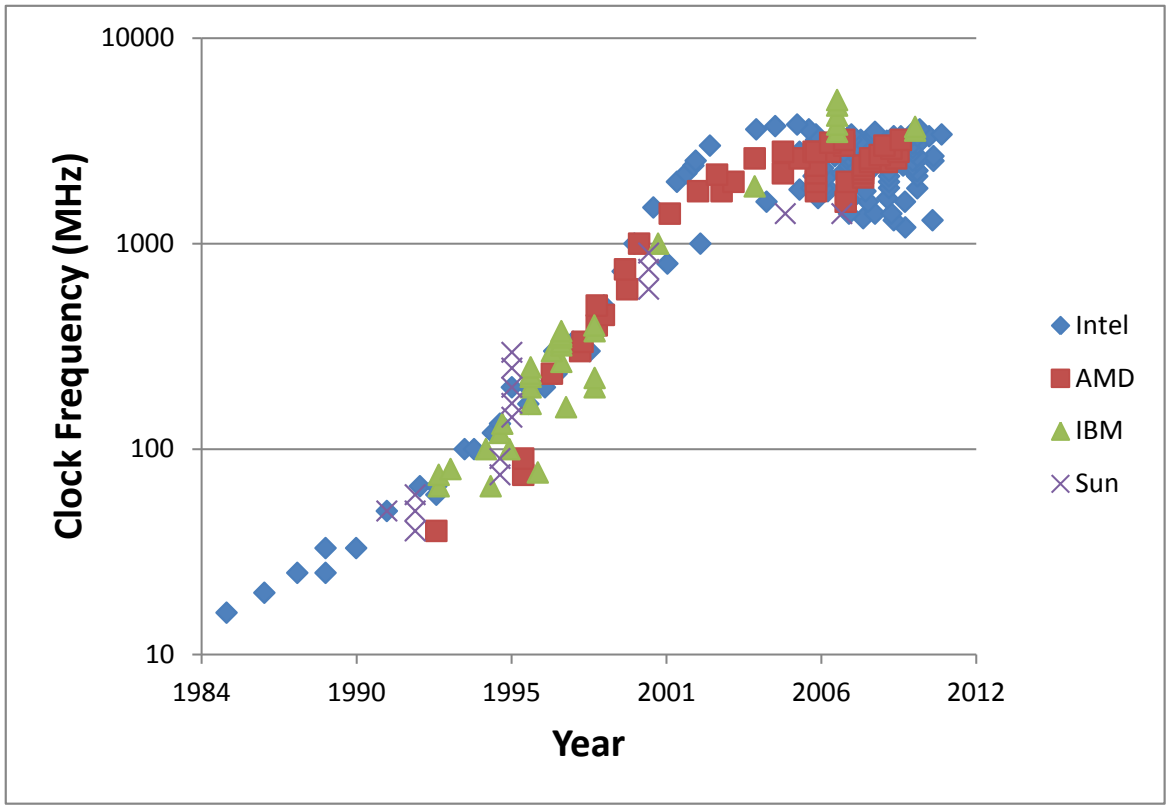

Figure 1.1: Processor frequency scaling over years. Data from Danowitz et al. [34].

A new hope for solving the computational challenge of the object recognition problem is to parallelize the algorithms and execute them on modern parallel platforms. However, parallel programming is totally different from serial programming. Without proper training, it is very hard to write correct, fast, and scalable parallel programs. As a result, there is an enormous gap between application developers and expert parallel programmers. Application developers are familiar with the algorithms used in the applications, but they lack the knowledge of parallelizing the algorithms on parallel platforms. On the other hand, expert parallel programmers are familiar with optimizing programs on parallel platforms, but do not know what computations are important for developing the applications. This is called the `implementation gap` [102, 28, 25], and will be further discussed in Section 2.3.

In order to bridge the implementation gap, we propose using patterns to architect the applications. Patterns capture recurring computations in applications; by identifying application patterns, we can identify the most important computations in the application. An architecture of an application is a hierarchical composition of patterns, and describes the organization of the application. Once the patterns are available, we can compose an application by enforcing a control flow following the software architecture. The implemen-

tation gap can be bridged by having a pattern list of the target application. The expert parallel programmers can then focus on parallelizing and optimizing this pattern list, and the application developers can compose the patterns following their software architectures.

The task of parallelizing and optimizing an application pattern on a parallel platform is not trivial. We must understand the algorithms used in the application pattern, and the underlying parallel hardware platform. To parallelize and optimize the selected application patterns, we propose using patterns to guide the process of design space exploration. There are three layers of design space to explore when parallelizing and optimizing an application pattern. The first is the algorithm layer. Sometimes an application pattern can be computed by different algorithms. The second layer is the parallelization strategy layer. Given an algorithm, there might be different ways to parallelize the algorithm. The third layer is the platform parameter layer. Given a parallelization strategy, sometimes we need to adjust the parameters to find the best mapping to the underlying hardware platform. By identifying patterns in the algorithm and parallelization strategy layers, we can develop initial insight based on the trade-offs among different patterns. This will help reduce the design space at an early stage. The remaining design space can be explored by an autotuner. Any application pattern can be parallelized and optimized with this approach. Gries et al. uses a similar strategy to explore the design space of developing application-specific instruction-set processors [59].

To bridge the implementation gap between object recognition application developers and expert parallel programmers, we have performed pattern mining on object recognition applications, and summarize a pattern list for object recognition in Section 3.3. A library with all the listed application patterns can be used to compose many different object recognition applications.

When parallelizing and optimizing application patterns on one kind of platform, exploring the design space of algorithm and parallelization strategy is usually more important than the design space of platform parameters. Because variations in the underlying platform are limited, the platform parameter space is restricted. In such a case, an exhaustive search is enough for exploring the design space of algorithm and parallelization strategy. We parallelize and optimize three application patterns on Nvidia GPUs, achieving 195× speedup on the Eigensolver pattern [26], 5.7× speedup on the Breadth-First-Search Graph Traversal pattern [110], and 28.6× speedup on the contour histogram pattern [111]. This design space exploration experience is described in Chapter 5.

However, when parallelizing and optimizing application patterns across different platforms, exploring the design space of platform parameters becomes important. Different platforms have different strengths, and favor different kinds of parallelism. To develop a portable and optimized library, it is necessary to apply autotuning technology when installing the library in order to tune library performance according to the hardware specialties. We propose the OpenCL for OpenCV (OLOV) project to achieve this goal. OpenCL [116] is a cross-platform programming model that is supported by many different parallel platforms. OpenCV [21] is an open source library with many computer vision algorithms. By extending the OpenCV library with the application patterns we discovered using OpenCL, we can provide the building blocks to compose many object recognition systems on many different

hardware platforms. We develop two cross-platform autotuners in the OLOV project; these two autotuners and their performance on different platforms are introduced in Chapter 6. We plan to implement more autotuners in the future.

In order to show how the library can be used to develop an object recognition application, we pick one state-of-the-art object recognition system, analyze it, devise a software architecture of the system, and then integrate the parallel application patterns following the software architecture. We are able to achieve 119× speedup for the training stage, and 115× speedup for the deployment stage [111]. This proves the effectiveness of the parallel library we have developed.

The main contribution of this thesis is to propose a systematic way to bridge the implementation gap, and to parallelize and optimize all kinds of different computations using patterns. This proposed approach is supported by a case study of object recognition applications. We find the most important application patterns of object recognition applications. We choose a subset of the application patterns and explore the design space to parallelize and optimize them. We achieve significant speedup in each application pattern. By integrating the application patterns together, we have developed a near-real-time object recognition system. This approach is very effective, and we can expect similar success in other application domains.

1.1 Thesis Contributions

The main contributions of this thesis are as follows:

- We propose architecting applications using application patterns. These application patterns capture recurring computations in the application domain. The application architectures provide a high-level abstraction over how application patterns can be used to develop applications. By providing a library covering all application patterns, an application can be designed by composing the application patterns following a pre-defined software architecture.

- We propose architecting computations using patterns in Our Pattern Language (OPL) [73]. A software architecture offers a coarse-to-fine-grained abstraction of a given computation. Based on these abstractions, we can define the design space to be explored in order to optimize the computations on a targeting hardware platform.

- We propose exploring the design space in three layers: the algorithm layer, the parallelization strategy layer, and the platform parameter layer. By systematically exploring the design space of these three layers, we can optimize any computation on any hardware platform.

- We perform pattern-mining on state-of-the-art object recognition systems, arriving at a pattern list of the most frequent computations in object recognition systems. By providing a highly optimized library covering the patterns in the list, we should be able to develop state-of-the-art object recognition systems, and accelerate these systems significantly on the targeted hardware platforms.

- We perform exhaustive analyses on three kernels for important patterns in this list – the eigensolver, BFS graph traversal, and contour histogram. We optimize these kernels on GPUs.
- We propose the OpenCL for OpenCV (OLOV) project, which extends the OpenCV library with a set of autotuners for computationally intensive kernels at GPU platforms.
- We propose the *Cocktail Format* to represent sparse matrices. This format is able to represent specialized submatrices of the input matrix using specialized formats.
- We develop the *clSpMV* autotuner to automatically tune the sparse matrix vector multiplication kernel on GPU platforms. This is the first autotuner in the OLOV project.
- We develop the *clPaDi* autotuner to automatically tune the pair-wise distance kernel on GPU platforms. This is the second autotuner in the OLOV project.
- We develop a highly optimized parallel object recognition system by using all functions we have developed in the parallel library.

1.2 Thesis Outline

This thesis is presented as follows:

- Chapter 2 summarizes the necessary background of this thesis. It quantifies the computational costs of object recognition applications, and explains the need for parallelizing applications. The challenges of parallel programming are then discussed. A major challenge is that the expertise for application development and for parallel programming are totally different. It is very rare for people to master both areas, so an implementation gap appears between application developers and expert parallel programmers. Previous work on bridging this gap is also introduced.
- Chapter 3 describes our proposal for bridging the implementation gap for object recognition applications: to develop a parallel application library for these applications. Key application patterns are mined from state-of-the-art object recognition applications. These key application patterns define the functions that the parallel application library should support. Expert parallel programmers can then focus on parallelizing and optimizing these key application patterns. At the same time, application developers can design their applications by defining a software architecture and deploying the application patterns in that software architecture.
- Chapter 4 discusses our proposed approach for implementing and optimizing any computation on any hardware platform. The idea is to architect the computation using patterns, then define the design space based on the software architectures, and finally explore that design space.

- Chapter 5 describes our experiences with optimizing a subset of object recognition patterns by using an exhaustive search strategy on the design space. Case studies include the Lanczos eigensolver, BFS graph traversal, and contour histogram accumulation.
- Chapter 6 introduces our OpenCL for OpenCV (OLOV) project. The goal of the project is to develop a cross-platform highly optimized library based on the object recognition application patterns. We also provide a detailed description of the two developed library kernels: the sparse matrix vector multiplication kernel and the pairwise distance computation kernel.
- Chapter 7 presents how we can employ the library to develop parallel applications, using an object recognition system as a case study. The target object recognition system is introduced, architected using patterns, and parallelized using the optimized library.
- Chapter 8 concludes this thesis and describes future work.

Chapter 2

Background

In this chapter, we discuss the background and motivation for this thesis. We first analyze the computational costs for object recognition applications, and explain why parallelism is necessary to support the computational need. Second, we show the challenges of parallel programming. Because the expertise required for application development and parallel programming is totally different, a gap exists between the two . We call this as the `implementation gap` [102, 28, 25]. Finally, we introduce prior research on bridging this implementation gap.

2.1 Object Recognition

Object recognition is a major branch in the field of computer vision. The ultimate goal of computer vision research is to make machines understand the real world via visual data. The most common visual data, which anyone can create and collect, are images and videos. If we can make machines understand the contents inside images and videos, it will be a significant advance in computer vision. In order to do this, we need to analyze how human beings understand the contents of images and videos. Humans usually identify the individual objects in the images and videos, and then reason about the overall contents through the interactions among the objects. The first step of making machines understand the contents of images and videos is therefore to make machines recognize the objects inside the images and videos.

In addition to its application in robotics, object recognition technology can improve our daily lives as well. There are two different scenarios for applying object recognition technology: the cloud and the client. For the cloud usage scenario, service providers own a database of multimedia content either loaded by users or prepared by content providers. These files are usually searched or categorized by text-based tags. If the tags are missing or misleading, it is very difficult for service providers to correctly respond to user queries. However, with the help of object recognition, the contents can be used to identify files and improve the robustness of search results.

For the client usage scenario, object recognition technology can be used in personal assistant services. With the high availability of digital cameras, people often have a large

collection of photographs. With the help of object recognition, we can search the contents of photos directly to help users find the photos they seek. Similarly, in augmented reality, when an object is recognized, we can provide additional information that helps the user to understand the object. With object recognition, we can also improve positioning applications by identifying specific landmarks and signs. With the appearance of smart phones, object recognition technology is used in many mobile applications as well. Vuforia [103] is the standard development toolkit that Qualcomm created to enable augmented reality on mobile platforms. Leafsnap [94] is an application that helps people to recognize trees by the leaves. Google Goggles [55] is a mobile application that recognizes text, landmarks, books, contact info, artwork, wine, and logos. Overall, object recognition is getting more and more attention on both the cloud and client sides.

The computation flow of most object recognition applications is summarized in Figure 2.1. This flow simulates the process through which human beings recognize objects. Humans are not born with the ability to recognize objects. Rather, people learn about objects from real examples as they grow up, and then recognize objects based on what they have learned. Similarly, a training stage allows machines learn about objects from example images, and then a deployment stage lets machines apply what they have learned on query images. In the training stage, a set of training images is used as input, which contains many example images for each object. The number of example images per object is a trade-off between recognition accuracy and total computation – more example images for each object lead to better recognition accuracy, but also require more computation. A feature extraction step is used to summarize representative features from the objects in the training image set. Based on the features collected from the objects, a model is built to differentiate different objects, and to identify each object. The output of this training stage is the model for all objects. In the deployment stage, a set of query images is given. Features are extracted from the query images, and then the objects are recognized by plugging the extracted features into the model built at the training stage.

In the following sections, we quantitatively analyze the computational costs for object recognition applications.

2.1.1 Computational Cost of Feature Extraction

Features are used to capture the distinctive characteristics of each object, and usually many image processing algorithms are involved in extracting features from a given image. In order to understand the computational costs for the feature extraction computations, we have selected three state-of-the-art feature descriptors and analyze the associated algorithms.

The Scale-Invariant-Feature-Transform (SIFT) [80] is the most famous feature in general-purpose object recognition applications. The SIFT feature is good at collecting key points in the object that are invariant under scaling and rotation. This is important because the same object may have different appearances in different images. The algorithm used in SIFT has four steps. (1) Finding extreme points such as corners in the scale space. This is the major bottleneck in the SIFT feature computation. The approximate number of floating point operations (FLOPs) per pixel for this step is 20,000 FLOPs. (2) Performing

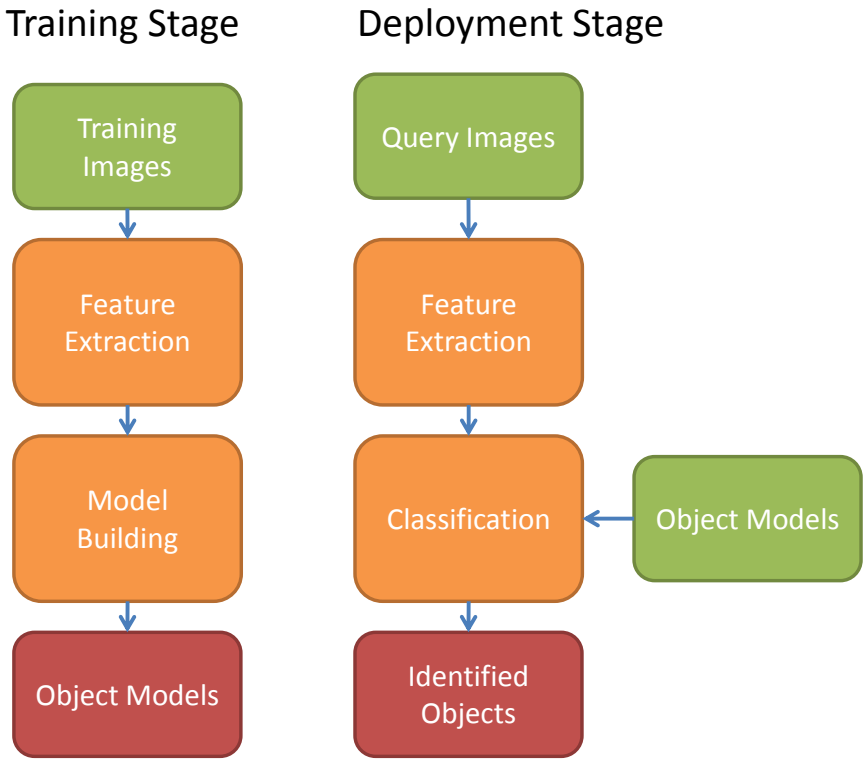

Figure 2.1: Object recognition computation flow.

key point localization to fit the extreme points to the nearby data for location, scale, and curvature. Extreme points that have low contrast or poorly locate on edges will be eliminated. (3) Assigning an orientation value to the key points. Both steps two and three are computationally-cheap compared to step one, because the number of key points is many magnitudes smaller than the number of pixels in the image. (4) Computing a distinctive descriptor for each key point. It serves as the unique signature of the corresponding key point. When a descriptor is computed, the orientation and location information of neighboring pixels around the key point are included. On average, a window of 60×60 neighboring pixels is used per key point. The large number of neighboring pixels involved makes this step expensive. Given an $M \times M$ image, assuming the number of key points extracted from the image is $M/30 \times M/30$ – a key point for every 30×30 window – then the approximate FLOPs per pixel of this step is about 1,000. As a result, the approximate number of FLOPs per pixel for the SIFT algorithm is about 21,000 FLOPs per pixel.

The Histogram of Oriented Gradients (HOG) descriptor [33] achieves the best performance on human detection applications. Given a detection window, the HOG descriptor summarizes the spatial locality and gradient orientations of all pixels within the detection window. In order to detect humans with different scales, usually the HOG descriptor is applied on a pyramid of images with different scales. The image sizes decrease from lower

layer to higher layer in the image pyramid. The scaling factor in the image pyramid is a trade-off between detection accuracy and the total amount of computation. A smaller scaling factor creates more images in the pyramid and can detect humans on more scales, but also increases the total number of computations. To estimate the total computation for the HOG descriptor, we assume the scaling factor to be 1.1 – that the width and height of an image in the pyramid are divided by 1.1× from the lower layer image. This scaling factor is used in the state-of-the-art human detection application, the poselet [20]. The HOG algorithm has three steps. (1) Normalizing the color channels of the image, and compute the maximum gradient value among the color channels on each pixel. The cost of this step is about 700 FLOPs per pixel. (2) Computing the weight of the gradient on every pixel, and accumulate the value to the proper histogram bin. The cost of this step is about 4,400 FLOPs per pixel. (3) Normalizing the histogram bins. This is a cheap step compared to the first two steps. Therefore, the approximate number of FLOPs per pixel for the HOG algorithm is 5,100.

Contour features describe the shapes of objects. The region based object detection algorithm proposed by Gu et al. [61] employs contour features to detect objects with special shapes. Among all existing contour detection algorithms, the global probability (gPb) algorithm [81] achieves the best results. The gPb algorithm has three steps. (1) Computing local contours using brightness, color, and texture cues [88]. Because there are four channels (one brightness channel, two color channels, and one texture channel), and each channel has three cues of different scales, $4 \times 3 = 12$ local cues are collected in this step. This is an expensive step, and costs about 77,600 FLOPs per pixel. (2) Computing the global contour using the normalized cut algorithm [108]. Spectral graph theory can be used to transform the normalized cut algorithm into an eigen-decomposition problem. We then need to find the eigenvectors of an affinity matrix that describes the similarity between each pair of pixels in the image. The Lanczos algorithm [12] is a good fit for this. Assuming that we need only 500 iterations to converge to the necessary eigenvalues, the cost of this step is 81,000 FLOPs per pixel. (3) Performing a linear combination on the local and global contours. This is a cheap step. As a result, the total cost of the gPb algorithm is about 158,600 FLOPs per pixel.

The computational costs for state-of-the-art feature extraction algorithms are summarized in Table 2.1. However, this is only a simplified approximation – we skip the index algebra and other cheap steps. As such, the actual cost is higher than these numbers. Even the smallest cost in Table 2.1 is still on the order of several thousand FLOPs per pixel. Therefore, when considering images with more than one megapixel, the total computational cost will be tens to hundreds of gigaFLOPs or more.

Table 2.1: Computational costs for state-of-the-art feature extraction algorithms.

Image Feature	SIFT [80]	HOG [33]	gPb [81]
Approximate FLOPs per pixel	21,000	5,100	158,600

2.1.2 Computational Cost of Classification

After the image features are extracted, they are represented by feature vectors, which are collections of numbers. However, these numbers do not directly tell us any information about objects. What we need to do is build a model to evaluate whether a feature vector corresponds to an object. In order to do so, we need a training stage to build the model and then apply the model on the query feature vectors. Machine learning algorithms are usually used in classification. To understand the computational costs of classification algorithms, we analyze two state-of-the-art classification algorithms using FLOPs per vector element as the measuring unit. A vector element is a number in the feature vector.

The Support Vector Machine (SVM) algorithm [31] is one of the machine learning algorithms that achieves the highest accuracy on the classification problem. Given a set of points that are a mixture of instances from two different categories, the training stage of the SVM algorithm finds a maximum-margin hyperplane that separates the points into the different categories. The hyperplane cuts the sample space into two halves, each corresponding to one category. The deployment stage of the SVM algorithm evaluates the category of the query vector by checking which half it locates. For simplicity, we estimate the computational cost of linear SVM. For a more complicated SVM kernel, the required computation is larger. At the training stage we need to solve a quadratic programming problem. One famous way of solving the problem is to apply the Sequential Minimal Optimization (SMO) algorithm [101]. This algorithm iteratively selects two vectors and adjusts their weights until convergence. Many different heuristics have been proposed to select the two vectors in each iteration. One famous heuristic is the second order heuristic [45]. This heuristic first finds the vector that largely violates the Karush-Kuhn-Tucker (KKT) constraints, then finds the paired second vector that most improves the objective function. The computational cost of the training stage is proportional to the number of iterations in the SMO algorithm. Because the SMO algorithm updates the weights of two vectors in an iteration, the total number of iteration is usually in the order of the number of the training points. Most of the machine learning benchmarks have more than a thousand training points, so the estimated FLOPs per vector element is 15,000 FLOPs in the training stage. The deployment stage is conducted by performing kernel functions between the query vector and the support vectors, which are a subset of the training vectors. Assuming that the number of support vectors is 500, the estimated FLOPs per vector element is 1,000.

K-Nearest Neighbor (KNN) [8] is also a famous approach for classification. There is no training needed in the KNN algorithm – all computation happens at the deployment stage. Given a query vector, the KNN algorithm finds the k closest neighbors from the training examples, and lets the k nearest neighbors vote on the query vector's category. For a training set of 1,000 vectors, the computational cost per vector element at the deployment stage is 3,000 FLOPs based on the L2 distance. However, a naive KNN algorithm is error prone. People usually apply other techniques to improve the results. The region-based object recognition application developed by Gu et al. [61] is an example of this: both the training and deployment stages use the nearest neighbor concept. In the training stage, the distances between similar and different object parts are computed. The weights of all object parts are then computed by the importance of the object parts. An "important part"

means that it reappears in similar object examples (small distance), and is distinguishable from other different objects (large distance). The computational cost of the training stage is 3,000 FLOPs per vector element. In the deployment stage, only object parts with non-zero weights are considered. Assuming that we have only 200 important parts from the 1,000 total parts, the computational cost of the deployment stage is then 600 FLOPs.

The computational costs for state-of-the-art classification algorithms are summarized in Table 2.2. Again, this is a simplified approximation. We do not include the index algebra and other cheap steps, so the actual cost is higher.

Table 2.2: Computational costs for state-of-the-art classification algorithms.

Classification Algorithm	SVM [31]	KNN [8]	Gu et al. [61]
Training: Approximate FLOPs per vector element	15,000	0	3,000
Deployment: Approximate FLOPs per vector element	1,000	3,000	600

2.1.3 Computational Cost of Real Object Recognition Systems

At this point, we have estimated the computational cost of feature extraction and classification algorithms. By plugging in these numbers, we can analyze the total computational cost of real object recognition systems. We use the region-based object recognition developed by Gu et al. [61] as our case study. It uses the gPb contour feature and the nearest-neighbor-based classification algorithm.

In the training stage, we analyze the computational cost of the object recognition system on three different benchmarks: the ETHZ shape benchmark [50], the PASCAL VOC benchmark [44], and the ImageNet benchmark [39]. The ETHZ shape benchmark has five object categories. The number of training images is 127. The average size of each image is about 0.2 megapixels. The feature extraction cost is $127(\text{images}) \times 200,000(\text{pixels}) \times 158,600(\text{FLOPs per pixel}) = 4$ teraFLOPs. In the model-building step, each region in each image corresponds to a training sample. The average number of regions per image is approximately 200. The dimension of the feature vector is 128. Because the model building cost is proportional to the number of samples in the training set, the estimated FLOPs per vector element is now $127(\text{images}) \times 200(\text{regions}) \times 3(\text{FLOPs}) = 76,200$ FLOPs. The model building cost is $127(\text{images}) \times 200(\text{regions}) \times 128(\text{vector dimension}) \times 76,200(\text{FLOPs}) = 248$ gigaFLOPs. The total cost is therefore 4.25 teraFLOPs. The PASCAL VOC benchmark has 20 object categories, and the number of training images is 5,011. The average size of the images is 0.18 megapixels. The feature extraction cost is $5,011 \times 180,000 \times 158,600 = 143$ teraFLOPs. The model-building cost is $5,011 \times 200 \times 128 \times 5,011 \times 200 \times 3 = 386$ teraFLOPs. The total cost is therefore 529 teraFLOPs. The ImageNet database has more than 10 million images on more than 10 thousand object categories. It hosts the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) to encourage progress in the large-scale object recognition problem. ILSVRC uses a subset of the ImageNet images. It has 1,000 object categories, and the number of training images is 1.2 million. The average size of the images is 0.18 megapixels. The feature extraction cost is $1,200,000 \times 180,000 \times 158,600 = 34$ petaFLOPs.

The model-building cost is $1,200,000 \times 200 \times 128 \times 1,200,000 \times 200 \times 3 = 22$ exaFLOPs. The total cost is therefore 22 exaFLOPs.

In the deployment stage, we can discuss the computational costs under three different scenarios:

- **Object Recognition on Benchmarks**: This scenario reflects the computational cost required for object recognition research. The ETHZ shape benchmark [50] has 128 test images, the average size of which is 0.2 megapixels. The feature extraction cost is $128 \times 20,0000 \times 158,600 = 4$ teraFLOPs. For the classification step, we need only to find whether the important object regions show up in the query images. The average number of important object regions per image is 12. Because we need to compute the distance between the important object parts in the training images with the query regions, the estimated FLOPs per vector element is 127(training images)×12(regions)×3(FLOPs) = 4,572 FLOPs. The computational cost of the classification step is 128(testing images)×200(regions per testing image)×128 (vector dimension)×4,572(FLOPs) = 15 gigaFLOPs. The total cost is therefore 4.02 teraFLOPs. The PASCAL VOC benchmark [44] has 4,952 testing images. The feature extraction cost is $4,952 \times 180,000 \times 158,600 = 141$ teraFLOPs. The classification cost is $4,952 \times 200 \times 128 \times 5,011 \times 12 \times 3 = 23$ teraFLOPs. The total computational cost is therefore 164 teraFLOPs. The ILSVRC benchmark has 100,000 testing images. The feature extraction cost is $100,000 \times 180,000 \times 158,600 = 2.85$ petaFLOPs. The classification cost is $100,000 \times 200 \times 128 \times 1,200,000 \times 12 \times 3 = 111$ petaFLOPs. The total computational cost is therefore 114 petaFLOPs.

- **Object Recognition in One Image**: This scenario reflects the computational cost when an user wants to identify objects from a photo. In real applications, the capability of finding objects from 1,000 categories is far more useful than from 5 or 20 categories. Therefore, we assume that the classification model is based on the ImageNet benchmark, which has 1,000 categories. Regarding the dimensions of the photo, we use an iPhone 4S camera setting, which is 8 megapixels. Based on such a resolution, the feature extraction cost is $8,000,000 \times 158,600 = 1.3$ teraFLOPs, and the classification cost is $200 \times 128 \times 120,000 \times 12 \times 3 = 111$ gigaFLOPs. The total computational cost is therefore 1.4 teraFLOPs.

- **Object Recognition on YouTube Videos**: This scenario reflects the computational cost when a service provider wants to analyze videos that users upload. According to the official YouTube statistics [109], 60 hours of video are uploaded every minute. Assuming that the resolution is 480p, or $850 \times 480 = 408,000$ pixels, that the frame rate is 30 fps, and that we apply the object recognition system on every frame, the feature extraction cost is $60 \times 60 \times 60 \times 30 \times 408,000 \times 158,600 = 420$ petaFLOPs. The classification cost is $60 \times 60 \times 60 \times 30 \times 200 \times 128 \times 120,000 \times 12 \times 3 = 717$ petaFLOPs. The total computational cost is therefore 1137 petaFLOPs.

The computational costs of applying the region based object recognition algorithm developed by Gu et al. [61] on different scenarios are summarized in Table 2.3

Table 2.3: Computational cost for real object recognition systems.

		Feature Extraction FLOPs	Classification FLOPs	Total FLOPs
Training	ETHZ [50]	4 tera	248 giga	4.25 tera
	PASCAL VOC [44]	143 tera	386 tera	529 tera
	ILSVRC [39]	34 peta	22 exa	22 exa
Deployment	ETHZ [50]	4 tera	15 giga	4.02 tera
	PASCAL VOC [44]	141 tera	23 tera	164 tera
	ILSVRC [39]	2.85 peta	111 peta	114 peta
	One Image	1.3 tera	111 giga	1.4 tera
	YouTube Video	420 peta	717 peta	1137 peta

2.1.4 Solution to the Bursting Requirements of Computation: Parallel Programming

According to Table 2.3, the computational costs of applying this state-of-the art object recognition system to real-world problems are enormous. Even classifying a single image requires 1.4 teraFLOPs. On a 3.0 GHz single-core desktop CPU, this can take 8 minutes to process. On a less powerful mobile platform, it can take up to 160 minutes. This delay makes it impossible to turn object recognition technology into a practical mobile application. On the cloud side, the problem is no easier. To analyze YouTube videos that users upload in one minute, we would need to compute 1137 petaFLOPs. On a 3.0 GHz single-core desktop CPU, this would take 12 years to process.

Further, to make the object recognition application realistic, we would need far more than 1,000 object categories. And, to ensure good quality of a classification model, we would need many training images for each object category. With the advancement of camera and storage technologies, the resolution of images and videos will keep growing. With the high availability of smart phones, people will upload more images and videos to the cloud. Therefore, the amount of data we need to analyze is exploding, and so is the number of required computations. Because of the power wall [9], CPU frequency does not scale over 4 GHz. Therefore, it is impossible to solve the computational challenge of object recognition algorithms on single-core processors. We need to parallelize the algorithms and take advantage of these parallel platforms.

2.2 Challenges in Parallel Programming

Even though parallelizing object recognition algorithms and executing them on parallel platforms can solve the computational challenge, it is not a trivial solution. Parallel programming is difficult, and it takes considerable effort and expertise to parallelize an existing algorithm on a parallel platform. We summarize the challenges of parallel programming in the following seven sections.

2.2.1 Variation of Hardware Platforms

The computing industry as a whole is moving towards parallel computing. One of the driving forces comes from hardware designers, who are aware of the power wall and are on the frontier of exploring parallel hardware platforms. However, parallel architecture does not have a standard, and researchers keep proposing new computer architectures to explore parallelism on hardware platforms. As a result, there are a wide variety of different parallel hardware platforms available in the market. For example, `core level parallelism` integrates many stand-alone processors on a single die. `Thread level parallelism` launches many threads on a single core. `Superscalar` processors [71] execute multiple instructions in parallel. `Out-of-order execution` [65] rearranges the ordering of instructions to execute independent instructions in parallel. `Very Long Instruction Word (VLIW)` [51] bundles several operations into a single instruction, and performs all the operations in parallel. `Single Instruction Multiple Data (SIMD)` [65] fetches a single instruction and then applies that instruction on multiple data operands in parallel.

In addition to the different architectural designs in the execution units, the memory hierarchies of different hardware platforms also vary. There are `Uniform Memory Access (UMA)` [65] machines, for which all the processors share physical memory uniformly. There are also `Non-Uniform Memory Access (NUMA)` [65] machines, for which physical memory is partitioned into several pieces, and the cost of accessing data in a piece is related to the distance between the processor and the memory chunk. The CPU family usually has a memory hierarchy of DRAM, Translation Lookaside Buffer (TLB) cache, L3 cache, L2 cache, L1 cache, and registers. The GPU family usually has a memory hierarchy of DRAM, L2 cache, L1 cache, software manageable scratchpad, texture cache, and constant memory. Heterogeneous CPU-GPU hybrid designs usually let multiple CPUs and one GPU share the L3 cache, and all the CPUs and GPU have their own internal memory hierarchy as well.

These variations in the capabilities of hardware platforms make parallel programming difficult. If we do not want to spend time evaluating the performance on all available platforms, then we must choose a single platform, and optimize our software for it. Without expertise in computer architecture, it is very difficult to determine which platform to choose, and how to optimize it. If portability is a major concern, then addressing the huge space of hardware platforms is a significant challenge. All the varieties in the hardware platforms make parallel programming difficult.

2.2.2 Variation of Programming Models

Instead of coding in assembly language, it is more common to use a programming language that has a higher abstraction level. Similarly, when programming on parallel hardware platforms, instead of specifically coding parallel instructions, programmers usually employ models that generate parallel instructions automatically. Again, because there is no standard parallel programming model, programming language researchers have created many parallel languages for different programming models. `OpenMP` [99] is a parallel Application Programming Interface (API) that relies on the fork-join programming model. It creates parallel sections in the code. When entering the parallel sections, many threads are forked;

when exiting the parallel sections, the created threads are joined. `Threading Building Blocks (TBB)` [114] is a parallel API that relies on the workpile programming model. A thread pool is maintained, and a scheduler assigns tasks to idle threads. `POSIX Threads (pthreads)` [24] is a parallel API that relies on the Single Program Multiple Data (SPMD) model. Many threads are created, and the same program is executed by all threads, but different threads can access different data sets according to their thread indices. `Compute Unified Device Architecture (CUDA)` [96] is a parallel API that relies on the kernel parallelism programming model. Many thread blocks are created, and scheduled to be executed on processors with wide SIMD units. `OpenCL` [116] is a parallel API that mixes kernel parallelism and task queues. In this programming model, a task queue is created, and many kernels are enqueued. Within a kernel, many work groups are created and scheduled to available SIMD units.

Different programming models have different assumptions and different overhead. CUDA, for example, assumes that all thread blocks are independent; only threads within a thread block can communicate with one another. The kernel launch introduces overhead, and branching in the execution path within a thread block also introduces overhead. In order to achieve the correct results, we need to be aware of the assumptions. In order to optimize the parallel code, we need to reduce the overhead, which is best understood by knowing how the programming model is mapped to the underlying hardware platform. This requires a background in computer architecture.

Given an algorithm that we want to parallelize and a specific parallel hardware platform that we want to use, we need to understand the trade-offs among all parallel programming models that are applicable on the parallel hardware platform. Then we must choose the best programming model for the purpose. The knowledge and experience required to understand all of these parallel programming models makes parallel programming difficult.

2.2.3 Finding Parallelism in Algorithms

Not all algorithms have parallelism to explore – some are naturally serial. For example, algorithms that enforce a strict ordering on the data to be processed are serial. Huffman encoding and Dijkstra's shortest path [30] are in this category: they rely on a priority queue, and always extract one element with the highest priority from the queue. If multiple elements are extracted from the priority queue and processed, the results might be suboptimal. Algorithms that enforce specific dependency on every step are also serial – iterative methods such as conjugate gradient and gradient descent are in this category. Because every iteration is dependent on the previous one, there is no way to parallelize across the iterations.

Some algorithms can be parallelized, but do not always have enough parallelism to be explored. For example, the Breadth-First-Search (BFS) graph traversal algorithm [30] can be parallelized because all the frontier nodes are independent and can be processed in parallel; however, the number of frontier nodes per level is graph dependent. If the graph is a single path, all the BFS levels have only one frontier, so the algorithm ends up with serial execution.

Different algorithms have different assumptions and different properties. In order to find

parallelism in an algorithm, we must perform a dependency check on the procedure and identify independent steps. If the number of independent steps is not sufficient to saturate all processing units, we need clever algorithmic improvement to remove the dependency. Otherwise, performance will be bounded by the number of independent steps. As a result, identifying sufficient parallelism in a given algorithm makes parallel programming difficult.

2.2.4 Memory Optimizations

Hardware vendors continue to increase the number of processing units on a single die, so the peak theoretical number of FLOPs scales nicely over time. However, memory bandwidth cannot keep up with the pace of computing power. We collect the peak performance and memory bandwidth numbers for Intel and Nvidia mainstream products from year 2004 to 2010, and summarize these in Figure 2.2 and Figure 2.3. The left y-axis corresponds to the peak performance in gigaFLOPs, and the right y-axis to the memory bandwidth in gigabytes per second. Assuming we read two single precision floating point numbers (eight bytes in total) and perform eight floating point operations on these two numbers. Then in average we perform one floating point operation per byte we read from memory. Under this condition, the measuring unit of both right and left y-axises are the same. According to the plots, the peak performance of both the Intel and Nvidia products is significantly greater than the available memory bandwidth, and the gap is growing with time. It is very fast to issue instructions when data are in registers, but it is very slow to read data from memory to registers. Very likely, there will be many computing units available, but the data will not have arrived yet. As a result, optimizing memory operations is essential for parallel programs.

Again, a background in computer architecture is necessary to understand the memory hierarchy of the underlying hardware platform. Memory optimizations are highly related to the underlying memory system. Some common memory optimization strategies include the following:

- `Compaction` [42]: The granularity of a single memory transaction is the size of a cache line. Usually the cache line is 64 to 128 bytes. If the operands are located in multiple cache lines, many memory transactions are needed to collect all operands and then to perform the operations. Compaction optimization groups operands into the same cache line, reducing the number of necessary memory transactions.

- `Alignment` [42]: Even if the operands are grouped consecutively, they might be stored across the cache line boundaries. Extra memory transactions are then needed to load all the operands. Alignment optimization aligns the operands with cache line boundaries, reducing the total number of memory transactions.

- `Blocking` [42]: This strategy takes advantage of the temporal locality of the data. The data are grouped into blocks that fit into caches, and blocked data are reused as many times as possible. If a hierarchy of caches is available, we can have a hierarchy of blocks that fit into different levels of the caches.

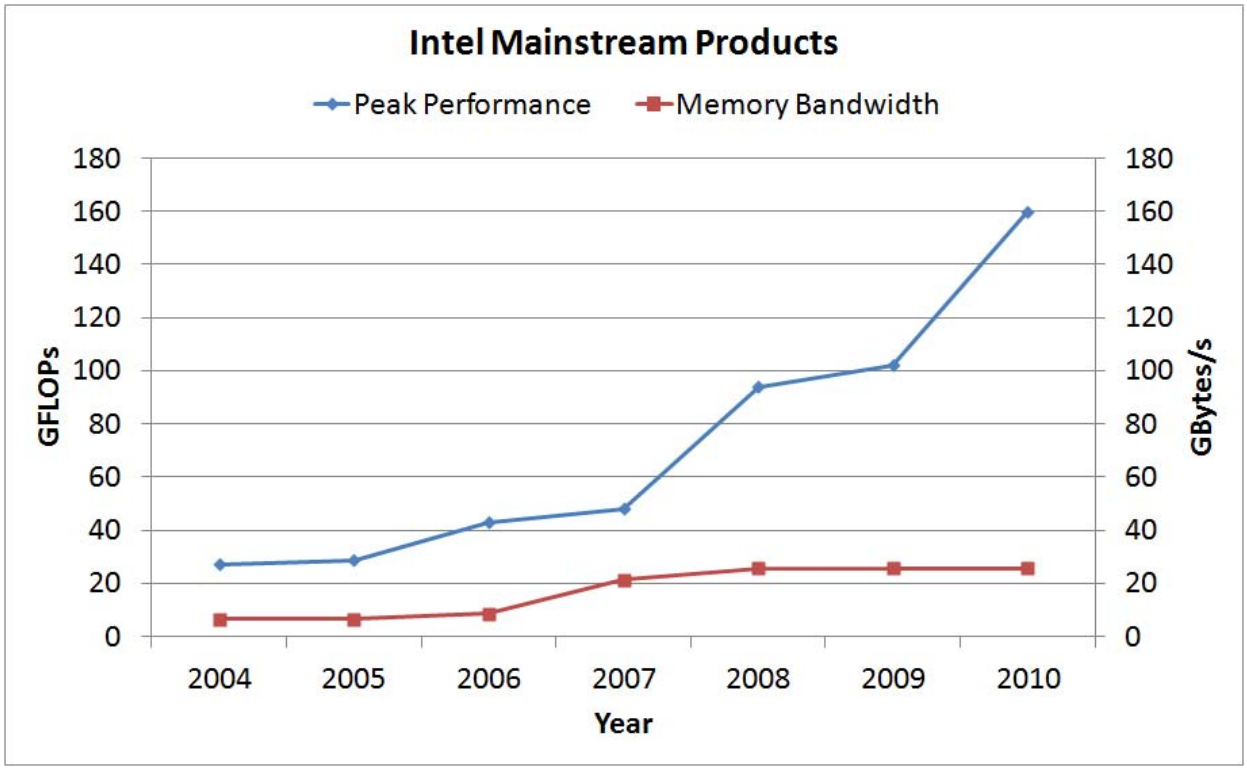

Figure 2.2: The peak performance (in GFLOPs) and the memory bandwidth (in GBytes/s) for Intel mainstream products over the years.

- `Prefetching` [42]: This strategy takes advantage of the spatial locality of the data. After a cache line is loaded, we can predict the next cache line and start fetching it before the program queries. If the program queries that exact cache line later on, memory latency is reduced.

Although these memory optimizations can be performed in serial programs, the parallel paradigm is considerably more complicated. For example, on a NUMA machine, the physical memory is partitioned into several pieces, and the memory latencies of accessing different pieces are different. Given a computation, we need to block the operations in a way such that we spend more time on accessing data from low-latency piece and spend less time on accessing data from high-latency piece. As such, the need of memory optimization makes parallel programming difficult.

2.2.5 Scalability and Amdahl's Law

Parallel programs have overhead. For example, creating/scheduling/terminating threads and processes incurs overhead. Communication/synchronization across multiple threads/processes/cores also introduces overhead. The more parallelism we explore, the more overhead we might encounter. This overhead can put an upper-bound on the maximum amount

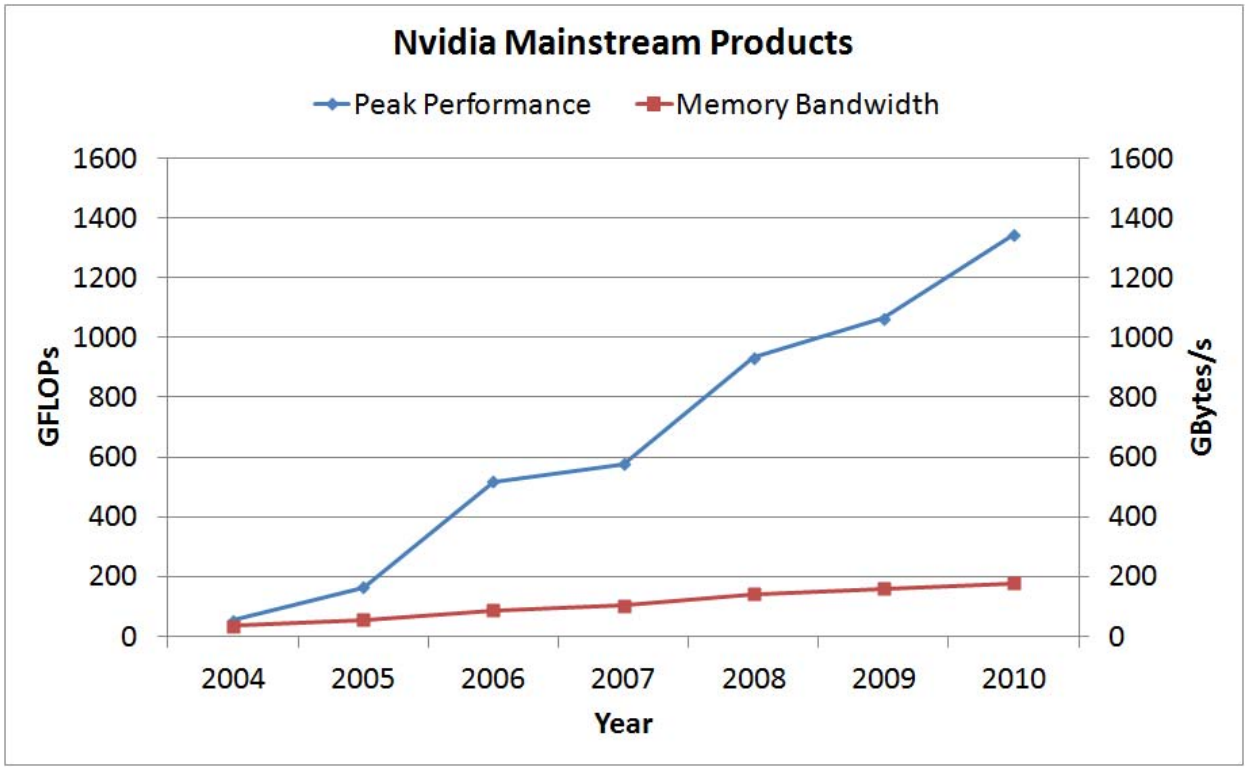

Figure 2.3: The peak performance (in GFLOPs) and the memory bandwidth (in GBytes/s) for Nvidia mainstream products over the years.

of parallelism we can explore. Because of the overhead, the more parallelism we express, possibly the less performance we can achieve with it. It is also possible that such performance would even be worse than a serial implementation. Moreover, according to Amdahl's Law [5], the maximum benefit we can achieve by parallelizing an algorithm is bounded by the serial fraction. Assuming that the fraction of the serial part of the algorithm is s, the parallel part is $1 - s$. Let t be the execution time of the serial algorithm, and let t' be the execution time after we have parallelized the parallel portion using p processing units. The maximum speedup we can get from parallelism is then computed as follows:

$$\frac{t}{t'} = \frac{1}{s + \frac{1-s}{p}}. \tag{2.1}$$

Even if we have an infinite number of processing units available, the ultimate speedup is bounded:

$$\lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s}. \tag{2.2}$$

As a result, the benefit from parallelizing the algorithm is bounded by the parallel overhead and the serial fraction of the algorithm. In order to scale our parallel algorithm to

as many processing units as we want, we must reduce the overhead and the serial portion of the algorithm. In order to do this, we need a deep understanding of the hardware architecture, the programming model, and the algorithm. This scalability concern makes parallel programming difficult.

2.2.6 Load Balancing

Some algorithms have a massive number of independent tasks, but the amount of work in each task is irregular. In this situation, some processing units perform more operations, while others are idle. This load balance problem deteriorates the performance of the parallel algorithm. Sparse matrix vector multiplication is an example of this. One way to parallelize the computation is to assign different matrix rows to different processing units. The amount of work per row is proportional to the number of non-zeros per row. If the first row of the matrix has n non-zeros while all other rows have 1 non-zero, then the execution time for the first processing unit is proportional to n, while the execution times for other processing units are constant. The overall execution time is dominated by the first processor, while all other processors are idle most of the time.

In order to solve the load balance problem, we must find the proper granularity of the tasks, and schedule them evenly across all available processing units. More aggressively, we can dynamically adjust the work loads of all processing units. These complicated concerns regarding the load balance problem make parallel programming difficult.

2.2.7 Concurrency Bugs

In parallel programming, many resources are shared among different processing units. This has the potential to introduce concurrency bugs that do not happen in serial programs. Some well-known concurrency bugs include in the following:

- `Race Condition` [100]: A race condition happens when a thread is writing data to a memory address, and another thread is reading or writing to the same memory address. Under this situation, the execution ordering of the two threads affects the results. If the reading happens before the writing operation, it gets the old data instead of the new.

- `Deadlock` [100]: Deadlock happens while a thread stalls forever. For example, if two threads lock two different resources, but each waits for the other to release the resource, then both will stall forever.

- `Livelock` [100]: Livelock happens while a thread performs operations but the overall algorithm never progresses. For example, if two threads both need the same two resources to make progress, and each owns one resource, then they may continue releasing and exchanging that resource for the other. Although the threads are performing operations, neither has both resources at any time, so the overall algorithm does not progress.

- `Starvation` [100]: Starvation happens when a task is never processed. This can happen if the task scheduler uses a priority queue to schedule tasks. Before a low priority task is scheduled, there are always higher priority tasks inserted into the queue, so that the low priority task is never scheduled and processed.

In order to achieve correct results and ensure that the program terminates, we must avoid these concurrency bugs. These bugs make parallel programming difficult.

2.3 The Implementation Gap

We can classify programmers into two categories according to their expertise: application developers and expert parallel programmers. As shown in Figure 2.4, the application developers know the application very well, but do not know much about parallel programming. On the other hand, expert parallel programmers are well trained for parallel implementations, but are not familiar with the computations that the application needs. The special expertise needed for either end creates the implementation gap.

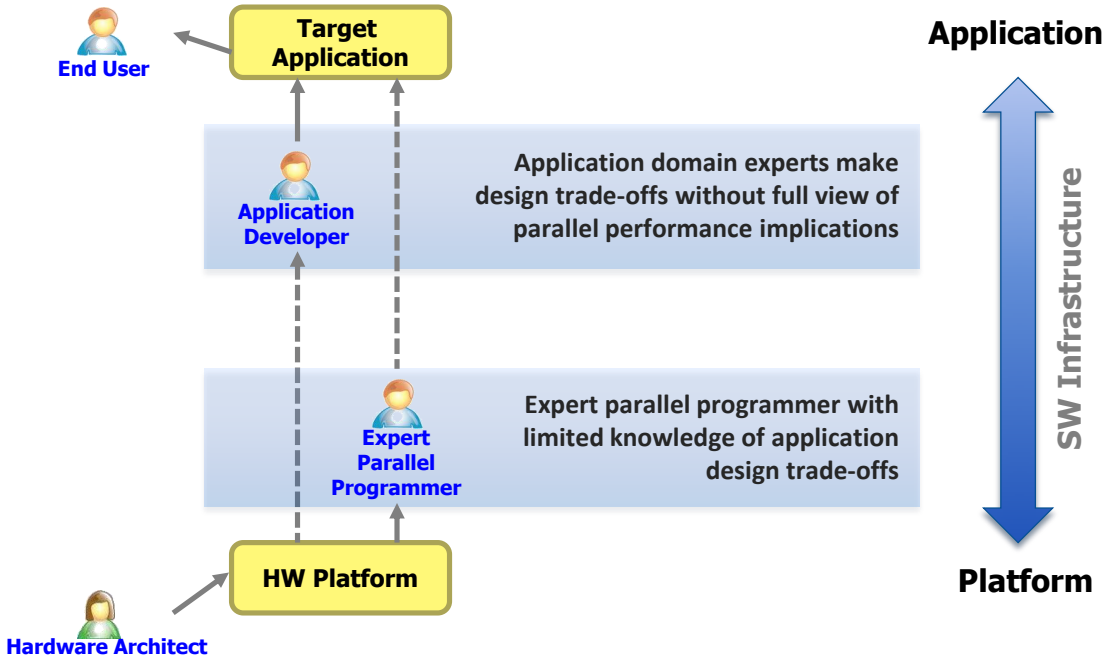

Figure 2.4: The implementation gap between application developers and expert parallel programmers.

From the perspective of application developers, the implementation gap appears because developers want their applications to run quickly and efficiently. If there is no such need, developers can rely on their naive serial implementation, and the implementation gap does

not exist. However, this is not true for all applications. As discussed in Section 2.1, object recognition applications require an enormous number of computations, and there is no hope of performing all of them on a single processor. An alternative method to reduce the implementation gap would be to make application developers learn parallel programming. However, as discussed in Section 2.2, parallel programming is very difficult, and it takes years to master. If portability is a major concern it might take even longer, because different hardware platforms require different optimization approaches.

Conversely, expert parallel programmers have years of experience with parallelizing algorithms on different parallel hardware platforms, but do not have the application domain knowledge. Regarding object recognition applications, for example, they do not know how to define and extract features from images, how to build effective and highly accurate models, and how to apply the models to classify query images. When many features and models are available, expert parallel programmers do not understand which one is better under what circumstances. It would take years for the expert parallel programmers to master the application domain.

From both the application developer end and the expert parallel programmer end, the implementation gap is difficult to close. Therefore, we need intermediate layers between the two to bridge the gap.

2.4 Prior Work

Researchers and programmers are aware of the implementation gap, and have tried different ways to bridge it. Chong proposes using pattern-oriented application frameworks to help speech recognition application developers to efficiently develop parallel speech recognition systems [28]. The proposed application frameworks are very productive, and application developers need to make only some customization decisions – all other implementation details are covered by the frameworks. However, Chong's frameworks are not very flexible, supporting only a set of pre-defined plug-ins. Catanzaro proposes using compiler transformation algorithms to parallelize python source code automatically [25]. This idea can be generalized to many data-parallelism based computations, and can be applied to many different applications. However, it is too general, and does not take advantage of domain-specific knowledge. DeVito el al. propose using domain-specific languages to build portable parallel mesh-based partial differential equation solvers [41]. This approach is general enough for application developers, and also takes advantage of domain-specific knowledge. However, application developers need to learn the new language. Like these, different methods for bridging the implementation gap have different advantages and disadvantages – typically trade-offs among efficiency, productivity, portability, and flexibility.

There is also research that tries to bridge the implementation gap for computer vision applications. OpenCV [21] is an open-source computer vision library with a large collection of functions related to computer vision applications. From the perspective of object recognition applications, it provides many image processing and machine learning routines. The initial implementation is serial, focusing on improving the productivity of computer vision application developers. Because developers of the OpenCV project are aware of the com-

putational challenges of computer vision applications, they are gradually providing some parallel routines. However, since productivity and portability are their major concerns, the efficiency of these parallel routines is not very satisfying. Fung et al. propose the OpenVIDIA project [53] to solve the problem. The OpenVIDIA is a collection of computer vision related computations optimized on GPUs. It is an ongoing project, so only a limited number of functions are provided, including image processing, optical flow, and feature tracking. It relies heavily on the CUDA programming model, and cannot be ported to GPUs from other vendors. The GpuCV project proposed by Allusse et al. [1] is similar to the OpenVIDIA project. However, instead of using the CUDA programming models, it employs OpenGL and OpenGL Shading Language [105] to provide a set of image processing routines. The API design follows the OpenCV library. If an application relies on a function in OpenCV and that function is also supported by GpuCV, then the application developer can easily switch to the accelerated version.

In summary, the OpenCV library is the most famous intermediate layer that bridges the implementation gap, and covers a very large portion of the computations in computer vision applications. However, productivity and portability are a higher priority, so performance is sacrificed. Other projects focus instead on efficiency, and provide optimized implementation on GPU platforms. However, these are ongoing projects – they cover only a very restricted portion of the overall computations, and can be executed on only a restricted number of platforms.

2.5 Summary

Object recognition applications usually are composed of two stages: the training stage of building a model based on training examples, and the deployment stage of applying the model to query images. The computations required for both stages include image processing and machine learning. The image processing computations are used to summarize the important features from the images, while the machine learning computations are used to develop a model to identify objects from the feature vectors. We have estimated the number of FLOPs required for both the feature extraction and machine learning computations. State-of-the-art computations are very expensive, and the total number of computations on an object recognition system is proportional to the size of the images, the size of the training image set, and the size of the query image set. A realistic object recognition system needs to identify thousands of different object categories, and in order to accurately identify an object, many examples are needed. By putting all these requirements together, the total computational cost of the training stage is enormous. Further, with the advancement of camera technologies, the size of images are increasing. With the spread of smart phones, users upload a huge number of images and videos to service providers. These factors cause an exploding computational cost at the deployment stage. Because frequency scaling stalls, the performance of single processor cannot scale up any more. The only hope for resolving this computational challenge is to parallelize object recognition applications.

Parallel programming is difficult. Many different parallel platforms are available, and it requires lengthy training to understand all of them. Similarly, many different parallel

programming models are available, it is difficult to master all of them. Sometimes it is a challenge just to find sufficient parallelism in the existing algorithms. Because memory bandwidth scales much more slowly than computation capability, memory optimization is necessary for parallel programs. Parallel programs have overhead, so it is also difficult to make performance scale linearly to the number of available processing units. Even if an algorithm has massive parallelism, we must also find a good scheduling algorithm to balance the work load among all available processing units. Finally, parallel programming is subject to new classes of bugs that do not happen in serial programs.

Application developers have expertise in application domain knowledge, but usually do not know how to write parallel programs. Conversely, expert parallel programmers have expertise in optimizing computations on many parallel hardware platforms, but do not know how to make design decisions for applications. It is very rare for any programmer to master both the application domain and parallel programming. As a result, an enormous gap exists between application developers and expert parallel programmers. This is the implementation gap.

Researchers are aware of the implementation gap, and many different methods have been proposed to bridge the gap. These different methods have their own advantages and disadvantages – usually trade-offs among efficiency, productivity, flexibility, and portability. In the following chapters, we introduce our own approach to bridging the implementation gap, and offer case studies to evaluate the effectiveness of the proposed approach.

Chapter 3

Parallel Application Library for Object Recognition

This chapter introduces our proposal for bridging the implementation gap between object recognition application developers and expert parallel programmers described in Figure 2.4. We propose using a parallel application library as an intermediate layer. In order to deploy the library in a systematic way, we propose architecting the application using application patterns. Finally, we perform pattern mining in state-of-the-art object recognition systems. These application patterns define the functions that the parallel application library should support.

3.1 Parallel Application Library

We propose using a parallel application library to bridge the implementation gap between object recognition developers and expert parallel programmers. A library is a collection of functions, each of which has an implementation and an interface. The interfaces of the functions are simple and exposed to library users, who need to know only the capabilities of the functions. When users need a specific capability, they can call the function using the pre-defined interface. This is simple and straightforward. On the other hand, the implementations of the functions are complicated, and hidden from library users. Library users do not need to know how the capability of the function is achieved; only the library developers need to know the details of the function implementations. As a result, the parallel application library nicely separates the expertise required for application developers and expert parallel programmers. Application developers can focus on designing their application using function interfaces without worrying about how to parallelize the functions. Similarly, expert parallel programmers can focus on optimizing the functions supported by the library without worrying about how the functions are used in real applications.

Every solution to the implementation gap has its advantages and disadvantages – usually trade-offs among efficiency, productivity, portability, and flexibility. As such, we discuss our proposed parallel application library from these four perspectives:

- **Efficiency**: Efficiency measures how well the library is parallelized and optimized.

Execution time is the most common way to quantify efficiency. Because the computational cost of object recognition applications is enormous, efficiency is our top priority. We propose a systematic method of parallelizing and optimizing computations in Chapter 4. In Chapters 5 and 6, following our proposal, we extensively explore the design space to determine the best implementations of certain library functions.

- **Productivity**: Productivity measures the programming effort needed to develop the application. It includes background for the application, knowledge for the programming environment, and time required for finishing the implementation. It is hard to quantify the required background and knowledge, so researchers only use implementation time to reflect the programming effort of developing an application [72]. A rough estimate of implementation time is lines of code. The fewer the lines of code, the better the productivity. Productivity is enabled by providing abstractions over the application, and these abstractions can be hierarchical. Higher level abstractions encapsulate more concepts, and provide high productivity. Lower level abstractions expose more details, but provide lower productivity. We propose using application-level software architectures to realize this hierarchical design. That is, the productivity of the parallel application library is defined by the software architectures. These application-level software architectures are discussed in Section 3.2.

- **Portability**: Portability measures how many computing platforms the library can be installed and executed on. As described in Section 2.2.1, there are many parallel hardware platforms available today, and these have different computer architectures. There is no one-size-fits-all solution. If we want portability, either we must provide different implementations for each platform, or we do not rely on the special features supported by the underlying platform. The first approach is doable, but requires significant programming effort. Although the interfaces of the library functions are the same, we require different implementations that are optimized for different platforms. Because the optimization strategies on different platforms differ considerably, the required programming effort is roughly linearly proportional to the number of platforms. The second approach instead uses only features that are compatible for all available platforms, but as a result, the code is not optimized for any platform. Because our top priority is efficiency, we sacrifice portability and focus on a subset of parallel hardware platforms.

- **Flexibility**: Flexibility measures how much customization an end user can perform. Turing completeness [29] is the flexibility level that all conventional general purpose programming languages achieve. Under such a flexibility level, an end user can perform any operation as long as it can be performed by a Turing machine. Fixed functionality is the least flexible case – an end user cannot perform any customization. We do not propose designing a programming language to bridge the implementation gap. Instead, we propose using a library to bridge it. Therefore, the flexibility we provide is restricted. The functions optimized in Chapter 5 are all fixed, and do not support user customization. The functions optimized in Chapter 6 are more flexible – users

can customize the sparse matrix format for the *clSpMV* autotuner, and the distance function for the *clPaDi* autotuner.

These factors define the scope of the parallel application library. Efficiency is our top priority, productivity is achieved by using application-level software architectures, portability is not a major concern, and flexibility is very restricted. Because flexibility is restricted, in order to make the library applicable to all major state-of-the-art object recognition systems, we must cover most computations used by these object recognition systems. An application can be refined by a hierarchy of different abstraction levels. In Section 3.2, we introduce our application-level software architecture, and discuss the abstraction level of the proposed parallel library. In Section 3.3, we perform pattern mining in state-of-the-art object recognition systems, and summarize the application patterns that we want to support in the parallel application library.

3.2 Application-Level Software Architecture

A software architecture describes the organization of a given application, and has two essential components: computations and control flow. Each software architecture typically includes many computations, which tell us what they are capable of but not how they achieve the goal. They serve as black boxes in the software architecture. Therefore, the computations provide an abstraction level to describe the application. Conversely, each software architecture has only one control flow. The control flow tells us how the computations relate to one another. The software architecture of an application can be hierarchical. The topmost level of the hierarchy describes the application in the most abstract way. If we want to know more about a computation, we can refine the computation into the lower hierarchy. This procedure continues until we are satisfied with the details we want to know about the application.

Application-level software architecture is the level where the refinement procedure terminates at the domain specific terminologies. The computations corresponding to the domain specific terminologies can be further refined, but the implementation details of these computations do not influence the output quality of the application. For example, the eigen-decomposition computation can be used in contour detection. From the viewpoint of application developers, as long as we can determine the eigenvalue and eigenvector pairs, we do not need details of the eigen-decomposition computation. Therefore, the application-level software architecture stops at the eigen-decomposition computation, and does not further refine into any lower layer hierarchies.

Figure 3.1 shows an example of the software architecture of an object recognition system. The topmost level of this hierarchy is shown in Figure 3.1(a). In this level of hierarchy, there is only one computation, which takes input images and summarizes the detected objects from the images. To know more about how the objects are detected, we proceed down to the next finer level of the hierarchy. The object recognition computation is refined in Figure 3.1(b). There are two computations in this level of hierarchy: feature extraction and classification. The control flow tells us that the features are extracted, and then the classification uses the

features to recognize the objects. Again, if we want to know more about how the features are extracted, we can go down the hierarchy to the third level. There are many features to extract from an image, Figure 3.1(c) shows the software architecture of extracting contour features for each image region. In this level of hierarchy, we have three computations: contour detection, segmentation, and contour feature extraction. The control flow describes that we need to detect contours from images, use contours to segment image into regions, and then compute the contour features for each image region. The refinement procedure can continue until we are satisfied with the details explained by the software architecture.

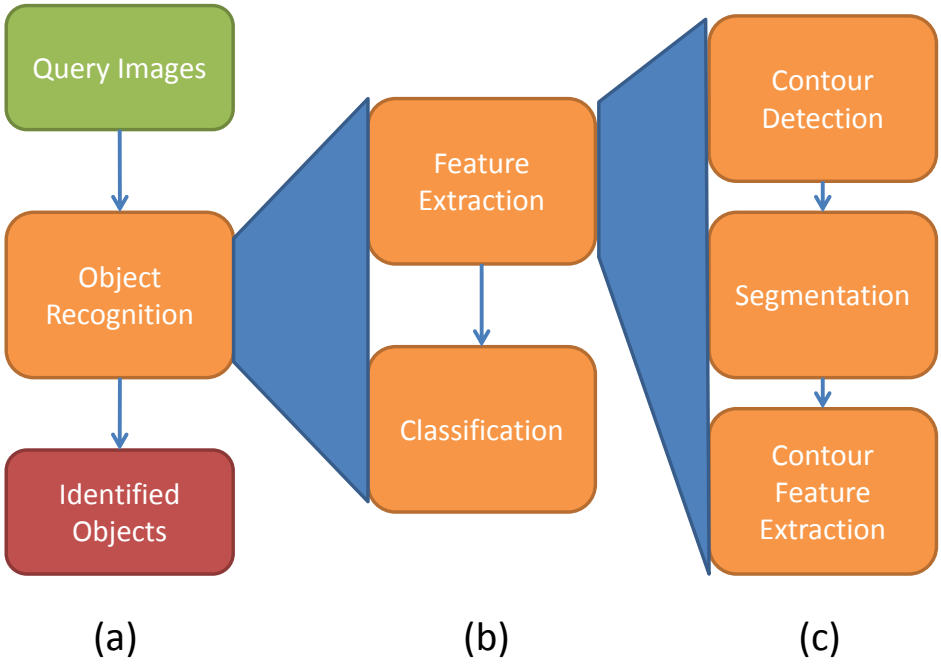

Figure 3.1: The application-level software architecture for an object recognition system.

The software architecture for an application is not unique. Depending on how we want to describe an application, we can have software architectures with different hierarchies, different computations, and different control flows. This flexibility is very useful when developing applications. For example, in Figure 3.1, in the second level of hierarchy the object recognition computation is refined into feature extraction and classification. When refining the feature extraction computation to the next level of hierarchy, we can propose different software architectures for our application. One software architecture uses SIFT [80], another uses HOG [33], and another uses contour features. Based on these different software architectures, the application developer can use domain knowledge to choose the one that best satisfies the application's purpose. If discussion is needed before making the final decision, the software architectures of different choices can help application developers to communicate and reason regarding the advantages and disadvantages of different implementations.

Different hierarchies in the software architecture define different abstraction levels – upper hierarchies are more abstract, while lower hierarchies describe more details. For our proposed parallel application library, the functions we support correspond to the computations, and the application developer follows the control flow in the software architecture to bind the computations together. The abstraction level in the library is a trade-off between productivity and flexibility. The higher the abstraction level, the simpler the control flow. The application developers can build the application more quickly, but do not have much flexibility to manipulate the application. The lower the abstraction level, the more complicated the control flow. The application developers will need more time to build the application, but are able to adjust for more detailed computations. For object recognition applications, there is no one dominating algorithm – researchers are exploring different features and different classification models. Therefore, flexibility is more important than productivity. As a result, the functions in the proposed parallel application library follow the finest application-level software architecture. That is, these functions are the ones whose detailed implementations do not influence the quality of object recognition systems, and so application developers would rather use them as black boxes.

3.3 Application Patterns for Object Recognition

An application pattern is a generalizable solution to a commonly occurring application computation. In order to understand the key application computations in state-of-the-art object recognition systems, we performed pattern mining [104] on 31 state-of-the-art papers. These papers have been selected from CVPR 2007 to CVPR 2011, in the object recognition track with oral presentations. The application patterns from these 31 papers are summarized in Table 3.1, in decreasing order according to their appearance frequency in the 31 papers. The object recognition systems of these 31 papers can be developed by composing these 15 application patterns together.

- `Convolution`: The convolution pattern updates the value of a pixel according to nearby pixels. One common computation is to apply a filter to the image. The value of a pixel is updated by a weighted sum over nearby pixels. The weights and the neighboring pixels are defined by the filter. Convolution can be used to perform Gaussian smoothing, compute gradients, and compute Hessians [113]. Almost all feature descriptors rely on this computation, including SIFT [80], HOG [33], and contour [81]. Another common computation is non-max suppression, in which a pixel is suppressed if it is not a local maximum. Whether a pixel is a local maximum or not can be detected by checking nearby pixels – if it has a neighbor with a larger value, then it is not a local maximum. This computation is used in segmentation-related computations [7, 61].

- `Histogram Accumulation`: The histogram accumulation pattern updates the value of a set of histogram bins according to input vectors. This computation is widely employed in many feature descriptors, including SIFT [80], HOG [33], and contour

Table 3.1: Pattern mining from 31 state-of-the-art object recognition papers.

Application Pattern	Number of Papers
Convolution	30
Histogram Accumulation	29
Vector Distance	22
Quadratic Optimization	15
Graph Traversal	9
Eigen Decomposition	6
K-means Clustering	6
Hough Transform	4
Nonlinear Optimization	4
Meanshift Clustering	2
Fast Fourier Transform	1
Singular Value Decomposition	1
Convex Optimization	1
K-medoids Clustering	1
Agglomerative Clustering	1

[81]. The primary motivation for using a histogram is fault tolerance. The same object can have different appearances in different images – the brightness can be different because light sources are different; the shape can be different because the object has moved. If we use an exact value to identify objects, it is too strict to find exact matches. However, if we discretize the values into histogram bins, the computation can tolerate slight differences and be more robust.

- `Vector Distance`: The vector distance pattern computes the distance between two vectors of the same dimension. This is usually used in model-building and classification computations. For example, the k-nearest neighbor [8] algorithm computes the distances between query vectors and exemplar vectors. Another common usage is for Support Vector Machine (SVM) classification [31], when we compute a kernel function between the query vector and the support vectors. If the distance function is defined as the kernel function, the SVM classification can be described as computing a weighted sum over the distances between the query vector and support vectors. Because SVM and nearest neighbor approaches are very common in machine learning algorithms, many object recognition systems use this computation [95, 129, 130].

- `Graph Traversal`: The graph traversal pattern traverses a graph, and can be used in a number of ways. One way is to apply graph traversal on images. An image graph is constructed by making every pixel a node and adding edges between nearby pixels. The image graph traversal pattern is commonly employed in segmentation and region related computations [7, 61, 38]. Another way is to apply graph traversal on graphical models. If the classification model is built based on graphical models, we must traverse the graph to update the probability at each node. Some object recognition systems

use graphical models, and thus apply graph traversal on the model [6]. Another way is to use a graph to represent the relationships among parts of an object [74].

- `Optimization`: The `Quadratic Optimization`, `Nonlinear Optimization`, and `Convex Optimization` patterns are all related to optimizations. When designing a model to recognize objects based on features, the goal is to maximize the probability that the objects are successfully detected. Therefore, the modeling problem is usually formulated as an optimization problem. If the objective function is quadratic, it is a quadratic optimization problem. If the objective function is convex, it is a convex optimization problem. If the objective function is nonlinear, it is a nonlinear optimization problem. For example, the SVM training procedure is formulated as a quadratic optimization problem [31].

- `Clustering`: The `K-means Clustering`, `Meanshift Clustering`, `K-medoids Clustering`, and `Agglomerative Clustering` patterns are all clustering algorithms. Clustering algorithms group sample points into clusters, and different clustering algorithms can be used in different situations. Both k-means clustering and k-medoids clustering patterns partition the data set into k clusters. K-means uses center of mass as the centroids of the clusters, while k-medoids uses sample points as the centroids of the clusters. Meanshift clustering does not require the user to specify the number of clusters – it iteratively groups nearby sample points together until convergence. Agglomerative clustering uses a greedy approach to group sets with the highest similarity together. Such clustering patterns are frequently employed to find visual words or codebooks for images [125, 76]. Clustering patterns can also be used to identify textons in images [81, 77].

- `Hough Transform`: The Hough transform pattern finds objects from images via a voting procedure among possible candidates. This pattern is commonly used to identify objects using parts of the object. In this case, every recognized part refers to a possible candidate. The more parts we recognize, the more likely an object appears in the image. This pattern is used in many region-based object recognition systems [61, 54, 13].

- `Eigen Decomposition`: The eigen-decomposition pattern finds eigenvalue and eigenvector pairs of a given matrix. The most well-known image segmentation work is the normalized graph cuts algorithm proposed by Shi and Malik [108]. Such an algorithm transforms the graph cuts problem into an eigen-decomposition problem. As a result, many segmentation-based object recognition systems use the eigen-decomposition pattern [38, 61].

- `Singular Value Decomposition`: The singular value decomposition pattern finds the unitary matrices and singular values of a given matrix. It is used in the Principal Component Analysis (PCA) algorithm to find the components with the largest variances – the larger the singular value, the larger the variance. The PCA algorithm can reduce

a high-dimensional data space to a low-dimensional data space. Guillaumin et al. [62] use this pattern to perform PCA in their image classification system.

- `Fast Fourier Transform`: The fast Fourier transform pattern is used to convert the original basis to the frequency domain. Because the convolution in the time domain is equivalent to multiplication in the frequency domain, this pattern can be used to accelerate convolution computations [40].

These 15 application patterns cover the design space of the 31 papers we collected. Because these papers describe cutting-edge object recognition systems, they represent existing state-of-the-art research. Therefore, we propose using this set of 15 application patterns in our parallel application library. By supporting the functions of these application patterns, we can develop many highly accurate object recognition systems.

3.4 Summary

In this chapter, we propose our solution to bridge the implementation gap between object recognition application developers and expert parallel programmers. We propose to design a parallel application library as an intermediate layer. Expert parallel programmers can then focus on optimizing the functions supported in the library, while object recognition application developers can develop their applications by using the library functions.

When designing the parallel application library, efficiency is our major concern. We want to provide a library that is highly efficient on the underlying parallel hardware platforms. Productivity is defined by the abstraction level we provide. Portability has lower priority, because different hardware platforms favor different parallelization strategies, and it is time consuming to optimize the same function across many different platforms. Flexibility is restricted – there are few customizations an user can perform on the functions. The goal is to enable users to develop parallel object recognition systems, not enable them to perform general purpose programming.

The abstraction level of the proposed parallel application library is decided according to application-level software architecture. A software architecture is a hierarchical description of the organization of an application using domain-specific terminologies. We use the lowest hierarchy of the application software architecture as the abstraction level in the parallel application library. The computations at this level of the hierarchy do not influence the quality of the object recognition systems, and therefore the application developers would rather use them as black box function calls.

Finally, in order to understand the functions we need to support in the parallel application library, we performed application pattern mining on 31 state-of-the-art object recognition papers. The result is the 15 application patterns in Table 3.1. By supporting these 15 application patterns, we can develop the object recognition systems in the 31 papers by composing these patterns together. Therefore, we propose to use these 15 application patterns in our parallel application library.

Chapter 4

Pattern-Oriented Design Space Exploration

In Chapter 3, we proposed using a parallel application library to bridge the implementation gap. As expert parallel programmers, we can focus on optimizing functions supported by the library. However, as discussed in Section 2.2, parallel programming can be very challenging – there is no one-size-fits-all solution to parallelize and optimize all application computations on all parallel platforms. Instead, we must explore the design space, find different ways to parallelize and optimize the application computations, evaluate the merits of different strategies, and make our final implementation decisions. In this chapter, we present a systematic method of exploring that design space using software architectures and patterns.

4.1 Implementation-Level Software Architecture

As introduced in Section 3.2, a software architecture describes the organization of an application. This can be hierarchical: a higher level of a hierarchy is more abstract, and a lower level of hierarchy explains more details. Every level of the hierarchy is composed of many computations and a control flow. A computation provides an abstraction over the capability of a black box, while a control flow describes how the computations relate to one another. Application-level software architectures are used by application developers, and the computations in these architectures correspond to domain-specific terminologies. These terms form the language that application developers use on a daily basis to communicate and to make design decisions for applications. On the other hand, implementation-level software architectures are used by expert parallel programmers, and reflect how an application computation is implemented and parallelized on a parallel hardware platform. If necessary, the architecture can be refined to instruction-level, meaning that every black box is an instruction.

4.1.1 Patterns and Our Pattern Language

Application developers use application patterns to communicate and make design decisions. Similarly, expert parallel programmers need a language for communication and implementation decisions. Further, if we want to support any arbitrary application, we need a language that is general enough to describe all kinds of different workloads in applications. Keutzer and Mattson [73] propose Our Pattern Language (OPL), which decomposes all computer science workloads into a collection of patterns. This perfectly matches our goal; therefore, we use OPL to describe implementation-level software architectures.

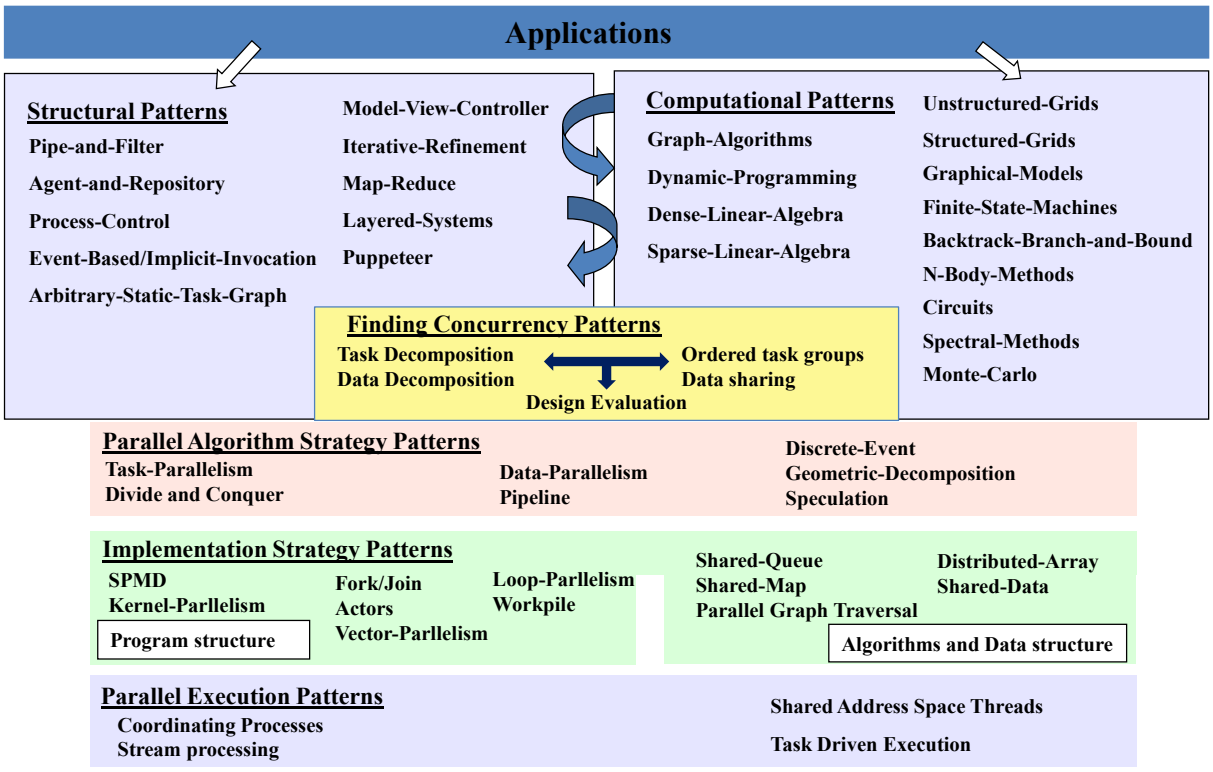

Figure 4.1: Organization of Our Pattern Language (OPL).

Our Pattern Language (OPL) is summarized in Figure 4.1. The topmost level of OPL is composed of structural and computational patterns. We use these two categories of patterns to describe the implementation-level software architectures of application computations. The structural patterns describe control flows in software architectures, while the computational patterns describe the workloads. Every level of the hierarchy be defined by connecting computational patterns using structural patterns.

The computational patterns in OPL summarize all kinds of workloads in computer science. Therefore, we can use these patterns to categorize the fundamental ideas of the application patterns. The 15 application patterns for object recognition described in Sec-

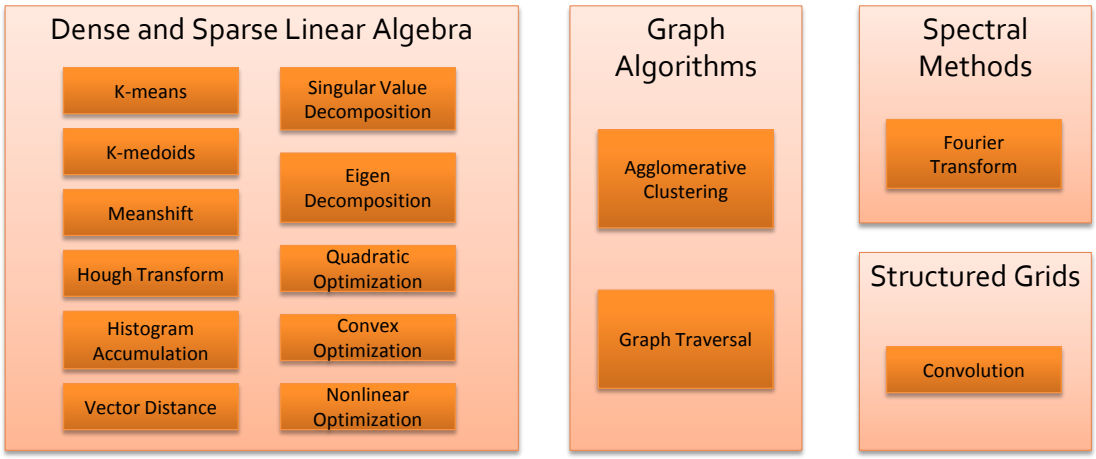

Figure 4.2: The computational patterns that cover the object recognition application patterns.

tion 3.3 can be categorized as in Figure 4.2. We can cover all 15 application patterns with only five computational patterns (Dense Linear Algebra and Sparse Linear Algebra are two computational patterns). Basically, the clustering, Hough transform, vector distance, singular value decomposition, eigen decomposition, and optimization patterns can be solved by linear algebra operations. If the input data is dense, we apply dense linear algebra operations; if the input data is sparse, we apply sparse linear algebra operations. Agglomerative clustering and graph traversal can be solved by graph algorithms. The fast Fourier transformation pattern is a subset of the spectral methods pattern, and the convolution pattern can be solved by the structured grids method. These five computational patterns are widely used in the high performance computing (HPC) field. As a result, we can take advantage of the existing literature on these patterns to optimize and parallelize our own application patterns.

Below the structural and computational patterns are many layers of other patterns. These patterns do not explicitly appear in software architectures – they are instead used primarily for helping expert parallel programmers understand application computations and parallel hardware platforms. With the information captured by these patterns, expert parallel programmers can design better software architectures that cleanly map application computations to underlying parallel hardware platforms. Finding concurrency patterns can provide a guideline for identifying parallelism in workloads. For example, we can decompose the workloads by task, by data, or by both. Our decision at this level brings us to the parallel algorithm strategy patterns. If we decompose the workloads by data, for example, then very likely we would like to express data parallelism in the workloads. Implementation strategy patterns summarize the programming models and data structures we can use to express the parallelism. For example, if we need to perform the same instruction on every element of a

vector, we can apply the vector parallelism programming model on the shared data. Parallel execution patterns summarize the parallel hardware supports that enable the programming models. Readers are referred to Mattson et al. [89] for more details about these lower-layer patterns.

4.1.2 Architecting Computations Using Patterns

By using structural patterns to describe software organization and computational patterns to describe workload, we can design implementation-level software architectures for all kinds of computations. Figure 4.3 shows an example of designing the implementation-level software architecture of the Euclidean distance computation. Given vectors $a = [a_1 a_2 \dots a_n]$ and $b = [b_1 b_2 \dots b_n]$, the Euclidean distance d between vector a and b is defined as the following:

$$d = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{4.1}$$

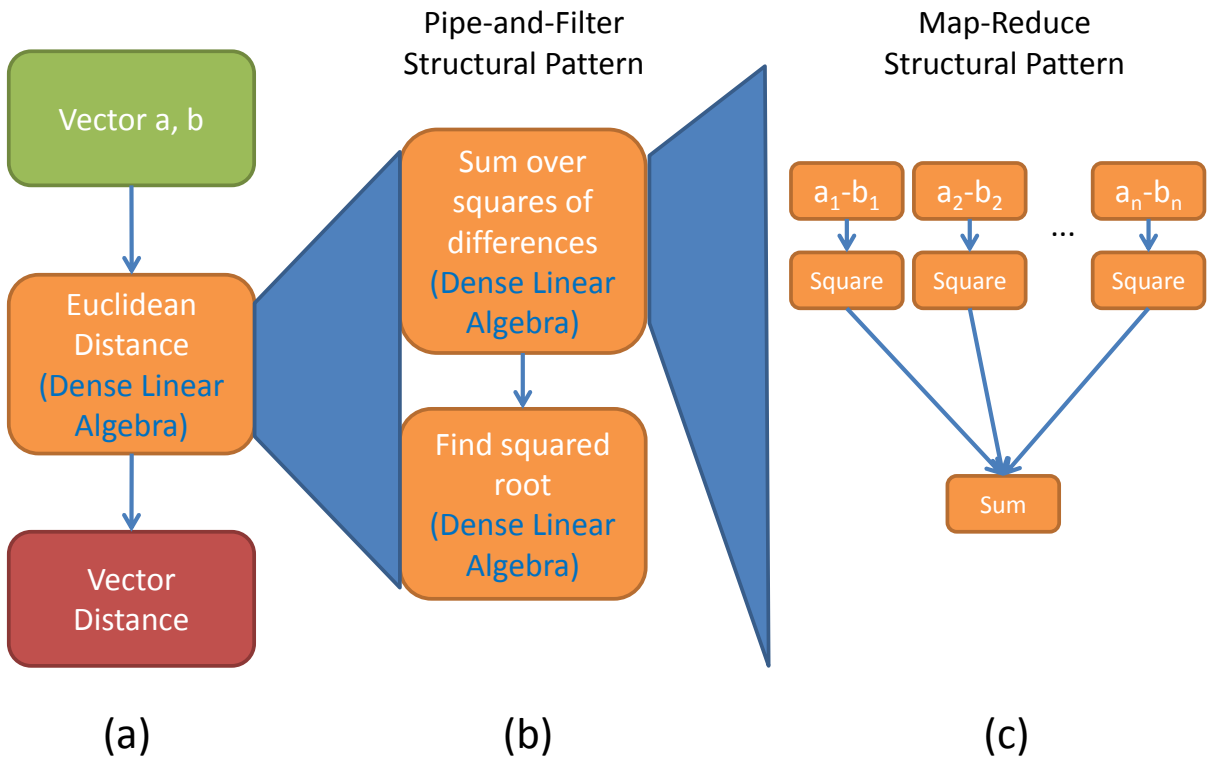

Figure 4.3: The implementation-level software architecture of the Euclidean distance computation.

Before explaining the software architecture of the Euclidean distance computation, we must introduce the patterns used in the architecture.

- `Pipe-and-Filter Structural Pattern`: This computation is organized by a sequence of filters. Each filter takes input data from the preceding filters, operates on the input data, and then passes the output to other filters.
- `Map-Reduce Structural Pattern`: This computation is organized in two phases, a map phase following by a reduction phase. The map phase includes a collection of independent computations, and the reduction phase summarizes the results generated from the map phase.
- `Dense Linear Algebra Computational Pattern`: This is a union of computations that can be described by performing arithmetic operations on dense arrays of data.

The topmost level of hierarchy of the Euclidean distance software architecture is shown in Figure 4.3(a), represented by a single computational pattern: dense linear algebra. This can be further decomposed to Figure 4.3(b), where the pipe-and-filter structural pattern is used to describe the relationship between two computations. We must compute the summation over the squares of differences, then compute the square root of this value. These two computations can be identified as dense linear algebra workloads. The first computation can be further refined into Figure 4.3(c), where we have many independent (a_i, b_i) pairs. We compute the difference of each pair, the square of the difference, and add the results from all pairs together. This computation flow can be described by the map-reduce structural pattern. Again, every computation in this level of hierarchy is a dense linear algebra computation.

By analyzing the structure of the software architecture, we can understand the dependency of the computations and hence the parallelism in the architecture. In Figure 4.3(b), the second computation is dependent on the first computation, so we cannot express parallelism at this level. In Figure 4.3(c), every pair is independent, and we can express parallelism across the pairs.

In summary, an implementation-level software architecture is defined by hierarchically decomposing a computation using structural and computational patterns. Different hierarchies of the architecture reveal different opportunities for parallelizing the computation. Using patterns to architect computations gives us a systematic way of describing, analyzing, and parallelizing computations.

4.2 Design Space

A software architecture describes the implementation of a computation; however, usually a computation can have many different implementations. The only explicit components of a computation are inputs and outputs. An implementation is valid as long as the intermediate steps correctly convert the inputs to outputs. When a computation is complicated enough, there will be many valid implementations to accomplish the conversion procedure. The

superset of all valid implementations forms the design space of a computation. Different implementations have different characteristics, such as computational complexity, dependency, granularity of parallelism, memory access behavior, and so forth. In order to optimize and parallelize the implementation of a computation, we must evaluate all of these different implementations to find one that best performs on the underlying platform. As a result, it is essential to understand the design space when analyzing the strengths and weaknesses of different implementations. An explorable design space can be divided into three layers: algorithms, parallelization strategies, and platform parameters. These layers are introduced in the following sections.

4.2.1 The Design Space of Algorithms

The algorithm layer explores different procedures for correctly converting inputs to outputs, and is the most important layer of the three. There is considerable variety among algorithms. Some have strict data dependency, and are naturally serial; others have many independent tasks, and can be parallelized. Some algorithms have poor data access patterns, and achieve very low memory bandwidth; others have regular data access patterns, and can saturate available memory bandwidth. Some algorithms have higher computational complexity; others have lower computational complexity. Without exploring the design space of algorithms and choosing a proper algorithm, we cannot get any significant benefit from exploring the design space of parallelization strategies and the design space of platform parameters.

We use the prefix sum computation as an example of the design space of algorithms. Given an input array $a = [a_1, a_2, \ldots, a_n]$ with n elements, the prefix sum of a is an array of n elements, $b = [b_1, b_2, \ldots, b_n]$, such that $b_i = \sum_{k=1}^{i} a_k$. This is a simple computation, but still many different algorithms can achieve the same goal. In addition to the patterns introduced in Section 4.1.2, we now need one additional pattern to explain the software architectures used in the algorithms:

- `Iterative-Refinement Structural Pattern`: The computation is organized by performing a set of operations repeatedly until a predefined termination condition is reached.

A naive algorithm for the prefix sum computation is shown in Figure 4.4. Figure 4.4(a) shows the implementation-level software architecture, while Figure 4.4(b) shows the data dependency graph of the algorithm. The software architecture is simple – we use the iterative-refinement structural pattern to go over each element in array b in a serial fashion. Each element is computed by adding its predecessor to one element in array a. The computational complexity is $O(n)$, but the algorithm is serial. The strict dependency on array b forbids us from parallelizing the algorithm.

An algorithm with massive parallelism is shown in Figure 4.5. The software architecture of this algorithm uses the map-reduce structural pattern to map the computation of each element in array b independently. The amount of parallelism is at least the size of array b. However, the computational complexity increases to $O(n^2)$. Moreover, the workloads are

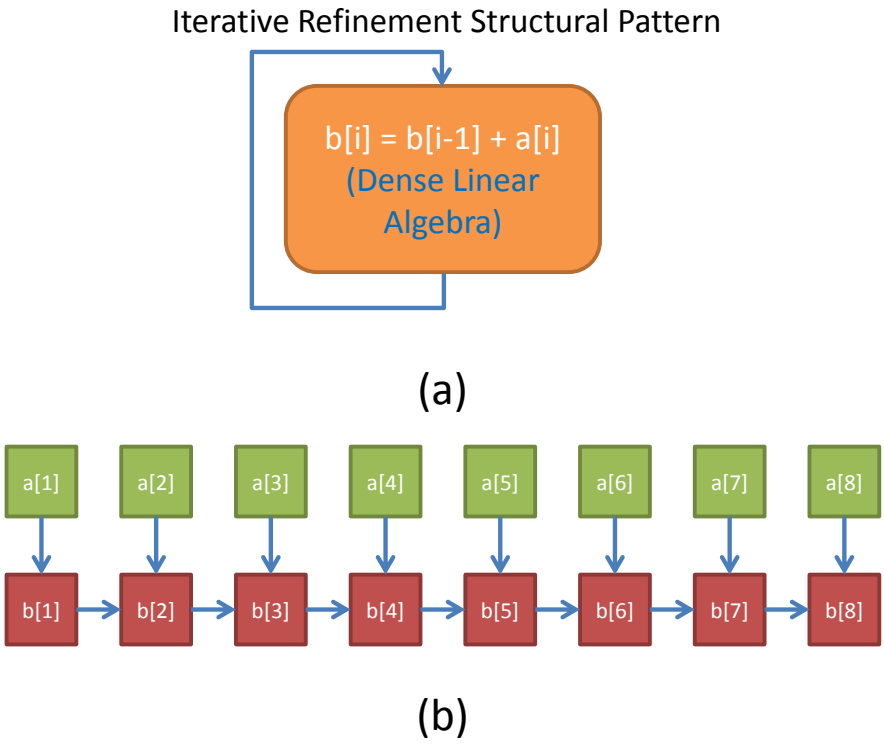

Figure 4.4: The implementation-level software architecture and data dependency graph of a serial prefix sum algorithm.

extremely unbalanced. For $b[1]$, only an assignment operation is necessary. Conversely, we need to sum over $a[1]$ to $a[n]$ to compute the value of $b[n]$. Even if we are able to distribute workloads evenly to all available processing units, we still require more than n processing units to make the parallel algorithm faster than the serial algorithm. This is definitely not a scalable approach, as data size is typically orders of magnitude larger than the processor number.

A better parallel algorithm for prefix sum proposed by Hillis and Steele is shown in Figure 4.6 [66]. The first level of the software architecture is an iterative-refinement structural pattern. We divide the computation into $\log_2(n)$ levels. In level i, we compute the exact sum of the first 2^i elements in b, and the partial sum of the remaining elements in b. The computations at each level are described in the second level of hierarchy. We use the map-reduce structural pattern to map the computation of each element in b independently. In a level, the operations on each element in b are uniform – we read two values, add them together, and update the current value. Therefore, it is easy to balance the workloads of the processing units. The computational complexity of this algorithm is $O(n \log_2 n)$. As long as the number of processing units is larger than $\log_2 n$, we should see a speedup compared to the serial algorithm. Even if the data size is 2^{32} (2^{32} is the largest integer on a 32-bit system), we need only more than 32 processing units to make the parallel prefix sum algorithm faster

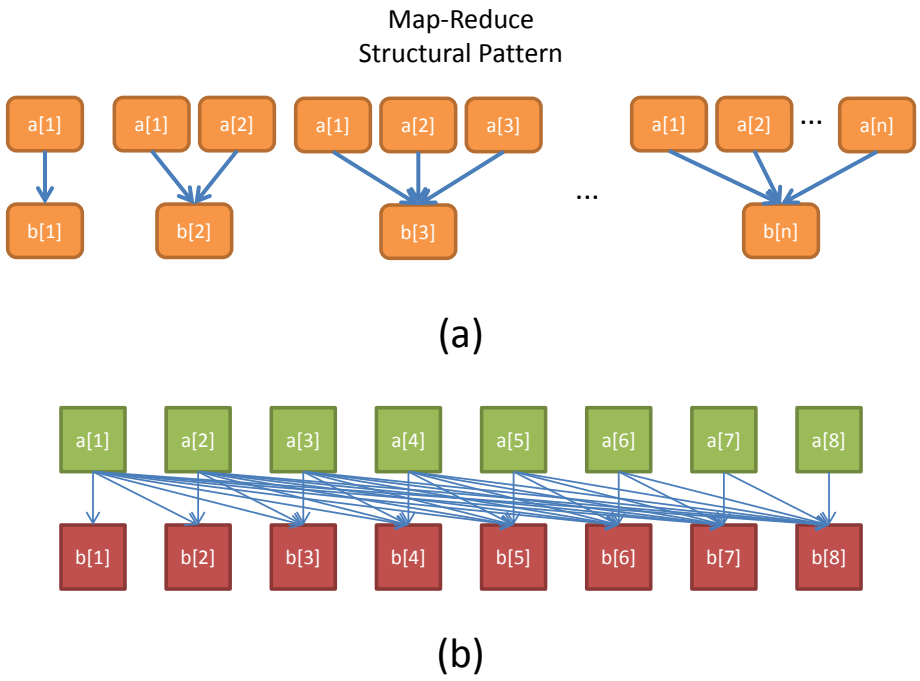

Figure 4.5: The implementation-level software architecture and data dependency graph of a bad parallel prefix sum algorithm.

than the serial algorithm. For example, Harris et al. achieved 10× speedup compared to a sequential implementation on a CPU by applying the parallel prefix sum algorithm on an Nvidia Geforce 8800 GTX card [64], which has 128 processing units.

In summary, different algorithms express different dependency on the computations. In order to parallelize and optimize a computation on a given platform, it is essential to explore the design space of algorithms. Using software architectures to describe algorithms gives us a systematic method for analyzing the strengths and weaknesses of the algorithms. Based on this analysis, we can choose an algorithm that best fits the capabilities of the underlying hardware platform, and so achieve the best performance.

4.2.2 The Design Space of Parallelization Strategies

The parallelization strategy layer explores the design space of expressing parallelism on an algorithm. In this layer, we do not change or modify the given algorithm, instead we only express the parallelism on that algorithm. Different parallelization strategies result in different utilizations of the underlying hardware platform. In exploring this design space, we try various parallelization strategies to determine the most efficient method of utilizing the underlying hardware platform.

Expressing parallelism on an algorithm can be achieved by annotating the corresponding

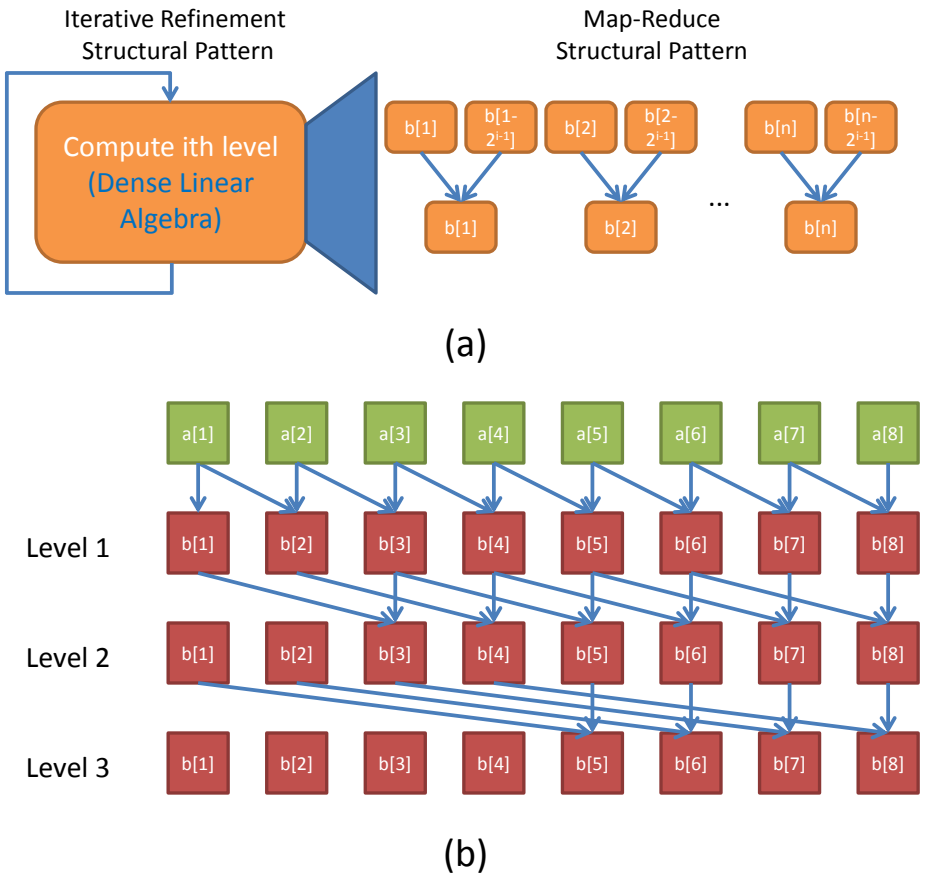

Figure 4.6: The implementation-level software architecture and data dependency graph of a better parallel prefix sum algorithm.

software architecture. Figure 4.7 shows two parallelization strategies for the parallel prefix sum algorithm introduced in Figure 4.6. In the parallel prefix sum algorithm, every element in array b can be computed in parallel. Because the number of available processing units is usually smaller than the array size, we must assign many tasks to a processing unit. Moreover, in modern processor architectures, one processing unit usually hosts many threads to hide memory latency, because memory latency is longer than the instruction execution time. We therefore annotate the software architecture with thread indices. Figure 4.7(a) shows the parallelization strategy of dividing the array into two partitions, and assigning one thread per partition. Figure 4.7(b) shows the strategy of assigning tasks to two threads in an interleaved manner.

Different platforms have different characteristics, and require different parallelization strategies. A CPU platform is composed of several big processors, each with its own cache hierarchy. If the strategy in Figure 4.7(a) is employed, each processor accesses only the data it needs. Conversely, if the strategy in Figure 4.7(b) is used, because the memory transaction is managed at the granularity of the cache-line-size, when accessing data in array b, other

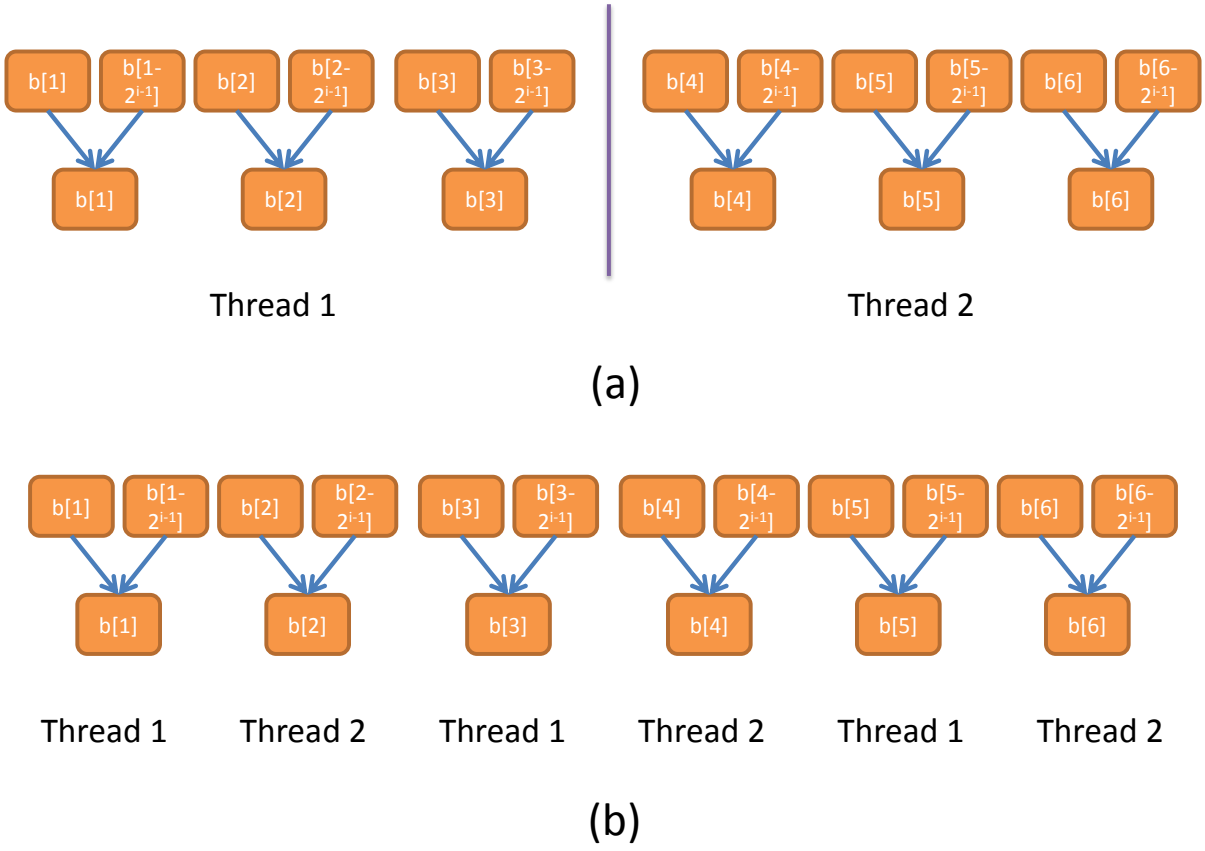

Figure 4.7: Different parallelization strategies for the parallel prefix sum algorithm.

data nearby will loaded as well. To be more specific, if $b[1]$ and $b[2]$ are in the same cache line, data in $b[1]$ and data in $b[2]$ will be loaded together in the same memory transaction. As a result, each processor will load redundant data, wasting memory bandwidth. The first parallelization strategy should be chosen if the execution platform is a CPU.

On the other hand, a GPU platform is composed of many small processors. A group of processors forms a single instruction multiple data (SIMD) lane. A SIMD lane accesses consecutive data and performs the same operations on those data. For the strategy in Figure 4.7(a), the two threads are not accessing consecutive data. Two memory transactions are needed before the two threads can begin to execute instructions. Conversely, for the strategy in Figure 4.7(b), the two threads are accessing consecutive data, so only one memory transaction is required to load the data for the SIMD lane to process. As a result, the second parallelization strategy should be chosen if the execution platform is a GPU.

In summary, the design space of parallelization strategies covers different ways of expressing parallelism on an algorithm. In order to optimize the implementation on a given platform, we must explore this layer and determine the proper strategy based on the capa-

bility of the underlying hardware platform.

4.2.3 The Design Space of Platform Parameters

All parallelization strategies are coupled with platform parameters. The platform parameter layer defines the possible combinations of valid parameter settings on the underlying hardware platform.

If we want to implement the parallel prefix sum algorithm using the strategy in Figure 4.7(a) on a CPU platform, we can define at least three platform parameters: thread count, partition size, and loop unrolling factor. Thread count is the number of threads we want to create. Different CPUs have different numbers of cores, and one core can host one or more threads. For example, an Intel Core i7 920 has four cores and supports hyperthreading technology [69], so maximally eight threads can be hosted on that CPU concurrently. Valid configurations therefore include from one to eight threads. Partition size controls how the array is partitioned. Partitioning the array evenly is an option. However, because the memory transaction is at the granularity of the cache-line-size, when reading data at the end of a partition we might also bring in data that belong to the next partition. These redundant data waste memory bandwidth. If we can arrange the partition size to ensure that it is a multiple of the cache-line-size, we can better utilize the available memory bandwidth. The cache-line-size of different platforms might be different, so we must try out different partition sizes to find the best fit for our targeting platform. A valid parameter setting will force the partition size to be multiple of 1 (do not pad the partition at all), 2, 4, 8, 16, 32, 64, or 128 bytes – eight valid configurations. Loop unrolling is an optimization method that reduces the overhead of loop control. This parameter is set according to how aggressively we want to unroll the loop. Assuming we are interested in unrolling the loop at most eight times, then we will have eight valid configurations. The total number of valid configurations of our implementation is therefore 8(thread count) $\times$8(partition size) $\times$8(loop unrolling factor) = 512.

In the prefix sum example, we have introduced three algorithms, two parallelization strategies, and 512 platform parameters. The design space of platform parameters is orders of magnitude larger than the other two layers because platform parameters are defined by numerical values, and there can be many valid numerical values for a given parameter. When multiplying the configurations of each parameter together, the total number of valid configurations grows rapidly. Fortunately, unless we want to achieve absolute peak performance of the underlying platform, we do not need to thoroughly explore the design space of platform parameters – reasonable assignments should be sufficient. For the previous example, setting the thread count to eight, partition size to a multiple of one byte (do not pad the partition), and loop unrolling factor to one should work just fine. Eight is the maximum number of threads that can be hosted on the CPU. The size of the input data is typically significantly larger than the cache size. When partitioning the input data, only the head and the tail of the partition might waste the memory bandwidth. Even if we do not pad the partitions, the overhead is small. Loop unrolling factor reduces the overhead of the loop, but it does not boost performance significantly. The best possible configuration can likely

outperform this setting by several percent, but it will never be several times faster.

In summary, the design space of platform parameters covers the valid configurations of parallelization strategies. This space is very large, but most of the time we can rely on our architecture knowledge and experience with the platform to pick reasonable configurations. Moreover, the performance gain by exploring this layer is smaller compared to the other two layers.

4.3 Design Space Exploration

Every computation can have many implementations, and different implementations perform differently on different hardware platforms. The superset among all valid implementations forms the design space. For our parallel application library, the goal is to accelerate object recognition systems. Therefore, the primary objective is execution time. Given a computation, input data, and a parallel platform, we would like to find an implementation from the design space such that the total execution time is minimized. In order to find such optimal implementation, it is essential to explore the design space, compare different designs, and find the one that outperforms the others.

The termination criteria of design space exploration is arbitrary. It is very hard to find a theoretical estimate of the minimum execution time. Even if an implementation completely saturates the underlying hardware platform, it is very difficult to prove that no other algorithms exist that requires fewer operations and can further reduce the execution time. Moreover, theoretical bounds might not be always true in practice. Take sorting algorithms as an example, under the worst case analysis, the computational complexity of the quick sort algorithm is $O(n^2)$, while that of the heap sort algorithm is $O(n \log n)$. However, in practice, the quick sort algorithm usually requires fewer operations and is faster than the heap sort algorithm [30]. As a result, parallel programmers usually set up an arbitrary boundary on the design space to explore, and only examine implementations within this boundary. One common guideline of setting the boundary is based on prior work. That is, given a computation, compare all published works, and find the most efficient implementation.

When setting boundary of design space we want to explore, if we only want to compare different algorithms and parallelization strategies, we can apply an exhaustive search to find the optimal implementation. However, if we are also interested in platform parameters, since the numeric values of the parameters can expand the design space with an exponential rate, we need to explore the design space automatically.

4.3.1 Exhaustive Search

The exhaustive search method tries all valid implementations, and selects the one that performs best. It is a brute-force approach, but very accurate and effective. When we are only interested in evaluating different algorithms and parallelization strategies, we can apply this method to find the best implementation on a given hardware platform. Sometimes we can also prune the design space before applying an exhaustive search. For example, when

one design is obviously worse than other designs, we can ignore it when exploring the design space. In Section 4.2.1 we introduced three algorithms to implement the prefix sum computation. The second algorithm shown in Figure 4.5 is obviously worse than the other two, because the computational complexity is very large. We can therefore remove this algorithm from consideration and so reduce the size of the design space.

The exhaustive search method is widely employed in high performance computing research. When researchers propose a new parallel algorithm, they compare the performance with other existing algorithms. When researchers optimize a computation on a platform, they try out different parallelization strategies and report the best one. Harris et al. optimized the prefix sum computation on an Nvidia Geforce 8800 GTX platform [64]. When exploring the design space of algorithms, they tried $O(n)$ serial prefix sum, $O(n \log_2 n)$ parallel prefix sum by Hillis and Steele [66], and $O(n)$ parallel prefix sum by Blelloch [17]. When exploring the design space of parallelization strategies, they avoided memory bank conflicts by padding the arrays. Because of these efforts, they were able to achieve 20× speedup compared to the serial algorithm. Williams et al. optimized the sparse matrix vector multiplication computation on four multicore platforms: Intel Clovertown, AMD Opteron, Sun Niagara2, and STI Cell [123]. In the algorithm layer, they tested the performance of different sparse matrix data structures. In the parallelization strategy layer, they tried SIMD, software pipelining, prefetching, caching, thread assignment, and NUMA-aware affinity assignment. By extensively exploring the design space, they were able to claim the best performance on all four platforms.

Often the trade-offs among different algorithms and parallelization strategies are not obvious. If the size of the design space is not too large, employing the exhaustive search method to find the best implementation in the design space is both accurate and effective. We apply this method to the three case studies in the parallel application library in Chapter 5.

4.3.2 Autotuning

When the design space is too large to be manually explored, we must write programs that explore it automatically. These programs are called *autotuners*, and the procedure of automatically exploring a design space is called *autotuning*. Given a design space, an autotuner samples the space, evaluates the performance of each sampled point, and reports the best configuration. It can sample different algorithms, different parallelization strategies, and different platform parameters.

However, the autotuning procedure is very expensive because an autotuner must consider hundreds to thousands of sample points in the design space. As a result, researchers usually develop autotuners only for widely-used and expensive computations. For example, the matrix matrix multiplication computation has a computational complexity of $O(n^3)$, and is widely used in many linear algebra problems. Thus, it is important to optimize this computation as much as possible. ATLAS [122] and PHiPAC [16] are autotuners that researchers have developed to optimize matrix matrix multiplication on all single-core platforms. GotoBLAS [56] is an autotuner that optimizes this computation on multicore platforms. The

Sparse Matrix Vector multiplication (SpMV) computation is very important in sparse linear algebra. It has a computational complexity of $O(n^2)$, and is a memory-bounded problem. Im et al. developed the Sparsity autotuner [68] to optimize this computation on single-core platforms. Fast Fourier Transform (FFT) is used for many signal-processing applications, and although the computational complexity is $O(n \log n)$, the memory movement behavior is very complicated. FFTW [52] is an autotuner that optimizes the memory access pattern on single-core platforms.

Employing an autotuner to consider all the sample points in a design space will reveal the best implementation. However, when the design space is enormous, this strategy might not be practical. Researchers apply three approaches to accelerate this procedure. First, we use heuristics to reduce the design space, which requires expertise in architecture. The more we know about the characteristics of the underlying hardware platform, the better we can find good heuristics to prune the design space. Second, we can assume that each parameter is independent, and so search through each parameter independently. Datta et al. used this approach to autotune stencil computations on multicore and manycore platforms, achieving good results [36]. Third, machine learning algorithms can be employed to understand the big picture of the design space through a small subset of sample points. Ganapathi et al. used the Kernel Canonical Correlation Analysis (KCCA) machine learning algorithm to autotune stencil computations and significantly reduce autotuning time without sacrificing the quality of the results.

Overall, autotuners are powerful tools for exploring design spaces and optimizing computations on underlying platforms. Because autotuning is very time-consuming, it should be employed only on widely-used and expensive computations. We would ideally develop autotuners for the most commonly used application patterns. The OLOV project fulfills this goal by collecting autotuners for the key application patterns, and is introduced in Chapter 6.

4.4 Summary

In this chapter, we have proposed a systematic method for optimizing and parallelizing a computation. An implementation of a computation can be described by an implementation-level software architecture – a hierarchical composition of structural patterns and computational patterns, which are defined in Our Pattern Language (OPL) [73]. An implementation-level software architecture provides an overview of the structure and organization of an implementation. Based on this architecture, we can easily identify data dependency, and hence analyze the potential for expressing parallelism in the architecture.

A computation converts inputs to outputs. However, it does not specify how this conversion is accomplished. As a result, there are many implementations that can perform a given conversion procedure. The set of all implementations that correctly convert inputs to outputs forms a design space. Because different implementations perform differently on different hardware platforms, we must explore the design space to parallelize and optimize a given computation. The design space itself can be divided into three layers. The algorithm layer covers different steps of transforming inputs to outputs. Given an algorithm, the

parallelization strategy layer covers different approaches for expressing parallelism on the software architecture. Given a parallelization strategy, the platform parameter layer covers different parameter configurations associated with the strategy.

There are two approaches to exploring a design space. Exhaustive search manually tries all possible designs in the space and reports the best one. Researchers apply this approach to compare different algorithms and parallelization strategies when the design space is not too large. The autotuning approach automatically tries all or a subset of designs in the space and reports the best design. This is employed when the design space is enormous. Because the autotuning procedure is expensive, researchers generally develop autotuners only for widely-used and expensive computations. An autotuner will extensively explore the design space of algorithms, parallelization strategies, and platform parameters. As a result, the final implementation is highly optimized and very likely achieves the best performance available from the underlying hardware platform. We employ both approaches to parallelize and optimize our proposed parallel application library for object recognition, the details of which are introduced in Chapters 5 and 6.

Chapter 5

Case Studies of the Parallel Application Library for Object Recognition

Table 5.1: 15 important application patterns for object recognition.

Convolution	Histogram Accumulation	Vector Distance
Quadratic Optimization	Graph Traversal	Eigen Decomposition
K-means Clustering	Hough Transform	Nonlinear Optimization
Meanshift Clustering	Fast Fourier Transform	Singular Value Decomposition
Convex Optimization	K-medoids Clustering	Agglomerative Clustering

We summarized the key application patterns in Section 3.3. Table 5.1 shows the 15 important application patterns from our study. A parallel application library that covers all 15 application patterns can be used to develop most state-of-the-art object recognition systems. Before focusing on developing the entire library, we would like to showcase how much performance gain can be achieved by deploying the library in real object recognition systems. Therefore, in Chapter 7, we choose the object recognition system by Gu et al. [61], and evaluate the performance of accelerating the system with our library. In order to do so, we need the library to cover at least application patterns that are bottlenecks of the system. In this chapter, we introduce three case studies that are major bottlenecks of the system, and illustrate how we explore the design space to parallelize and optimize these application patterns.

5.1 Eigensolver for the Normalized Cut Algorithm

`Eigen Decomposition` is an application pattern in Table 5.1. Given a square matrix A, the eigen-decomposition problem finds eigenvalue λ and eigenvector v pairs such that the equation $Av = \lambda v$ is satisfied. An eigenvalue λ with its corresponding eigenvector v

forms an eigen-pair (λ, v). If the size of matrix A is $n \times n$, the number of distinct eigen-pairs ranges from 1 to n. There are two categories of different approaches to solve the eigen-decomposition problem: direct methods and iterative methods. Direct methods are employed when we need all or a significant portion of the eigen-pairs, while iterative methods are used when we require only a small portion of the pairs. Because we are interested in developing a library for object recognition, not for linear algebra, we can reduce the problem size by focusing on the eigen-problems in object recognition. Most segmentation-based object recognition systems rely on the normalized cut algorithm proposed by Shi and Malik [108]. Therefore, we want to parallelize and optimize the eigen-problem in the normalized cut algorithm.

An image graph $G = (V, E)$ can be constructed by making every pixel a node and then connecting neighboring nodes with edges. Partitioning the graph G is equivalent to segmenting the original image. The normalized cut algorithm defines a metric to measure the quality of a graph partition. Let the edge weight $w(i, j)$ be the similarity between pixel i and j. If the graph G is partitioned into disjoint node sets A and B, then the normalized cut cost of such a partition is defined as follows:

$$NormalizedCut(A, B) = \frac{Cost(A, B)}{Cost(A, V)} + \frac{Cost(A, B)}{Cost(B, V)}; \tag{5.1}$$

$$Cost(X, Y) = \sum_{x \in X, y \in Y} w(x, y). \tag{5.2}$$

The best segmentation of the original image corresponds to the graph partition with the minimum normalized cut value. Shi and Malik demonstrated that the NP-hard graph partitioning problem can be approximated by solving a generalized eigen-system [108]. So, to be more specific, we must solve the generalized eigen-problem:

$$(D - W)v = \lambda D v, \tag{5.3}$$

where W is an affinity matrix with $W_{ij} = w(i, j)$, and D is a diagonal matrix constructed from W: $D_{ii} = \sum_j W_{ij}$. Only the $k + 1$ eigenvectors v_j with the smallest eigenvalues are useful in image segmentation and need to be extracted. The smallest eigenvalue of this system is known to be 0, and its eigenvector is not used in image segmentation, which is why we extract $k+1$ eigenvectors. The generalized eigen-problem can be further transformed into a standard eigen-problem:

$$A\bar{v} = \lambda\bar{v}, \tag{5.4}$$

$$A = D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}. \tag{5.5}$$

5.1.1 Exploring the Design Space of Algorithms

The first step of exploring the algorithm design space is to find a proper eigen-decomposition algorithm. Matrix A is Hermitian and positive semi-definite, and its eigenvalues are well

distributed. Additionally, we need only a few of the eigenvectors, corresponding to the smallest $k+1$ eigenvalues. Considering the issues above, the Lanczos algorithm is a good fit for this problem [12], and is summarized in Figure 5.1. The complete eigen-problem has complexity $O(n^3)$ where n is the number of pixels in the image, but the Lanczos algorithm is $O(mn) + O(mM(n))$, where m is the maximum number of matrix vector products, and $M(n)$ is the complexity of each matrix vector product – $O(n)$ in our case. Empirically, m is $O(n^{\frac{1}{2}})$ or better for normalized cut problems [108], meaning that this algorithm scales at approximately $O(n^{\frac{3}{2}})$ for our problems.

Algorithm: Lanczos
Input: A (Symmetric Matrix)
v (Initial Vector)
Output: Θ (Ritz Values)
X (Ritz Vectors)

Start with $r \leftarrow v$;
$\beta_0 \leftarrow \|r\|_2$;
for $j \leftarrow 1, 2, \ldots,$ **until** convergence
    $v_j \leftarrow r/\beta_{j-1}$;
    $r \leftarrow Av_j$;
    $r \leftarrow r - v_{j-1}\beta_{j-1}$;
    $\alpha_j \leftarrow v_j^* r$;
    $r \leftarrow r - v_j\alpha_j$;
    Reorthogonalize if necessary ;
    $\beta_j \leftarrow \|r\|_2$;
    Compute Ritz values $T_j = S\Theta S$;
    Test bounds for convergence ;
Compute Ritz vectors $X \leftarrow V_j S$;

Figure 5.1: The Lanczos algorithm.

For a given symmetric matrix A, the Lanczos algorithm proceeds by iteratively building up a basis V, which is then used to project this matrix A into a tridiagonal matrix T. The eigenvalues of T are computationally much simpler to extract than those of A, and converge to the eigenvalues of A as the algorithm proceeds. The eigenvectors of A are then constructed by projecting the eigenvectors of T against the basis V. More specifically, v_j denotes the Lanczos vector generated by each iteration, V_j is the orthogonal basis formed by collecting all the Lanczos vectors $v_1, v_2, \ldots, v_j$ in column-wise order, and T_j is the symmetric $j \times j$ tridiagonal matrix with diagonal equal to $\alpha_1, \alpha_2, \ldots, \alpha_j$, and upper diagonal equal to $\beta_1, \beta_2, \ldots, \beta_{j-1}$. S and Θ form the eigen-decomposition of matrix T_j. Θ contains the approximation to the eigenvalues of A, while S in conjunction with V approximates the eigenvectors of A: $x_j = V_j s_j$.

There are three computational bottlenecks in the Lanczos algorithm, so we need to explore the algorithm design space for these bottlenecks to optimize the computation.

The first bottleneck is sparse matrix vector multiplication (line 5 in Figure 5.1). Because the matrix is very large ($N \times N$, where N is the number of pixels in the image), and the multiplication occurs in each iteration of the Lanczos algorithm, this operation accounts for approximately 2/3 of the runtime of the serial eigensolver.

Sparse matrix vector multiplication (SpMV) is a well-studied kernel in the domain of scientific computing, due to its importance in a number of sparse linear algebra algorithms. A naively-written implementation runs far below the peak throughput of most processors. This poor performance typically results from low efficiency of memory access to the matrix, as well as to source and destination vectors.

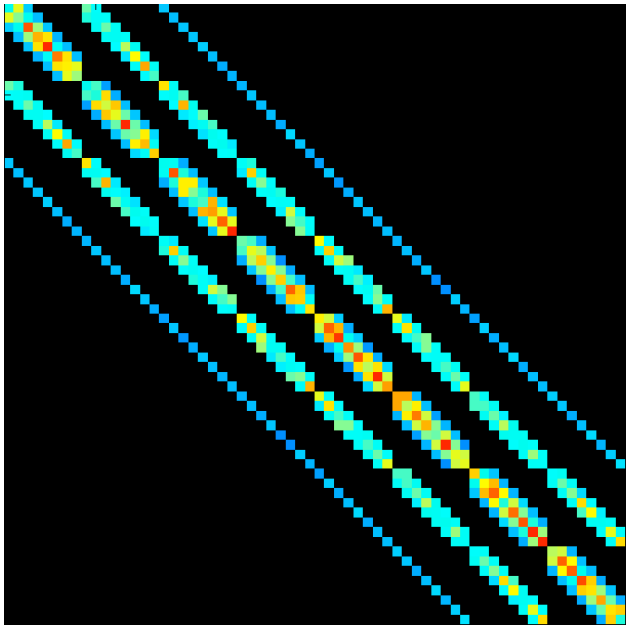

Figure 5.2: Example W matrix.

The performance of SpMV depends heavily on the structure of the matrix, as the arrangement of non-zeroes within each row determines the pattern of memory accesses. The matrices W arising from the normalized cut algorithm are all multiply-banded matrices, since they are derived from a stencil pattern where every pixel is related to a fixed set of neighboring pixels. Figure 5.2 shows the regular, banded structure of these matrices. The block pixels are all zero. It is important to note that the structure arises from the pixel-pixel affinities encoded in the W matrix, but the A matrix arising from the generalized eigen-problem retains the same structure. Our implementation exploits this structure in a way that applies to any stencil matrix. We store the non-zero diagonals of the matrix in consecutive arrays, following the diagonal format introduced by Bell and Garland [14]. In this diagonal format, we can statically determine the locations of non-zeroes. Thus, we need not explicitly store the row and column indices, as is traditionally done for general sparse

matrices. This algorithm nearly halves the size of the matrix data structure, and so doubles performance on nearly any platform.

The second bottleneck is reorthogonalization (line 9 in Figure 5.1). In perfect arithmetic, the basis V_j constructed by the Lanczos algorithm is orthogonal. In practice, however, finite floating-point precision destroys orthogonality in V_j as the iterations proceed. The most accurate Lanczos algorithm applies the full-orthogonalization strategy, which orthogonalizes the new Lanczos vector v_j against the latest basis V_{j-1} in every iteration. More work-efficient Lanczos algorithms preserve orthogonality by selectively reorthogonalizing new Lanczos vectors when the orthogonality property is worse than a predefined threshold. However, the computational complexity of orthogonalizing v_j against V_{j-1} is $O(jn)$. When the number of iterations grows, this operation can be more expensive than the SpMV operation. An alternative is to proceed without reorthogonalization, as proposed by Cullum and Willoughby [32]. We have found that this alternative offers significant advantages for normalized cut problems in image segmentation and image contour detection.

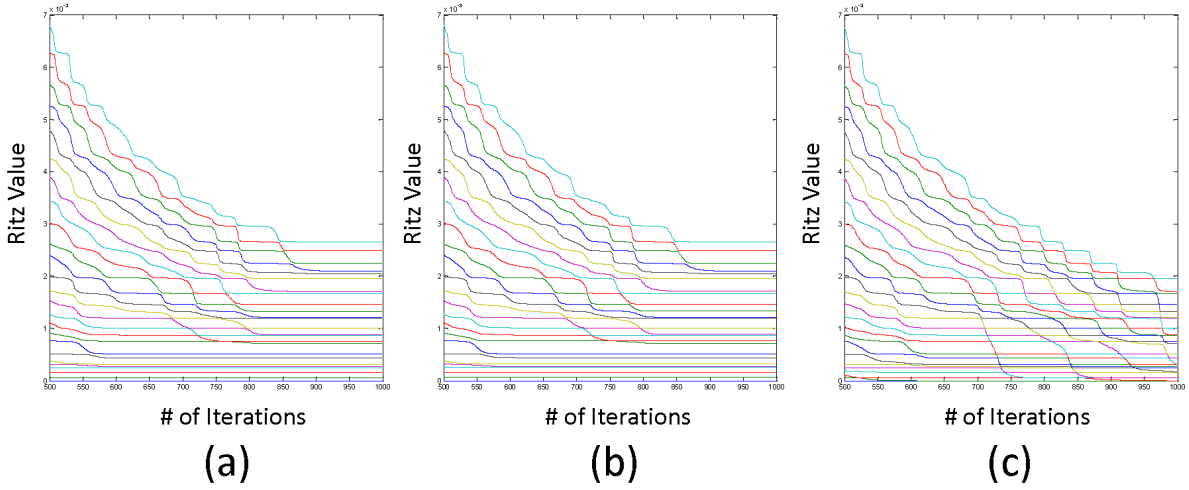

Figure 5.3: Convergence plot of the smallest 24 Ritz values from different strategies. (a) Full-reorthogonalization. (b) Selective-reorthogonalization. (c) No-reorthogonalization.

When V_j is not orthogonal, spurious and duplicate Ritz values will appear in Θ that must be identified and removed. This can be done by constructing $\hat{T}$ as the tridiagonal matrix by deleting the first row and first column of T_j. The spurious eigenvalues of T_j can then be identified by investigating the eigenvalues of $\hat{T}$. An eigenvalue is spurious if it exists in T_j only once and exists in $\hat{T}$ as well. For more details, see [32]. Because the lower eigenvalues of affinity matrices encountered from the normalized cut approach to image segmentation are well distributed, we can adopt the Cullum-Willoughby test to screen out spurious eigenvalues. Figure 5.3 shows the convergence plots from three different orthogonalization strategies. The x axis is the number of iterations; the y axis is the Ritz value. A Ritz value converges if the corresponding line stays flat and does not change. The plots track

the convergence paths of the smallest 24 Ritz values. The full-reorthogonalization, selective-reorthogonalization, and no-reorthogonalization strategies are employed in Figures 5.3(a), (b), and (c), respectively. As seen in the plots, all 24 Ritz values converge at around 850 iterations using full-reorthogonalization and selective-reorthogonalization strategies. Conversely, only 17 Ritz values converge at 1000 iterations using the no-reorthogonalization strategy. Although the no-reorthogonalization strategy requires significantly more iterations to converge, by getting rid of the expensive orthogonalization operations, this is still faster than the other two in practice. These results are discussed in Section 5.1.3.

This approach to reorthogonalization can be generally applied to all eigenvalue problems solved as part of the normalized cut method for image segmentation. In general, the eigenvalues corresponding to the different cuts (segmentations) are well spaced out at the low end of the eigen-spectrum. For the normalized Laplacian matrices with dimension N, the eigenvalues lie between 0 and N (loose upper bound) as $tr[A] = \sum_i \lambda_i = N$ and $\lambda_i \geq 0$. Since the number of eigenvalues is equal to the number of pixels in the image, one might think that as the number of pixels increases, the eigenvalues will be more tightly clustered, complicating convergence analysis using the Cullum-Willoughby test. However, we have observed that this clustering is not too severe for the smallest eigenvalues of matrices derived from natural images, which are the ones needed by the normalized cut algorithm. As justification for this phenomenon, we observe that very closely spaced eigenvalues at the smaller end of the eigen-spectrum would imply that several different segmentations with different numbers of segments are equally important. This is unlikely in natural images where the segmentation, for a small number of segments, is usually distinct from other segmentations. In practice, we have observed that this approach works very well for normalized cut image segmentation computations.

The third bottleneck is eigen-decomposition of the tridiagonal matrix T_j (line 11 in Figure 5.1). This can be solved by diagonalizing T_j infrequently, as it is necessary to do so only when checking for convergence, which does not need to occur at every iteration.

5.1.2 Exploring the Design Space of Parallelization Strategies

Based on our design decisions in the algorithm design space, we employ the Lanczos algorithm with no-reorthogonalization. As a result, every Lanczos iteration is composed of a SpMV operation with many vector updates. Our targeting hardware platforms are Nvidia GPUs, so we must explore the design space of parallelization strategies to better utilize the available resources.

For the vector updates, we employ the CUBLAS library [97] provided by Nvidia. For the SpMV operation, we store matrix A in the diagonal format introduced by Bell and Garland [14]. The diagonals of the matrix are stored in consecutive arrays, enabling high-bandwidth unit-stride accesses. We consider two different parallelization strategies in Figure 5.4. $A[j][i]$ refers to the ith element on the jth diagonal in matrix A. For the first strategy, we parallelize on the diagonals; for the second, we parallelize on the rows. The amount of parallelism expressed in the first strategy is m, while the amount expressed in the second strategy is n. n is orders of magnitudes larger than m. Every thread in the second strategy

Algorithm: Parallel SpMV Strategy 1
Input: A (Diagonal Matrix)
b (Vector)
n (Matrix Dimension)
m (Number of Diagonals)
$offset$ (Offsets of the Diagonals)
Output: c ($c \leftarrow Ab$)

```
1  parallel for i ← 1, ..., n
2      c[i] ← 0 ;
3  parallel for j ← 1, ..., m
4      for i ← 1, ..., n
5          cid ← i + offset[j] ;
6          c[i] ← c[i] + A[j][i] × b[cid] ;
```

Algorithm: Parallel SpMV Strategy 2
Input: A (Diagonal Matrix)
b (Vector)
n (Matrix Dimension)
m (Number of Diagonals)
$offset$ (Offsets of the Diagonals)
Output: c ($c \leftarrow Ab$)

```
1  parallel for i ← 1, ..., n
2      c[i] ← 0 ;
3      for j ← 1, ..., m
4          cid ← i + offset[j] ;
5          c[i] ← c[i] + A[j][i] × b[cid] ;
```

Figure 5.4: Two strategies of parallelizing the SpMV computation.

is independent, responsible for one element in array c. The threads in the first strategy require synchronization, because they might update the same element in array c. The second strategy accesses matrix A, vector b, and vector c in a unit-stride fashion. The first strategy accesses the matrix and the vectors in an irregular fashion. Based on these analyses, the second strategy is significantly better than the first.

For the affinity matrix composed from the normalized cut algorithm, we restrict each pixel to be related to neighboring pixels within a radius of 5. This assumption gives matrix A 81 diagonals, so the size of the offset array is 81 integers. For such a small array shared by every thread on the GPU, we can apply an additional optimization to store the offset array into the shared memory of the GPU. On Nvidia GPUs, a processor is composed of a set of Single Instruction Multiple Data (SIMD) processing units. The shared memory in a processor is available to all its SIMD processing units and has shorter latency compared to the global memory, which is shared by all processors on a GPU. The size of the shared memory is small, typically 16 KB to 48 KB, but it can certainly store 81 integers.

Based on these optimizations, we have achieved 40 GFLOPS for the SpMV computation on an Nvidia GTX 280 GPU.

5.1.3 Experimental Results

Here, we implement the parallel Lanczos algorithm in CUDA [96] and perform two experiments to demonstrate the effectiveness of our design space exploration. The input is a 154401×154401 sparse matrix with 81 diagonals composed from a 321×481 image following the normalized cut algorithm. We must compute the eigenvectors corresponding to the 9 smallest eigenvalues of the matrix. Choosing the no-reorthogonalization method with the

Cullum-Willoughby test in the Lanczos algorithm is an important step, so the first experiment compares the performance of our choice with two other common reorthogonalization strategies. The second experiment compares the parallel eigensolver with state-of-the-art serial and parallel eigensolvers on CPU platforms.

Eigensolver Reorthogonalization	Full	Selective	No (C-W)
Runtime (s)	15.83	3.60	0.78
Speedup	20.3 ×	4.62 ×	1 ×

Table 5.2: Execution times of different reorthogonalization strategies.

Table 5.2 shows the effect of various reorthogonalization strategies. These strategies are implemented in parallel using CUDA, and executed on an Nvidia GTX 280 platform. Full reorthogonalization ensures that every new Lanczos vector v_j is orthogonal to all previous vectors. Selective-reorthogonalization monitors the loss of orthogonality in the basis and performs a full-reorthogonalization only when the loss of orthogonality is numerically significant to within machine floating-point tolerance. The strategy we employ, as outlined earlier, is to forgo reorthogonalization, and use the Cullum-Willoughby test to remove spurious eigenvalues due to loss of orthogonality. As shown in Table 5.2, this approach provides a 20× gain in efficiency.

Eigensolver	MATLAB	TRLan [126]	Parallel MATLAB	No (C-W)
Runtime (s)	227	170	151.2	0.78
Speedup	291 ×	218 ×	194 ×	1 ×

Table 5.3: Execution times of different implementations.

To compare our parallel eigensolver with state-of-the-art eigensolvers, we use the MATLAB eigensolver, the TRLan package [126], and the parallel MATLAB eigensolver, and execute on an Intel Core i7 920 (2.66GHz) with 4 cores and 8 threads. The first two implementations are serial, and the third is parallel. Our implementation with the no-reorthogonalization strategy is executed on an Nvidia GTX 280 platform. The results are summarized in Table 5.3. By choosing a more efficient algorithm and parallelizing the algorithm on a massive parallel GPU platform, we achieve 291× speedup compared to a serial implementation, and 194× speedup compared to a parallel implementation. Conducting a detailed exploration of the design space has allowed our approach to outperform existing eigensolvers.

The eigen-decomposition pattern is the major bottleneck of the contour detection step in the object recognition system by Gu et al. [61]. The next step of the system is to segment images into regions based on the contours. The major bottleneck for the segmentation step is Breadth-First-Search (BFS) graph traversal on images. This pattern is introduced and optimized in the following section.

5.2 Breadth-First-Search Graph Traversal on Images

It is common to represent an image using a graph, with nodes representing image pixels and edges representing neighborhood relationships between pixels. So a pixel will only connect to its adjacent pixels, and the graph is highly structured. Graph algorithms are widely used for region and boundary related operations. For example, labeling of connected components can separate pixels into groups. Image color filling can mark pixels inside a region with the same color. Graph traversal algorithms can compute the city block distance transform, the chessboard distance transform, and local maximums of gray-scale images [22]. The watershed algorithm [90] can find image segmentations. More complicated graph algorithms are employed in state-of-the-art image segmentation techniques [48, 117, 132, 127, 7]. Breadth-First-Search (BFS) graph traversal on image graphs is a subset of the `Graph Traversal` application pattern in Table 5.1. Because it is frequently used in segmentation-based object recognition systems [61, 38], we want to parallelize and optimize this computation.

5.2.1 Exploring the Design Space of Algorithms

In general, queues are used for BFS graph traversal. First, the starting nodes from the graph are enqueued, and then each node is iteratively dequeued. The neighbors of the dequeued node are examined, and untraversed neighbors are enqueued. This procedure continues until the queue is empty. The basic idea of BFS graph traversal is to allow each dequeued node to actively access its neighboring nodes, and apply some operations on the neighboring nodes.

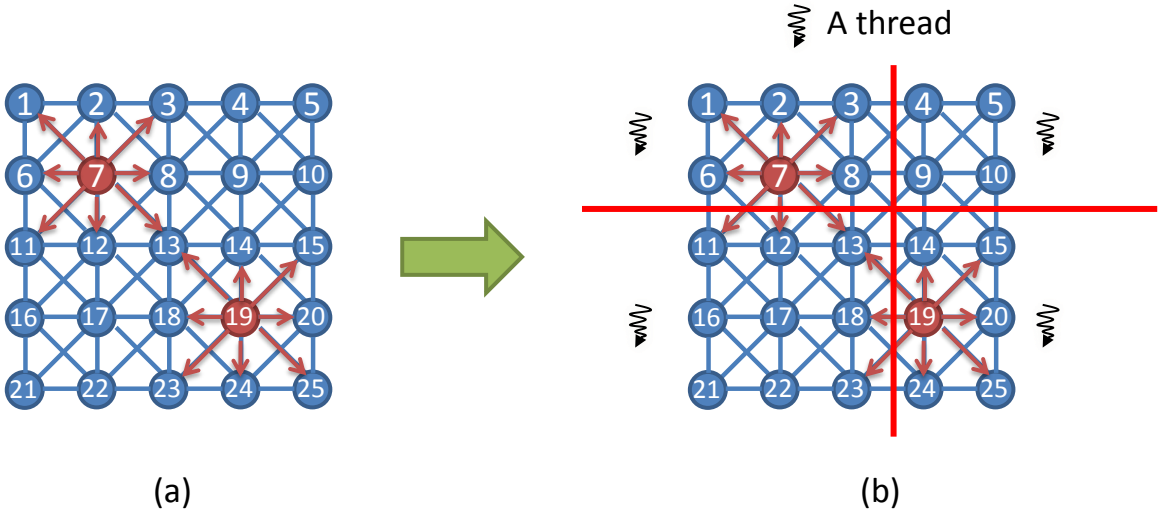

Figure 5.5: (a) A graph representation of an image. Nodes 7 and 19 are starting nodes, which will traverse their adjacent neighbors. (b) Parallel BFS graph traversal on a distributed graph.

Irregular data access poses a challenge when designing parallel BFS graph algorithms with good scalability. Existing parallel algorithms for graph traversal can be placed into two categories. The first uses a distributed graph representation and traverses this distributed graph in parallel. In this approach, starting nodes are identified for each distributed subgraph, and each subgraph is traversed in parallel. This algorithm is explained in Figure 5.5. In Figure 5.5(a), a 5×5 image is represented by a graph. Each node in the graph represents a pixel in the image, and each edge represents the neighboring relationship between pixels. We assume that the 8 nearest pixels are the neighbors of each pixel. Nodes 7 and 19 are two starting nodes. In the BFS graph traversal algorithm, nodes 7 and 19 will traverse their adjacent nodes. The distributed graph algorithm partitions the image graph into four subgraphs as shown in Figure 5.5(b). Since the structure of an image graph is highly regular, we only perform vertical and horizontal cuts. Based on this partition, the subgraphs can be traversed in parallel. The parallel BGL library uses distributed queues to represent the subgraphs, and operates on these distributed queues [57]. Scarpazza et al. optimized such a strategy on the Cell/BE processor [107], while Xia and Prasanna dynamically adjusted the number of working threads on the subgraphs to reduce synchronization overhead [128].

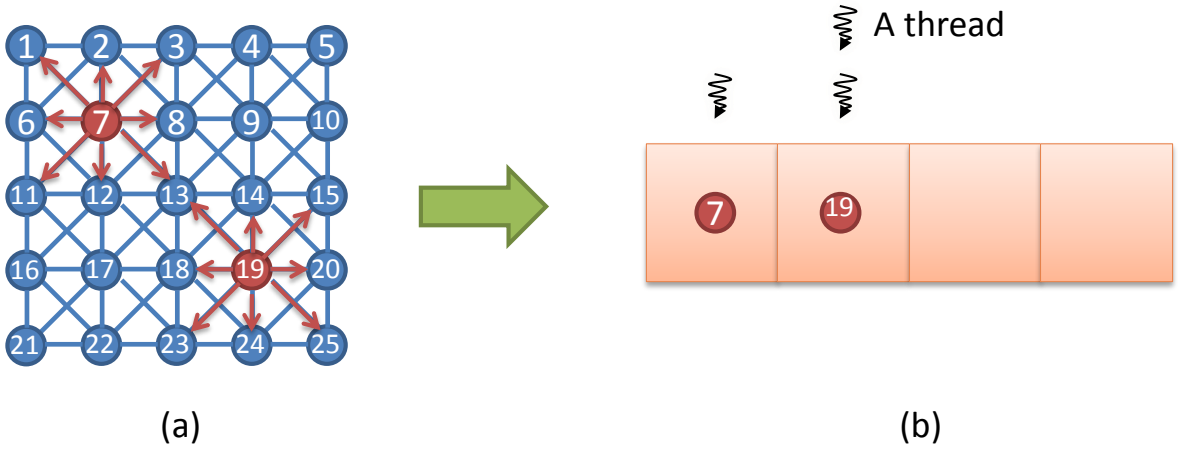

Figure 5.6: (a) A graph representation of an image. Nodes 7 and 19 are starting nodes, which will traverse their adjacent neighbors. (b) Parallel BFS graph traversal with a parallel task queue.

The second approach uses the concept of a parallel task queue to parallelize BFS graph traversal. This algorithm is explained in Figure 5.6. As shown in Figure 5.6(b), the starting nodes of the graph are collected in a queue, and then traversal tasks are distributed to available processing units in parallel. Bader and Madduri employed this approach on a Cray MTA-2 machine [11].

Here, we propose a novel approach for parallelizing the BFS graph traversal on images using highly-parallelizable structured grids computations.

Structured grids is a category of well-developed algorithms that are commonly used in the high performance computing (HPC) field [10, 15]. In structured grids computations,

data is arranged in a regular multi-dimensional grid, and each grid point is updated based on the states of the adjacent grid points. This update process continues until stop criteria are satisfied. Data arrangement is regular, and the data access pattern is regular and independent. The inherent parallelism in structured grids computation is considerable, and can be mapped to Single Instruction Multiple Data (SIMD) vector units. In addition, the order in which grid points are updated can be arranged to provide spatial locality, resulting in efficient use of the memory hierarchy. Additional optimization strategies are discussed in [35].

In mapping BFS graph traversal operations to structured grids, the computation is reversed: each graph node checks if it can be traversed by an adjacent node. This mapping is described in Figure 5.7. As shown in Figure 5.7(a), the naive BFS graph traversal algorithm makes nodes 7 and 19 traverse their adjacent nodes. In contrast, our own algorithm proposes mapping each node to a grid point, with edges mapped to the stencil operations on these grid points. As shown in Figure 5.7(b), the grid represents the spatial locations of graph nodes. Figure 5.7(c) illustrates the 8-point stencil operation that is applied on each grid point. In the proposed mapping, a stencil operation checks if a grid point can be traversed according to the configurations of its adjacent neighbors. For example, consider the corresponding grid point of node 13. The stencil operation checks the configuration of its 8 adjacent neighbors. Since grid points 7 and 19 are both traversed, grid point 13 can be traversed in the current iteration. However, now there are multiple sources that can traverse grid point 13. To enforce deterministic program behavior, we must specify whether grid point 13 should be traversed by grid point 7, grid point 19, or both, using a deterministic ordering. There are several ways to achieve this goal. If a deterministic serial implementation of the application is available, we can implement the behavior of the serial implementation in our parallel implementation. We can make decisions based on the nature of the application. We can also set up the traversal rule arbitrarily.

The algorithm for the proposed parallel structured grids computation for BFS graph traversal is summarized in Figure 5.8. Let i be a counter for the structured grids iterations; let $P_0, P_1, \ldots, P_i, \ldots$ be the traversal results after iteration i; let S be the set of starting nodes; let $R(p, q)$ be the rules specifying whether grid points p and q are connected; and let $N(p)$ be the set of neighbors of grid point p. Initially, P_0 is set to S (line 2). The status of each grid point is then iteratively updated by structured grids computations (lines 3-10 in Figure 5.8). For each iteration we distribute the computation of updating the status of each pixel to all available processing units (lines 6-8). Since the computation on each grid point is the same and also independent, we can evenly distribute the work. After each structured grids computation, if the status of all grid points remains the same then no additional traversal is required, and the computation is terminated (lines 9-10). On line 3 another termination criterion is used to control the runtime of the algorithm by ensuring that the total number of iterations will not exceed the maximum number allowed. The latest traversal results are returned as output (line 11). In order to efficiently check if the status of any grid points is changed in each iteration we employ a global variable *change* that is accessible by all processing units. Whenever a processing unit updates the status of a grid point, this global variable is set to *true*.

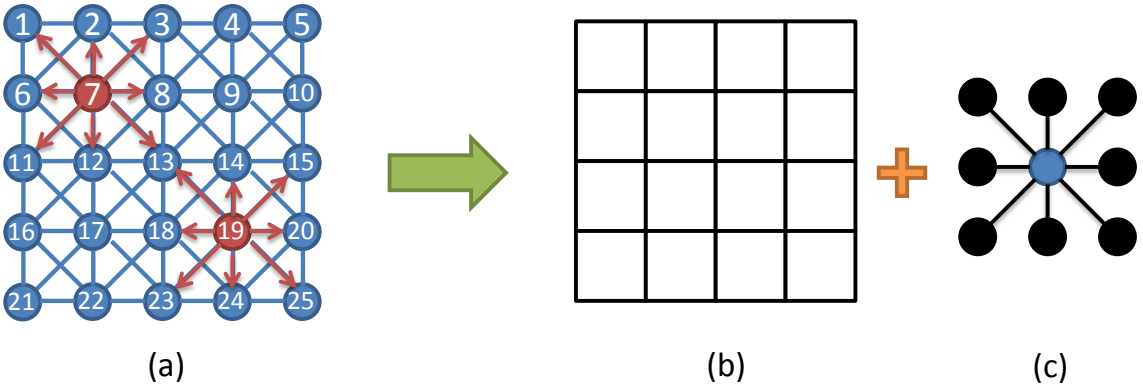

Figure 5.7: Mapping BFS graph traversal onto a structured grids computation. (a) A graph representation of an image. Nodes 7 and 19 are starting nodes, which will traverse their adjacent neighbors. (b) The structured grids representation of the graph. (c) The stencil operation on each grid point.

The Check_Update subroutine in Figure 5.9 is the update computation for each untraversed grid point. Suppose the subroutine is updating grid point p. It first verifies whether a neighbor of p in the previous stage P_{i-1} was enabled to traverse p (lines 1-5). The traversal is valid if neighboring pixels q and p satisfy the connection rule $R(q, p)$, and q has already been traversed (line 3). If multiple valid grid points surround a grid point, it is necessary to define an order of traversal to prevent nondeterministic updates (lines 6-8); however, if only one neighbor can traverse grid point p, then the computation is straightforward (lines 9-10). Finally, we check the status of grid point p, and set the global variable *changed* to *true* when the status of p is changed. Moreover, because there is no read operation on the global variable *changed* in this subroutine, all processing units can write the global variable asynchronously.

The basic idea of our proposal is to map the irregular BFS graph traversal computation to the regular structured grids computation. This mapping is feasible as long as the graph can be represented by structured grids. For example, instead of mapping a grid point to a pixel in the image, we can map a grid point to a set of pixels. This is equivalent to down-sampling the image. This proposed mapping strategy allows us to apply highly-parallel algorithms developed for structured grids to BFS graph traversal. For example, when implementing the proposed parallel BFS graph traversal algorithm on distributed systems, we can use ghost nodes or double buffering to deal with boundary grid points [10]. Adaptive mesh refinement algorithm [10] can also be used when the density of traversal can be defined in a coarse to fine grained fashion.

```
Algorithm: Structured Grid Traversal(I, S, R, N)
Input:   I (Input Image)
         S (Starting Nodes Set)
         R (Grid Connection Rules)
         N (Neighbor Definition)
Output:  P (Final image with traversed
             information stored in each grid point.)
1  i ← 0;
2  P_0 ← S;
3  while (i < Max_Iteration)
4      i ← i + 1;
5      changed ← false;
6      for each pixel p in I
7          if (Traversed (p) = false)
8              P_i ← Check_Update(p, I, P_{i-1}, R, N, changed);
9      if (changed = false)
10         break;
11 return P_i;
```

Figure 5.8: Parallel structured grids computation for BFS graph traversal.

Advantages and Disadvantages of the Proposed Parallel BFS Graph Traversal Algorithm

Comparing our proposed algorithm to existing parallel BFS graph traversal algorithms, we note three main advantages to our approach:

1. More extensive parallelism: In existing parallel BFS graph traversal algorithms, the traversal operation from different nodes can be done in parallel but the amount of parallelism is limited by the current working set of active nodes. In our proposed structured grids computation, all grid points are updated in parallel. In general, the number of nodes who can propagate information to its neighbors is much smaller than the number of grid points, resulting in more extensive parallelism.

2. Better scalability and load balancing: For graph traversal algorithms, the total computations necessary for traversal from different starting nodes are generally not known in advance. This makes it difficult to balance work loads for different processing units. In the distributed graph approach proposed in [57, 107, 128] it is difficult to define balanced graph partitions. The task queue approach in [11] works better for load balancing; however, task collection and redistribution are expensive. For structured grids, the computations on all grid points are the same. The workloads on available

```
Algorithm: Check_Update(p, I, P_{i-1}, R, N, changed)
1   valid_num ← 0;
2   for q ∈ N(p)
3       if (R(p, q) = true ∧ Traversed(q) = true)
4           valid(q) ← true;
5           valid_num ← valid_num + 1;
6   if (valid_num > 1)
7           order ← Decide_Trav_Order(valid, p, I, P_{i-1}, R, N);
8           Multiple_Grid_Trav(order, p, I, P_{i-1}, R, N)
9   if (valid_num = 1)
10          Single_Grid_Trav(valid, p, I, P_{i-1}, R, N);
11  if (Status(p) ≠ Status(P_{i-1}, p))
12      changed ← true;
```

Figure 5.9: The routine for updating information in each grid point.

processing units can be easily balanced, and the scalability is linear in the number of available processing units.

3. No race conditions: In general, if multiple nodes are enabled to traverse the same node, then the order in which updates are applied to the nodes is nondeterministic, resulting in a potential race condition. Our structured grids computation prevents this by ensuring that before a node updates itself it checks all adjacent neighbors that can traverse it, and then decides the access ordering according to the behavior of a serial implementation. This prevents any race condition from occurring.

A drawback of using the structured grids approach is that it leads to a larger number of computations than necessary – nodes that do not require an update also participate in computations. Potentially, this might consume more energy than other methods.

We now compare the computational complexity of the serial graph traversal algorithm, the existing parallel graph traversal algorithms, and the structured grids approach. Assuming that there are n starting nodes, where each starting node will finally traverse $a_1, a_2, \ldots, a_n$ nodes, respectively, the computational complexity of the serial algorithm is $O(\sum_{i=1}^{n} a_i)$. For existing parallel graph traversal algorithms with p processors, there is a difficulty in ensuring effective load balancing. So, we present them optimistically, assuming the following optimal load balancing scenario. Each processor takes care of n/p source nodes. The computational complexity of existing parallel graph traversal algorithms becomes $O(\frac{n}{p} \max_i(a_i))$, and the upper bound of the number of processing units is n. For structured grids computation, let the number of nodes in the graph be m, and the maximum distance that a starting node i can traverse be d_i. Then the computational complexity of the structured grids algorithm is $O(\frac{m}{p} \max_i(d_i))$, and p can scale to m. Suppose we have

m processing units, then the performance of parallel structured grids computation compared to the existing parallel graph traversal methods is $O(max(a_i))$, as opposed to $O(max(d_i))$. Usually $O(max(a_i)) = O((max(d_i))^2)$, so structured grids outperforms the existing parallel graph traversal methods when a large number of processing units is available.

5.2.2 Exploring the Design Space of Parallelization Strategies

We have presented three different parallel BFS graph traversal algorithms. The graph partition and task queue approaches are computationally efficient, but express less parallelism. The structured grids approach is computationally expensive, but expresses more parallelism. Therefore, the first two algorithms are better suited for CPU platforms, which are composed of a small number of powerful cores. However, the third algorithm is better suited for GPU platforms, which are composed of a large number of light-weight cores. We want to optimize these three algorithms on the targeting platforms. We compare their performance in Section 5.2.3.

Parallelizing the graph partition BFS graph traversal algorithm on a CPU platform is straightforward – we need only partition the graph into several subgraphs and use different threads to process them. Partitioning an image graph is also relatively simple, and involves specifying vertical cuts and horizontal cuts to evenly partition the image graph. The only challenge comes from each thread needing to communicate with the others when a traversal path crosses subgraph boundaries. For this, we employ a strategy similar to the distributed queue in the parallel BGL library [57]. Given n threads, each thread maintains n queues. Let q_{ij} be the jth queue maintained by the ith thread. q_{ii} contains the local traversal tasks for the ith thread. q_{ij} for $j \neq i$ contains tasks submitted from the jth thread. The jth thread submits a task to the ith thread if a traversal path from the jth subgraph goes to the ith subgraph.

Parallelizing the task queue BFS graph traversal algorithm on a CPU platform has three steps: task distribution, task processing, and task collection. The task distribution step evenly distributes tasks to all threads. The task processing step handles all tasks. A task is defined by a graph node – given a node, we need to access its neighbors, update its neighbors, and create new tasks associated with neighbors that have not been traversed. We let every thread have its own queue to store new tasks created by the thread. The task collection step merges all newly-created tasks from all threads into a single shared queue, which is used in the next iteration to evenly distribute tasks to all threads. This task collection step is the bottleneck of the algorithm. Because the threads do not communicate with each other in the task processing step, it is possible that one task is duplicate by many threads. To address this, two design decisions can be made here: we can sort the tasks and remove duplicates, or just leave the duplicate tasks. When distributing tasks, we always assign consecutive tasks to a thread. As a result, it is very rare for different threads to create duplicate tasks. Based on this observation, the first design strategy of removing the duplicate tasks is too expensive. So, we employ the second strategy to leave the duplicate tasks and process slightly more tasks.

While exploring the parallelization strategy design space of the previous two algorithms

Table 5.4: Comparison between serial algorithms and the proposed parallel algorithms.

MATLAB function	Core i7 (ms)	GTX 280 (ms)	Speedup
bwdist	532.8	252.2	2.11 ×
imregionalmin	452.4	79.3	5.70 ×
imfill	415.7	81.2	5.12 ×
watershed	2013	799.3	2.52 ×

on CPU platforms, we focus on coarse-grained parallelism. The overall computation is divided into several large pieces, and each piece is assigned to a thread. Conversely, while parallelizing the structured grids BFS graph traversal algorithm on GPU platforms, we need to find fine-grained parallelism. The finest granularity on an image graph is a node, so we use a GPU thread to check whether a node can be updated by its neighbors. The number of threads created is equal to the number of pixels in the image. However, because the operations on each node are the same, the SIMD units on the GPU can be efficiently used. Moreover, neighboring threads access neighboring nodes. To better take advantage of data locality, we use texture memory on the GPU to cache the image graph. This is the memory optimization we perform in our implementation.

5.2.3 Experimental Results

Speedup Against Serial BFS Graph Traversal Algorithms

We examine the effectiveness of our proposed parallel BFS graph traversal algorithm by implementing several image processing computations that use BFS graph traversal operations, and compare their runtime with serial implementations. In MATLAB, the *bwdist*, *imregionalmin*, *imfill*, and *watershed* functions all use BFS graph traversal operations, and the computational kernels of these functions are implemented by C++ mex functions. The mex functions are the interface to allow programmers to write C++ subroutines for MATLAB. Therefore, the performance of these functions is similar to highly optimized C++ implementations. We implement these functions using the proposed parallel BFS graph traversal algorithm, and compare the runtime with the serial MATLAB functions. We apply the algorithms on a 1600×1200 image. The serial implementation is executed on an Intel Core i7 920 (2.66 GHz) machine, while the parallel implementation is on an Nvidia GeForce GTX 280 card. The Nvidia card has 30 processing units, each of which has 8-way SIMD. Experimental results are summarized in Table 5.4: the parallel BFS graph traversal algorithm provides $2 - 6\times$ speedups. The variations result from differences in maximum traversal distances of different functions – this factor will be examined in the third experiment.

Scalability of the Proposed Algorithm

Scalability is an important measure of the effectiveness of a parallel algorithm. High scalability guarantees performance gain when the number of available processing units increases. To evaluate the scalability of our proposed parallel BFS graph traversal, we apply

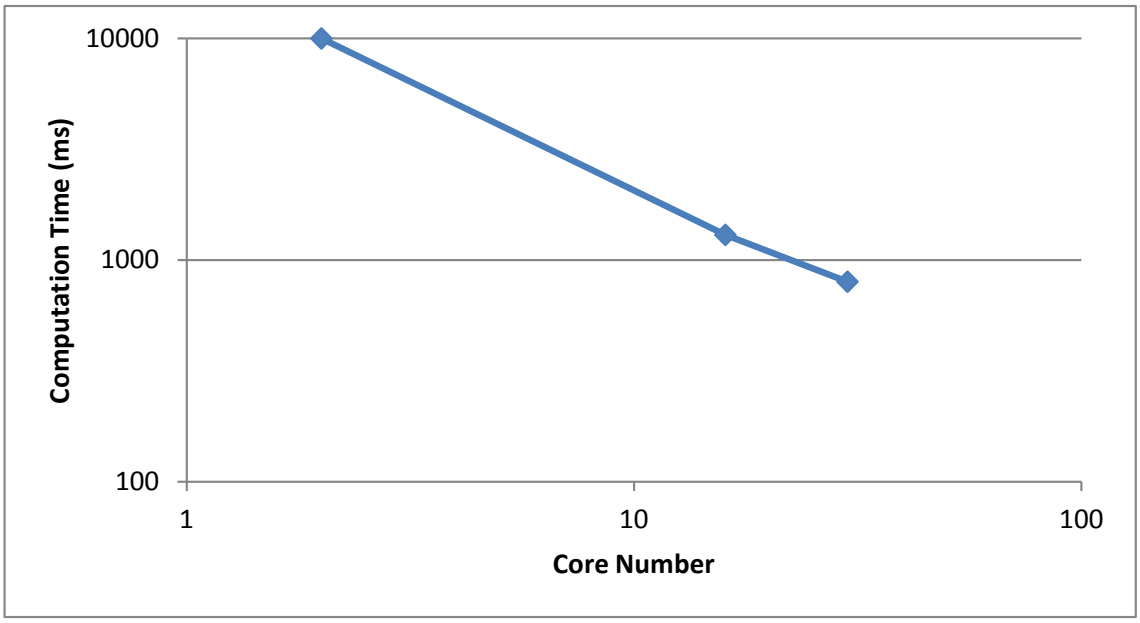

Figure 5.10: Scalability of the proposed parallel BFS graph traversal algorithm.

the watershed algorithm on the same 1600 × 1200 image, and execute on Nvidia GTX 280 (30 cores), 9800 GTX (16 cores), and 9400M G (2 cores). Results are shown in Figure 5.10: the proposed parallel algorithm scales linearly with the number of cores.

Effectiveness of the Proposed Algorithm on Different Images

As discussed in Section 5.2.1, the performance of the proposed parallel BFS graph traversal algorithm is proportional to the maximum distance that a starting node can traverse. In order to understand this property, we generate 7 images with different maximum traversal distances. The size of the images is 4096 by 4096, and the maximum traversal distance ranges from 4 pixels to 256 pixels. Using these seven images, we compare our proposed algorithm with the serial MATLAB *imregionalmin* function, the parallel graph partition algorithm, and the parallel task queue algorithm. The serial algorithm is executed on an Intel Core i7 920 (2.66 GHz) machine. We implement the graph partition algorithm and the task queue algorithm using OpenMP [99], and execute on an Intel Core i7 920 (2.66 GHz) machine with 4 cores and 8 threads. The parallel structured grids implementation is executed on an Nvidia GeForce GTX 280 card.

The experimental results are summarized in Figure 5.11. The graph partition algorithm and the task queue algorithm have similar performance, resulting in a 4× to 5× speedup against the serial algorithm. Also, the proposed structured grids implementation outperforms the other algorithms when the maximum traversal distance is less than 64 pixels. The runtime of the structured grids implementation increases when the maximum traversal

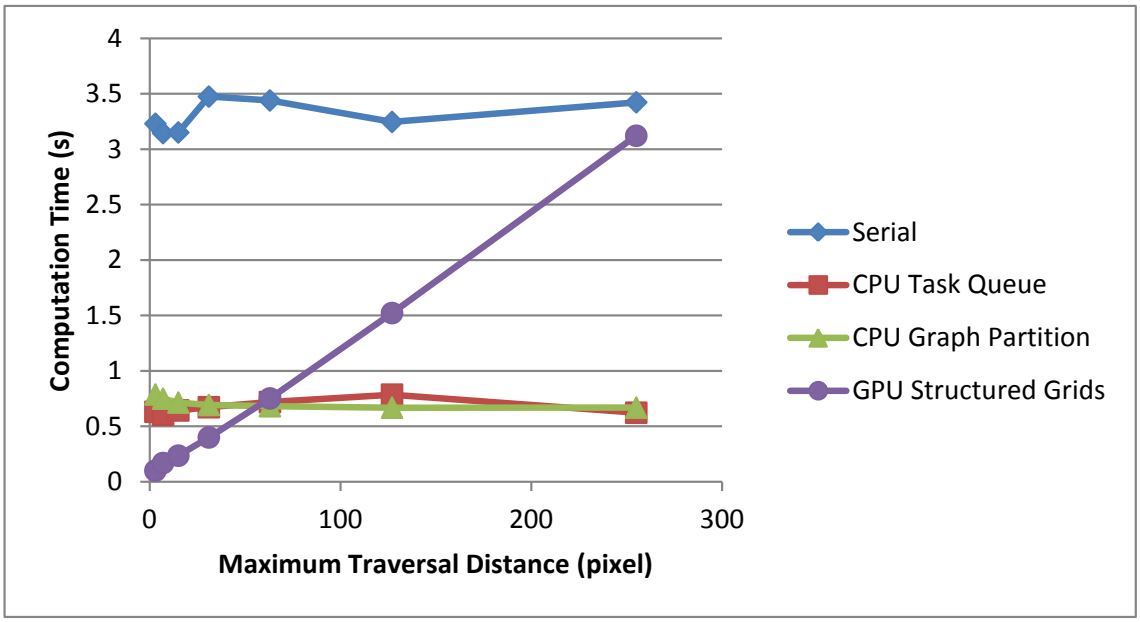

Figure 5.11: Runtime of the local minimum extraction algorithm on images with different maximum traversal distances.

distance increases. When the maximum traversal distance is only 4 pixels, the proposed parallel algorithm achieves 32× speedup against the serial implementation, and 6× speedup against the parallel graph partition and the parallel task queue implementation. Therefore, the proposed parallel BFS graph traversal is efficient when the maximum traversal distance is short. The maximum traversal distance is inversely proportional to the content complexity of an image. Therefore, the proposed parallel BFS graph traversal is most effective for complex images, which are very common. For example, images from Magnetic Resonance Imaging (MRI) are in this category. We have applied the watershed algorithm on hundreds of images in an MRI image database, ranging from 200×200 to 500×500 in size. The maximum traversal distance of these images is about 10-50 pixels. We have achieved speedup of 12× to 33× on structured grids versus serial, and 2× to 6× on structured grids versus graph partition and task queue.

The major bottleneck of the segmentation step in the object recognition system by Gu et al. [61] is the watershed algorithm, and can be solved by the BFS graph traversal pattern. Because the images in the benchmark suite are complicated, using the structured grids approach is very efficient. We will discuss the performance improvement in Chapter 7. After an image is segmented into regions, the next step is to collect contour features from regions. This pattern is discussed in the following section.

5.3 The Contour Histogram

`Histogram Accumulation` is a key application pattern in Table 5.1. This is used in almost all feature descriptors, such as SIFT [80], HOG [33], GIST [98], and contour feature [61]. A histogram accumulation computation can be defined by a vector, a histogram, and two functions. Let a be a vector of n elements $a = [a_1, a_2, \ldots, a_n]$, and h be a histogram of m bins $h = [h_1, h_2, \ldots, h_m]$. For every element a_i in a, we must find a histogram bin it can contribute to. The index of the corresponding histogram bin is related to the value of a_i, and to the index of a_i, which is i. Let $f(a_i, i)$ be the function that computes the corresponding histogram bin index based on a_i and i. If a_i contributes to histogram bin h_j, or $f(a_i, i) = j$, we must then compute the amount that a_i will contribute to histogram bin h_j. Let $v(a_i, i)$ be the function that computes the amount of contribution. The histogram accumulation computation can be described by $\sum_{i=1}^{n} h[f(a_i, i)] \leftarrow h[f(a_i, i)] + v(a_i, i)$.

Although the histogram accumulation computation can be simply defined by two arrays and two functions, the two functions can vary significantly. Different feature descriptors define different f and v functions. Because function f decides memory access behavior, optimizing the histogram accumulation computation is tightly related to function f. It is very difficult to design a library that can handle all kinds of f functions efficiently. As a result, different feature descriptors with different f functions should be optimized differently. Our proposed parallel library for the histogram accumulation application pattern should be a combination of all these special cases. We start this task by optimizing the contour feature in the region-based object recognition system by Gu et al. [61].

The contour descriptor is defined by encoding the gPb feature [81] of a region into a 128-bin histogram. The region will be evenly divided into 4×4 grids, and each grid contributes to 8 histogram bins. So, there are $4 \times 4 \times 8 = 128$ bins in total. For each grid, the gPb features of 8 orientations are accumulated in 8 different bins. In other words, the contour location information is discretized into a 4×4 coordinate system. The contour orientation information is discretized into 8 orientations. The contour descriptor of a region is shown in Figure 5.12. The leftmost column is a region with its gPb features. Given a pixel p inside the region, we evaluate its orientation and location. If the orientation is o, and the location falls in grid (i, j), then the gPb value of the pixel will be accumulated into histogram bin number $(o-1) \times 16 + (i-1) \times 4 + j$.

5.3.1 Exploring the Design Space of Algorithms

We have tried two algorithms to describe the contour histogram computation. The first algorithm is based on processing pixels within the bounding box of each region. This algorithm is summarized in Figure 5.13. Let R be the region set with n regions extracted from the input image, and we compute the contour histogram for each region. H is the corresponding histogram set for all regions. The size of each histogram is 128, so the dimension of H is $n \times 128$. In the algorithm, we first initialize H, and then accumulate gPb features into the histogram bins. For each region, we must accumulate the gPb features of all pixels inside the region into corresponding histogram bins. The Bounding_Box($R[i]$)

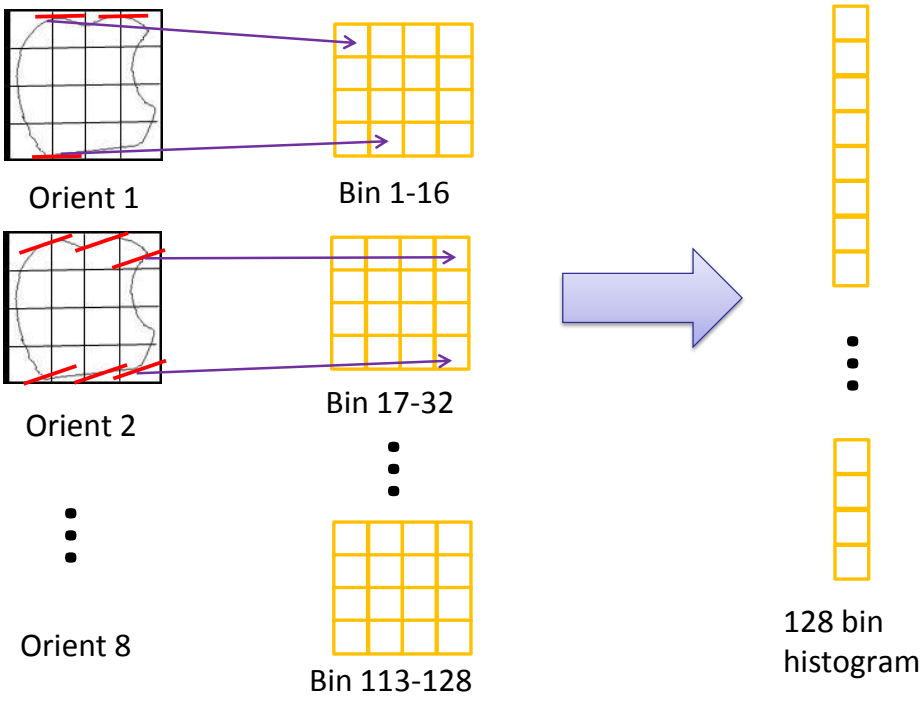

Figure 5.12: The contour feature used by Gu et al. [61].

function computes the bounding box of region $R[i]$. If a pixel p is inside the bounding box, and belongs to region $R[i]$, we employ the Orient(p) and Grid_Coor(p) functions to compute the orientation and grid coordinates of the pixel. This information is then used to compute the index of histogram bin *binid*. Finally, the gPb feature of orientation o and pixel p is accumulated into the histogram bin of index *binid*.

The second algorithm takes advantage of the fact that a region is divided into 4×4 grids and that the pixels within a grid contribute to the same histogram bin. This algorithm is summarized in Figure 5.14. The pixel-based algorithm uses two nested loops to compute the histogram, while the grid-based algorithm uses three nested loops. Given a region, this algorithm processes pixels within each grid separately. The 4×4 grid partition of the region is stored in array *grid*. $grid[4(x-1)+y]$ refers to the grid with coordinates (x, y). Given a grid, we process pixels within the grid and accumulate the gPb feature of the pixel to the corresponding histogram bin.

The grid-based algorithm seems to be more complicated than the pixel-based algorithm; however, the grid-based algorithm's memory behavior is better. When processing pixels within a grid, we must access only 8 histogram bins. We can use registers to store the contents of the 8 histogram bins, which gives us the shortest latency to update the bins. Conversely, the pixel-based algorithm must access all 128 histogram bins. This is unlikely to keep them all in registers, so we need to use lower-layer memory such as the L1 cache to store the histogram contents. This introduces longer delays when updating the contents.

```
Algorithm: Pixel-Based Contour Histogram
Input:   R (Region Set)
         n (Region Set Size)
         gPb (gPb Feature)
Output:  H (Histogram Set)
1   for i ← 1, ..., n
2       for j ← 1, ..., 128
3           H[i][j] ← 0;
4   for i ← 1, ..., n
5       for pixel p in Bounding_Box(R[i])
6           if (p ∈ R[i])
7               o ← Orient(p);
8               (x, y) ← Grid_Coor(p);
9               binid ← 16(o − 1) + 4(x − 1) + y;
10              H[i][binid] ← H[i][binid] + gPb[o][p];
```

Figure 5.13: The pixel-based contour histogram algorithm.

5.3.2 Exploring the Design Space of Parallelization Strategies

Two parallelization strategies can be applied on the pixel-based contour histogram algorithm. The first strategy processes each region in parallel. The histogram computation on each region is totally independent, so no communication is required. The number of regions in an image is usually in the range of 200 to 500 – not too large. CPU platforms are better suited for this kind of coarse-grained parallelism. The second strategy expresses parallelism on both regions and pixels. We can process each region in parallel. Moreover, when computing the histogram of a region, we can access pixels within the region bounding box in parallel. GPU platforms are the ideal candidates for this strategy, because they are good at expressing fine-grained parallelism and at hosting many threads concurrently. We distribute the regions to the GPU processors, using the SIMD unit of each to process consecutive pixels in parallel. To be more specific, a set of threads is created, and consecutive threads are responsible for consecutive pixels. Because operations on the pixels are the same, these threads can be executed concurrently on the SIMD pipeline. The data access pattern is vectorized because consecutive threads access consecutive pixels. However, different threads might update the content of the same histogram bin at the same time. In order to ensure the correctness of the results, we must use atomic operations to commit the update. An atomic operation is a sequence of instructions that are executed sequentially without interruption. Such operations are more expensive than regular instructions. Moreover, when collision occurs, the operations will be serialized.

For the grid-based contour histogram algorithm, we have also tried two parallelization

```
Algorithm: Grid-Based Contour Histogram
Input:   R (Region Set)
         n (Region Set Size)
         gPb (gPb Feature)
Output: H (Histogram Set)
1   for i ← 1, ..., n
2       for j ← 1, ..., 128
3           H[i][j] ← 0;
4   for i ← 1, ..., n
5       for j ← 1, ..., 16
6           for pixel p in grid[j]
7               if (p ∈ R[i])
8                   o ← Orient(p);
9                   binid ← 16(o − 1) + j;
10                  H[i][binid] ← H[i][binid] + gPb[o][p];
```

Figure 5.14: The grid-based contour histogram algorithm.

strategies. The first is based on the `Geometric Decomposition` pattern in OPL as shown in Figure 4.1. The idea of this pattern is to divide the input data into regular partitions using geometric information and then deal with each partition in parallel. In the grid-based contour histogram algorithm, we partition a region into 4×4 grids by employing three evenly-distributed vertical cuts and three evenly-distributed horizontal cuts. This is an exact match with the `Geometric Decomposition` pattern. We use GPU platforms to express this strategy. The regions are distributed to GPU processors. In a processor, 128 threads are generated for a region, where each thread walks through pixels in one orientation in a grid, and accumulates the gPb values in the corresponding histogram bin. Although the threads are grouped and executed using the SIMD unit, the data access pattern is not vectorized, which may incur significant memory access overhead. The second strategy focuses on one grid at a time. The regions are still distributed to GPU processors. However, in a processor, we serially process the 16 grids. The SIMD unit is used to perform parallel reduction over pixels within a grid. The data access pattern is vectorized. However, due to the nature of the parallel reduction, not all threads are working all the time.

In summary, each algorithm and parallelization strategy has its advantages and disadvantages. The parallel region strategy on the pixel-based algorithm explores coarse-grained parallelism on CPU platforms. The atomic operation strategy on the pixel-based algorithm explores fine-grained parallelism on GPU platforms, and has vectorized memory access, but uses more expensive atomic operations. The geometric decomposition strategy on the grid-based algorithm explores fine-grained parallelism on GPU platforms, but does not have vectorized memory access. The reduction strategy on the grid-based algorithm explores

fine-grained parallelism on GPU platforms, and has vectorized memory access, but the utilization rate of the threads is lower.

5.3.3 Experimental Results

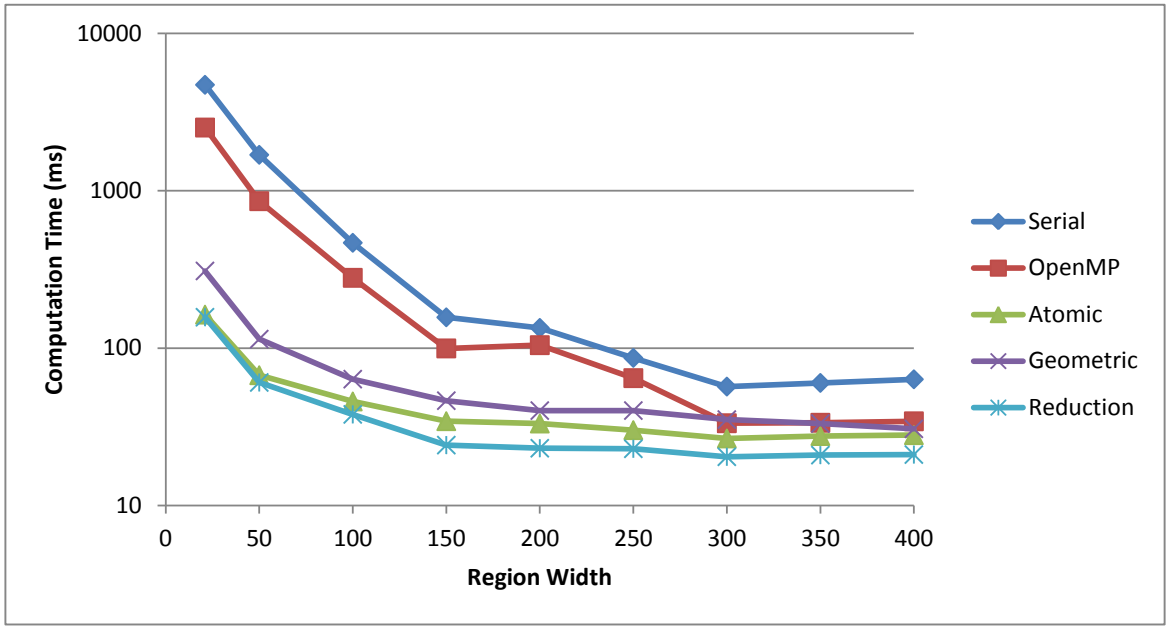

Figure 5.15: Execution time of different algorithms and parallelization strategies on images with various region sizes.

In our experiments, we implement the serial histogram computation and the parallel region strategy on an Intel Core i7 920 (2.66 GHz) machine, using OpenMP [99] with 8 threads, and we implement the atomic operation, geometric decomposition, and reduction strategies on an Nvidia GTX 480 machine, using CUDA [96]. We evaluate the performance of these strategies on a 512×410 image. An image can have regions with different sizes. Larger regions contain more pixels, while smaller regions contain fewer pixels. In order to understand whether region sizes will influence the performance of different parallelization strategies, we partition the benchmarking image into square regions with different sizes, and evaluate the performance of all implementations on regions with different sizes. Experimental results are summarized in Figure 5.15: the reduction strategy outperforms all other strategies, irrespective of the size of the regions in the image. We conclude that the non-vectorized data access and thread conflict problems are more serious than the thread stalling problem. Therefore, the reduction strategy is the best solution for the contour histogram problem – it achieves $5 - 30\times$ speedup compared to the serial contour histogram implementation.

5.4 Summary

In this chapter, we describe three case studies of application patterns that are the major bottlenecks of the object recognition system by Gu et al. [61]. These include eigensolver for the normalized cut algorithm, BFS graph traversal on images, and contour histogram. We apply the exhaustive search approach to explore the design space of algorithms and parallelization strategies on these computations, and make our design decisions based on experimental results.

We select the Lanczos algorithm to solve the eigen-problem from the normalized cut algorithm. The three major bottlenecks for the Lanczos algorithm are then examined and resolved. The SpMV computation is optimized by using the diagonal format to store the sparse matrix and parallelize on the matrix rows. Reorthogonalization overhead is reduced by using the no-reorthogonalization strategy with the Cullum-Willoughby test. The cost of computing the Ritz values is reduced by doing this infrequently. According to the experimental results, 20× speedup is achieved by reducing the reorthogonalization cost. Compared to a serial implementation, 280× speedup is achieved by applying all optimizations.

For the BFS graph traversal algorithm, we propose a new algorithm to express massive parallelism on image graphs. The idea is to map the original graph traversal algorithm to a structured grids algorithm. Instead of making a pixel propagate information to its neighbors, we ask all pixels to collect information from neighbors. This new algorithm achieves $2-5\times$ speedups compared to serial implementations on various computations that rely on BFS graph traversal. The new algorithm scales linearly on the number of cores, and it outperforms other parallel BFS graph traversal algorithms when the maximum traversal distance is less than 64 pixels. On complicated images, the new algorithm achieves $12-33\times$ speedups versus the serial algorithm, and $2-6\times$ speedups versus other parallel algorithms.

The contour histogram computation can be solved by two algorithms, and each can be parallelized by two different strategies. Every strategy has its advantages and disadvantages. We have evaluated the performance of all four strategies and concluded that the reduction strategy is the best solution for this problem, achieving $5-30\times$ speedups compared to the serial algorithm.

These three case studies illustrate how we can apply the exhaustive search method to explore design spaces, and so achieve significant speedups by parallelizing and optimizing a serial computation. This is a subset of the application patterns for our proposed parallel application library for object recognition. We can also continue performing case studies on the application patterns to enlarge the functionality of the library. The ultimate goal is to cover all application patterns listed in Table 5.1. This is our future work and will be further discussed in Section 8.2.

Chapter 6

The OpenCL for OpenCV (OLOV) Library

The micro-architectures of parallel microprocessors are increasing in their diversity. As a result, different hardware vendors have developed their own languages to exploit parallelism in their architectures, such as SSE [115] and AVX [70] for x86, CUDA [96] for Nvidia GPUs, and Stream [2] for AMD GPUs. Fortunately, the leaders of the parallel computing industry have standardized parallel computations with OpenCL [116]. The goal of OpenCL is to make parallel code portable to heterogeneous platforms. OpenCL is an open-source project with partners from industry and academia, including AMD, Apple, ARM, Blizzard, Broadcom, Codeplay, Creative, Electronic Arts, Ericsson, Fixstars, Freescale, Fujitsu, GE, Graphic Remedy, HI, IBM, Imagination Technologies, Intel, Kestrel Institute, Khronos, Alamos National Laboratory, Media Tek, Motorola, Movidia, Nokia, Nvidia, Petapath, QNX, Qualcomm, RapidMind, Renesas, S3 Graphics, Samsung, Seaweed Systems, Sony, ST Microelectronics, Symbian, Takumi, Texas Instruments, Toshiba, Vivante, and ZiiLABS. This long list of membership includes designers from various fields and platforms: GPU, CPU, SoC, mobile device, super computer, video game, and software. With these partnerships, we can expect OpenCL to have extremely wide coverage on all kinds of hardware platforms. If there is ever any standard programming model that rules how programmers write parallel programs, very likely it will be OpenCL.

In all case studies presented in Chapter 5, we optimize the computation on Nvidia GPUs using CUDA – if a user does not have an Nvidia GPU, he cannot use our code. Although portability is not our topmost concern when developing this parallel application library, we still want to provide a solution that can be executed on platforms from more than one vendor. Since OpenCL is the best currently-available solution for portability, we use it as the main parallel programming model for developing our library.

OpenCV [21] is the most well-known open-source computer vision library, and is widely used in many projects. It aims to provide functions for real-time computer vision applications. This goal matches our purpose of speeding up object recognition systems. When the library we develop reaches a mature stage, we plan to provide APIs that are synchronous with OpenCL's APIs, so that people can use them easily. As a result, we call our project of

developing a portable parallel application library the OpenCL for OpenCV (OLOV) project.

6.1 Overview of OLOV

In this section, we discuss our target platforms, and the scope of the OLOV project.

Many different parallel platforms exist today, such as Intel Nehalem, AMD Phenom, IBM Cell, Sun Niagara, Nvidia Fermi, and AMD Radeon. However, not all parallel platforms are commonly available. Generally, consumers purchase a desktop or laptop with a CPU from Intel or AMD, and a GPU from Nvidia or AMD. We are targeting platforms commonly employed by end users, and not specific to research environments, which narrows our interest to multicore CPU platforms and manycore GPU platforms.

By multicore CPU, we refer to architectures that duplicate several traditional CPUs on a chip. Each core has its own memory hierarchy with full instruction support, and operates with a high clock frequency. A manycore GPU, on the other hand, incorporates numerous small cores on a chip. Each GPU core has widely-vectorized algorithmic logic units, and can simultaneously execute many threads. However, the individual cores may not have full instruction support, and operate on a lower clock frequency compared to CPUs. From an architectural perspective, we present the advantages and disadvantages of these two mainstream parallel platforms as follows:

- Task Parallelism vs. Data Parallelism: Multicore CPUs are well suited for expressing coarse-grained task parallelism, while manycore GPUs are better for expressing fine-grained data parallelism. With multicore CPUs, each core can independently operate on different tasks, but there are only a relatively small number of available cores. Manycore GPUs, on the other hand, have a larger number of cores, each with a wide SIMD width. As the capabilities of CPUs and GPUs evolve, we can express data parallelism using the SIMD units in multicore CPUs. Similarly, we can express task parallelism using the scheduler in manycore GPU runtimes. However, the amount of available data parallelism in CPUs is still less than in GPUs, and compared to CPUs the task parallelism in GPUs remains restrictive.

- Irregular Data Access vs. Regular Data Access: Multicore CPUs effectively handle both regular data access and irregular data access, while regular data access is necessary to achieve high performance on manycore GPUs. With multicore CPUs, regular data access patterns do result in better cache usage, but the latency resulting from irregular data access is reduced by the memory hierarchy. GPUs, however, rely heavily on data parallelism – in the absences of regular data access, the memory bandwidth for GPU architectures may be significantly reduced, resulting in potential serialization of kernels.

As such, CPU and GPU platforms have different characteristics. When optimizing a computation on CPU platforms, we normally use coarse-grained parallelization strategies – we partition a computation into several expensive tasks, and assign independent tasks to each CPU core. On the other hand, when optimizing a computation on GPU platforms we

commonly use fine-grained parallelization strategies – that is, expressing data parallelism on consecutive data to utilize the wide SIMD unit on GPU cores. Although OpenCL is executable across all CPU and GPU platforms, performance is not portable. Code optimized for CPU platforms performs worse on GPU platforms, and vice versa. There is no one-size-fits-all solution. Because optimizing a computation on one platform is challenging enough, we focus our research on only one platform. We do plan to expand the OLOV project to support more platforms in the future, however.

Comparing CPU and GPU platforms, GPUs provide larger memory bandwidth on regular memory accesses. For example, the memory bandwidth of an Nvidia GTX 480 is 177 GB/s, and the memory bandwidth of an AMD Radeon 6970 is 176 GB/s. CPU memory bandwidth, however, is always bounded by DRAM, and is therefore significantly lower. The memory bandwidth of an Intel Core i7 980 is only 25.6 GB/s. Moreover, because GPU platforms have wider SIMD units, the peak floating point number performance is also higher. The GTX 480 and Radeon 6970 can perform 1345 GFLOPS and 2700 GFLOPS, respectively, while the Core i7 980 can perform only 160 GFLOPS. As a result, as long as a computation accesses memory regularly and performs the same operations on many data, we can expect GPU platforms to deliver better performance. Most of the application patterns in Table 3.1 satisfy these properties. As such, we target GPU platforms in the OLOV project.

However, even if we target only GPU platforms, the capabilities of each individual platform are nonetheless different. Nvidia GPUs are composed of a cluster of processors, and each processor has schedulers that distribute instructions to its SIMD units. Different GPUs have different numbers of cores, different SIMD widths, different numbers of schedulers, different numbers of registers, different sizes of local memory and caches, and different global memory bandwidths. AMD GPUs are also composed of a cluster of processors; however, each processor is composed of many stream cores, and each stream core is itself composed of a Very Long Instruction Word (VLIW) unit. Different AMD GPUs have different numbers of processors, different numbers of stream cores per processor, different VLIW widths, different numbers of registers, different sizes of local memory and caches, and different global memory bandwidths. Because of all these architectural differences, we need different optimization strategies on each individual platform. Given a computation, the superset of all valid implementations forms the design space, and we must explore the design space to find the best implementation on a specific platform. It is impossible to ask all library users to explore the design space themselves. Therefore, we need to develop autotuners that automatically explore the design space and optimize a computation on a specific platform when a user installs our library.

Although we can develop autotuners for every application pattern in Table 3.1, the research time for developing an autotuner is significantly longer than that of the case studies discussed in Chapter 5. As a result, we focus on computations that are frequently used and computationally intensive. We have developed two autotuners for two computations that satisfy these requirements. The first is for sparse matrix vector multiplication, which is the major bottleneck in `Histogram Accumulation`, `Quadratic Optimization`, `Eigen Decomposition`, `Nonlinear Optimization`, `Singular Value Decomposition`, and `Convex Optimization` application patterns. The second autotuner is for the `Vector Distance`

pattern. The details of these autotuners are discussed in Sections 6.3 and 6.4.

In summary, the OLOV project develops a collection of autotuners for computationally-intensive application patterns on GPU platforms using OpenCL.

6.2 OpenCL Programming Model

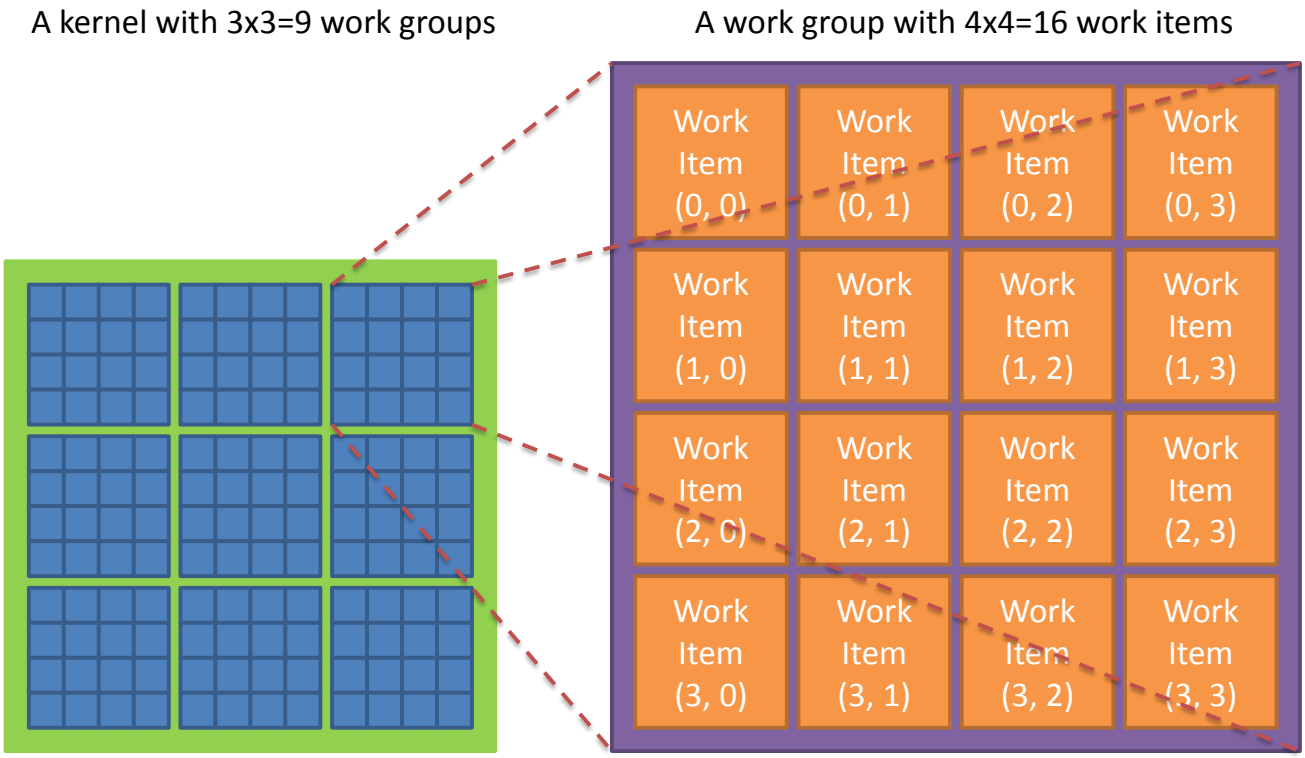

Figure 6.1: The programming model of expressing data parallelism in OpenCL.

The OpenCL programming model is a combination of task and data parallelism. A task queue is created to store OpenCL kernels – if these kernels are independent and are launched asynchronously, they can be executed in parallel. This is the preferred way to express task parallelism on CPU platforms.

Data parallelism is expressed within an OpenCL kernel, and is described by Figure 6.1. A programmer organizes a computation by work groups, each of which is independent and cannot communicate with others. A work group is composed of many work items. These work items work together and share the register file and local memory belonging to the work group. The work items in a work group can be synchronized using shared local memory. Both work groups and work items can be indexed in one-dimensional, two-dimensional, or three-dimensional fashion. For example, in Figure 6.1, we create two-dimensional $3 \times 3 = 9$ work groups, and each work group has two-dimensional $4 \times 4 = 16$ work items. Data

parallelism is expressed by work items in a work group performing the same instructions on different data. We follow this programming model to express data parallelism on GPU platforms.

Although the OpenCL programming model remains the same for all GPU platforms, the avenues through which the programming model maps to real hardware architectures are still different. On Nvidia GPUs, each work group is scheduled to one processor, and a warp of work items is scheduled together on the SIMD units in a processor. The warp size of current Nvidia GPU architecture is 32, so 32 work items are gathered and executed on the SIMD units of a processor. On AMD GPUs, each work group is also scheduled to one processor, and a wavefront of work items is scheduled together on the stream cores. The wavefront size of current AMD GPU architectures is 64, so 64 work items are gathered and executed on each stream core. A stream core is composed of a VLIW unit with a width of four or five. In order to effectively use these VLIW units, we must employ explicit vector types, such as *float4* or *int4*, to advise the compiler to generate width-four VLIW instructions for each work item. As a result, when writing OpenCL programs, we can use scalar data types (such as *float* or *int*) on Nvidia GPUs, but should use vector types on AMD GPUs. The architectural differences between Nvidia and AMD GPUs are also considered in the OLOV project.

6.3 The Sparse Matrix Vector Multiplication Autotuner

Many of the optimization and linear algebra application patterns in Table 3.1 can be solved by iterative methods, composed of matrix vector multiplication operations with vector updates. For example, the conjugate gradient method can be used to solve linear systems, and Krylov subspace methods can be employed to find eigenvalues and eigenvectors [131]. Further, many matrices are naturally sparse, so we must perform sparse matrix vector multiplication (SpMV) operations in the optimization and linear algebra application patterns. Since the matrix size is orders of magnitude larger than the vector, SpMV operations dominate the execution time of iterative methods. In order to accelerate these iterative methods it is essential to optimize the SpMV kernel.

The SpMV kernel is notorious for its extremely low arithmetic intensity (the upper bound of the flop:byte ratio is 0.25 – two flops for eight bytes on the single precision floating point data type), and for its irregular memory access patterns [123]. The SpMV kernel is a pure memory bounded problem as shown in the Roofline model [124]. Although the peak floating point operations per second (FLOPS) of modern microprocessors is increasing rapidly, the maximum memory bandwidth is not improving at a similar pace. As a result, the SpMV kernel usually performs poorly, achieving only 10% of peak performance on single-core, cache-based microprocessors [120]. Studies to improve performance of the SpMV kernel can be categorized into two directions: applying architecture-specific optimizations, and applying new sparse matrix formats.

Interest in SpMV has increased with the advent of more powerful multicore CPUs and

manycore GPUs. Williams et al. [123] evaluate different optimization strategies on AMD Opteron X2, Intel Clovertown, Sun Niagara2, and STI Cell SPE. Bell and Garland [14] optimize different SpMV kernels with different sparse matrix formats on Nvidia GPUs. Bordawekar and Baskaran [19] further optimize the SpMV kernel with the Compressed Sparse Row (CSR) sparse matrix format on Nvidia GPUs. Choi et al. [27] implement Blocked Compress Sparse Row (BCSR) and Sliced Blocked ELLPACK (SBELL) formats on Nvidia GPUs.

Researchers have also proposed various sparse matrix formats with the goal of minimizing the memory footprint and enforcing some regularity on the access pattern. Buluc et al. [23] employ the symmetric Compressed Sparse Block (CSB) and bitmasked register block data structures to minimize the storage requirement of blocked sparse matrices. Monakov et al. [92] propose the Sliced ELLPACK (SELL) format as an intermediate format between the CSR and ELL formats. Vázquez et al. [118] suggest the ELLPACK-R format, which can preserve the data alignment requirement on Nvidia GPUs.

Different sparse matrices have different characteristics, and different microprocessors have different strengths. So, in order to achieve the best SpMV performance for a specific sparse matrix on a specific microprocessor, an autotuner is necessary to adjust the sparse matrix and platform parameters. The Optimized Sparse Kernel Interface (OSKI) library [120] is a state-of-the-art collection of sparse matrix operation primitives on single-core cache-based microprocessors that relies on the SPARSITY framework [68] to tune the SpMV kernel. Its major optimization strategy includes register blocking and cache blocking. Autotuning is employed by Choi et al. [27] and Monakov et al. [92] to find the best block sizes and the slice sizes of the given sparse matrices on Nvidia GPUs. Guo and Wang [63] also autotune the parameters of the CSR SpMV implementation on Nvidia GPUs. Grewe and Lokhmotov [58] develop a code generator to create CUDA and OpenCL code for SpMV kernels to facilitate the autotuning process – however, their paper focuses on the generated CUDA code. For OpenCL code, only CSR SpMV results are presented, so it is unclear how their generator would perform on other sparse matrix formats.

In the OLOV project, we have developed the *clSpMV* autotuner, which optimizes sparse matrix data structures and SpMV performance on GPU platforms [112].

6.3.1 Exploring the Design Space of Algorithms

We optimize the SpMV computation from two directions: proposing a new sparse matrix format, and performing optimizations that are specific to underlying GPU platforms.

Introduction to the Cocktail Format

Many SpMV studies have developed novel sparse matrix formats. However, there is no one-size-fits-all solution: every sparse matrix representation has its own strengths and weaknesses. The symmetric Compressed Sparse Block (CSB), the bitmasked register block data structures by Buluc et al. [23], the Blocked Compress Sparse Row (BCSR), and the Sliced Blocked ELLPACK (SBELL) data structures by Choi et al. [27] all assume dense blocks in the sparse matrix. The performances of the Sliced ELLPACK (SELL) format by

Monakov et al. [92] and the ELLPACK-R format by Vázquez et al. [118] rely on variations in the number of non-zeros per row in the sparse matrix. The experimental results from Bell and Garland [14] also demonstrate that the best SpMV results are heavily dependent on the choice of sparse matrix format.

Based on the observation that different sparse matrix formats are good at different sparse matrix structures, we have developed the *Cocktail Format* to take advantage of the different matrix formats. The *Cocktail Format* is a combination of many different sparse matrix formats. It partitions a given matrix into several submatrices, each specialized for a given matrix structure. The trivial case finds a single best format for a given sparse matrix. The most complicated case partitions the sparse matrix into many submatrices and represents them using different formats. The list of sparse matrix formats in the *Cocktail Format* can be arbitrary. In *clSpMV*, we support nine sparse matrix formats in three categories. These categories and matrix formats are summarized below, and further explained in later paragraphs.

- Diagonal-based category: formats that store dense diagonals.
  - DIA: stores dense diagonals.
  - BDIA: stores a band of diagonals together.
- Flat category: formats that require a column index for every non-zero data point on the sparse matrix.
  - ELL: packs non-zeros into a dense matrix.
  - SELL: cuts the matrix into slices, using different ELL settings for each slice.
  - CSR: the common compressed sparse row format.
  - COO: the common coordinate format.
- Block-based category: formats that store dense blocks.
  - BELL: the blocked variant of the ELL format.
  - SBELL: the blocked variant of the SELL format.
  - BCSR: the blocked variant of the CSR format.

There is also research that partitions a sparse matrix into many submatrices. However, the partitions are very restricted. Vuduc [121] partitions the sparse matrix into two to four submatrices with different dense block sizes. However, he focuses on only the BCSR format. If the number of dense blocks per row is regular, the BELL format is a better choice. Bell and Garland [14] partition the sparse matrix into the ELLPACK portion and the COO portion; however, they do not take advantage of dense blocks in the matrices. The *Cocktail Format* is the first proposal that partitions the matrix into many different specialized regions.

Diagonal-Based Formats

$$B = \begin{bmatrix} 3 & 7 & 0 & 0 \\ 0 & 4 & 8 & 0 \\ 1 & 0 & 5 & 9 \\ 0 & 2 & 0 & 6 \end{bmatrix} \tag{6.1}$$

Diagonal-based formats capture dense diagonals. We use matrix B in Equation (6.1) to explain the data structure of the formats in this category.

```
Offsets  =  [-2  0  1]
Data     =  [0  0  1  2,  3  4  5  6,  7  8  9  0]
```

Figure 6.2: The DIA format of matrix B.

DIA Format As explained in [14], the diagonal (DIA) format is composed of two arrays: the `Offsets` array that stores offsets of each diagonal, and the `Data` array that stores dense diagonals. The `Data` array is a 1D array, the commas are used to separate different diagonals for representation convenience. Figure 6.2 shows the DIA format of matrix B. There are advantages and disadvantages to the DIA format:

- Advantages: The DIA format does not require explicit column indices for each non-zero datum. The format has single-stride and aligned access on the matrix data. It also has single-stride access on the multiplied vector.
- Disadvantages: The DIA format requires zero fillings in the `Data` array to ensure that the lengths of all diagonals are the same. On sparse diagonals and short diagonals, the zero filling overhead might be significant.

```
Offsets  =  [-2  0]
BandPtr  =  [0  4  12]
Data     =  [0  0  1  2,  3  4  5  6,  7  8  9  0]
```

Figure 6.3: The BDIA format of matrix B.

BDIA Format The banded diagonal (BDIA) format is a variation of the DIA format. Instead of storing disjoint diagonals, it stores a band as a whole. A band is a series of consecutive diagonals. This format is composed of three arrays. The `Offsets` array stores the offset of the first diagonal in each band. The `BandPtr` array stores the position of the first element of each band – in other words, the elements of band i are stored between `BandPtr`$[i]$ and `BandPtr`$[i+1]$. The `Data` array is exactly the same as in the DIA format. Figure 6.3 shows the BDIA format of matrix B. The offset of the first band is -2, and

the offset of the second band is 0. The starting position of the first band is 0 in the `Data` array, and the starting position of the second band is 4 in the `Data` array. There are also advantages and disadvantages to the BDIA format:

- Advantages: The BDIA format does not require explicit column indices for each non-zero datum. This format has single-stride and aligned access on the matrix data. It also has single-stride access on the multiplied vector. It can use shared local memory to cache the multiplied vector.
- Disadvantages: The BDIA format requires zero fillings in the data array to ensure that the lengths of all diagonals are the same. On sparse diagonals, the zero filling overhead might be significant. Compared to the DIA format, BDIA requires an additional `BandPtr` array.

Flat Formats

Flat formats require explicit storage of the column indices of all non-zeros. We use matrix B to explain the data structure of the formats in this category.

ELL Format The ELLPACK (ELL) format [60] packs all non-zeros towards the left, and stores the packed dense matrix. Assuming that the packed dense matrix has dimension $m \times n$, the ELL width of the original matrix will be n, the width of the packed dense matrix. The ELL format is composed of two arrays: the `Col` array stores the column indices of all elements in the dense $m \times n$ matrix, and the `Data` array stores the values of the dense $m \times n$ matrix. Figure 6.4 shows the ELL format of matrix B. The ELL format has several advantages and disadvantages:

```
Col  =  [0 1 0 1,  1 2 2 3,  0 0 3 0]
Data =  [3 4 1 2,  7 8 5 6,  0 0 9 0]
```

Figure 6.4: The ELL format of matrix B.

- Advantages: The access pattern of the `Col` and `Data` arrays is single-stride and aligned.
- Disadvantages: Assuming that the packed dense matrix has dimension $m \times n$, the ELL format needs zero paddings on every row that has fewer non-zeros than n. These zero paddings potentially introduce significant overhead. This format also requires random access on the multiplied vector.

SELL Format The sliced ELLPACK (SELL) format is proposed by Monakov et al. [92]. The idea is to cut the original matrix into slices, then pack the slices into dense matrices with different dimensions. The slices are constructed by cutting the matrix horizontally. The SELL format is composed of three arrays. The `SlicePtr` array stores the beginning position

```
SlicePtr  =  [0  4  10]
Col       =  [0  1,  1  2;  0  2,  1  3,  3  0]
Data      =  [3  4,  7  8;  1  2,  5  6,  9  0]
```

Figure 6.5: The SELL format of matrix B.

of each slice in both `Col` and `Data` arrays – that is, the elements of slice i are stored between `SlicePtr`$[i]$ and `SlicePtr`$[i+1]$. The `Col` and `Data` arrays are similar to the ELL format, storing the column indices and values of each element in the slices. Figure 6.5 shows the SELL format of matrix B with slice height 2. The semicolons in the array separate different slices. Monakov et al. [92] employ autotuning to determine the best slice height, reorder the matrix to further reduce zero paddings, and propose the variable-height slice format. According to their experimental results, the matrix reordering technique and the variable-height format result in only marginal improvements. Since these strategies might increase the complexity of the policies in the online decision-making stage, the current *clSpMV* does not include these approaches. Regarding the slice height, we develop kernels with slice heights equal to multiples of the GPU alignment requirement (128 bytes). There are advantages and disadvantages to the SELL format:

- Advantages: The access pattern of the `Col` and `Data` arrays is single-stride and aligned. The format requires fewer zero paddings compared to the ELL format.
- Disadvantages: The SELL format still requires zero paddings for each slice, and these zero paddings potentially introduce significant overhead. The format requires random access on the multiplied vector. It also needs an additional `SlicePtr` array to store the slice positions.

```
RowPtr  =  [0  2  4  7  9]
Col     =  [0  1,  1  2,  0  2  3,  1  3]
Data    =  [3  7,  4  8,  1  5  9,  2  6]
```

Figure 6.6: The CSR format of matrix B.

CSR Format The compressed sparse row (CSR) format is the most common sparse matrix format. It is composed of three arrays. The `RowPtr` array stores the beginning position of each row in both `Col` and `Data` arrays – the elements of row i are stored between `RowPtr`$[i]$ and `RowPtr`$[i+1]$. The `Col` and `Data` arrays are used to store the column indices and values of each non-zero. Figure 6.6 shows the CSR format of matrix B. The CSR format has one primary advantage and several disadvantages:

- Advantages: The CSR format requires very few zero paddings.

- Disadvantages: The CSR format might have unaligned access on both the `Col` and `Data` arrays. The access pattern on the multiplied vector is random. This format might also have load balancing problems if the number of non-zeros per row varies significantly.

```
Row  =  [0  0,  1  1,  2  2  2,  3  3]
Col  =  [0  1,  1  2,  0  2  3,  1  3]
Data =  [3  7,  4  8,  1  5  9,  2  6]
```

Figure 6.7: The COO format of matrix B.

COO Format The coordinate (COO) format explicitly stores row indices. It is composed of three arrays – the `Row`, `Col`, and `Data` arrays store row indices, column indices, and the values of all non-zeros in the matrix, respectively. Figure 6.7 shows the COO format of matrix B. There are advantages and disadvantages to using the COO format:

- Advantages: The COO format requires very few zero paddings. There is no load balancing problem. As shown in [14], this format can deliver consistent performance regardless of the structure of the matrix.
- Disadvantages: The COO format has the worst memory footprint. The format requires explicit indexing on both row and column indices. It also requires random access on the multiplied vector.

Block-Based Formats

$$C = \begin{bmatrix} 0 & 1 & 2 & 3 & g & h & i & j \\ 4 & 5 & 6 & 7 & k & l & m & n \\ 8 & 9 & a & b & o & p & q & r \\ c & d & e & f & s & t & u & v \end{bmatrix} \tag{6.2}$$

Block-based formats are variations of flat formats. Instead of storing each non-zero independently, these formats store a block contiguously. We use matrix C in Equation (6.2) to show examples of block-based formats. For block sizes, because AMD platforms always prefer the *float4* data type [3], while Nvidia platforms achieve similar performance on both *float* and *float4* data types, we employ block sizes that are multiples of 4. Moreover, when using texture memory to cache the multiplied vector, the OpenCL API always returns a *float4* value. If we do not use all four elements in the returned value, memory bandwidth is wasted. Therefore, using block sizes with widths being multiples of 4 is always an advantage. The block sizes supported by *clSpMV* are 1×4, 2×4, 4×4, 8×4, 1×8, 2×8, 4×8, and 8×8. A blocked matrix is represented by blocks instead of singular elements. According to the block height, we define a super-row of a matrix to be a submatrix of consecutive h rows, where h is the height of the block. In other words, a super-row is the smallest row unit that can operate on a blocked matrix.

```
Col  = [ 0  0,  4  4  ]
Data = [ 0  1  2  3,  8  9  a  b,
         4  5  6  7,  c  d  e  f;
         g  h  i  j,  o  p  q  r,
         k  l  m  n,  s  t  u  v  ]
```

Figure 6.8: The BELL format of matrix C. The block size is 2×4.

BELL Format The blocked ELLPACK (BELL) format is a variation of the ELL format. Instead of storing singular non-zeros, it stores a block of consecutive non-zeros. Each block requires only one column index, so the memory footprint is reduced. If the height of the block is larger than 1, the same data read from the multiplied vector can be reused across all the rows in the block. The BELL format is composed of two arrays: the `Col` array stores the column indices of the first elements from all blocks, and the `Data` array stores the values of all blocks. Moreover, we need special arrangement to enforce single-stride memory access on the `Data` array. Because the 1×4 block is the smallest unit of all block sizes we support, the `Data` array is managed in a 2D fashion. The first dimension corresponds to the data of a 1×4 block, and the second to the number of 1×4 units in the block dimension. Figure 6.8 shows the BELL format of matrix C. The block size is 2×4, so there are two 1×4 units in the block. The `Data` array can be viewed as a 2×16 array. We store the first 1×4 unit of each block, and then store the next 1×4 unit of each block. The BELL format has several advantages and disadvantages:

- Advantages: The BELL format offers single-stride data access on the `Data` array. The required memory storage of the column indices is reduced. If the block has a height larger than 1, the segment of the multiplied vector can be cached in registers and used across multiple block rows.
- Disadvantages: The BELL format requires zero fillings in the sparse blocks. It also requires zero paddings to make sure that the number of blocks per super-row is all the same. These fillings and paddings might introduce overhead.

```
SlicePtr = [ 0  4  8  ]
Col      = [ 0  0,  4  4;  0  0,  4  4  ]
Data     = [ 0  1  2  3,  4  5  6  7,
             g  h  i  j,  k  l  m  n;
             8  9  a  b,  c  d  e  f,
             o  p  q  r,  s  t  u  v  ]
```

Figure 6.9: The SBELL representation of matrix C. The block size is 1×4, and the slice height is 2.

SBELL Format The sliced blocked ELLPACK (SBELL) format is proposed by Choi et al. [27]. Although the data arrangement in *clSpMV* is different from that of Choi et al. [27], the idea is similar. In *clSpMV*, the SBELL format is composed of three arrays. The `SlicePtr` array stores the beginning position of each slice in the `Col` array – that is, the elements of slice i are stored between `SlicePtr`$[i]$ and `SlicePtr`$[i+1]$. The `Col` array stores the column indices of the first elements from all blocks. The `Data` array stores the values of all blocks. The data of a slice are stored consecutively. Like the case in the BELL format, in a slice the data of a 1×4 unit are stored consecutively, and the data of multiple 1×4 units are stacked into a large array. Figure 6.9 shows the SBELL format of matrix C, with block size 1×4 and slice height 2. Choi el al. [27] also rearrange the matrix rows to reduce the paddings of the SBELL format. Because we are using the SBELL format to represent only the portion of the matrix that is best for SBELL, the remaining singular non-zeros should be addressed by the flat formats. Therefore, we do not reorder the matrix in our implementation. There are advantages and disadvantages to the SBELL format:

- Advantages: The SBELL format offers single-stride data access on the `Data` array. The required memory storage of the column indices is reduced. If the block has a height larger than 1, the segment of the multiplied vector can be cached in registers and used across multiple block rows. This format requires fewer zero paddings compared to the BELL format.
- Disadvantages: The SBELL format requires zero fillings in the sparse blocks. It also needs zero paddings to ensure that the number of blocks per super-row in a slice is all the same. These fillings and paddings might introduce overhead. This format also requires an additional `SlicePtr` array.

```
RowPtr  =  [  0  2  4  ]
Col     =  [  0  4,  0  4  ]
Data    =  [  0  1  2  3,  g  h  i  j,
              8  9  a  b,  o  p  q  r;
              4  5  6  7,  k  l  m  n,
              c  d  e  f,  s  t  u  v  ]
```

Figure 6.10: The BCSR format of matrix C. The block size is 2×4

BCSR Format The blocked compressed sparse row (BCSR) format is also discussed by Choi et al. [27]. The data arrangement in *clSpMV* is different, but the idea is similar. The BCSR format is composed of three arrays. The `RowPtr` array stores the beginning position of each super-row in the `Col` array – the elements of super-row i are stored between `RowPtr`$[i]$ and `RowPtr`$[i+1]$. The `Col` array stores the column indices of the first elements from all blocks, and the `Data` array stores the values of all blocks. As with the BELL format, the data of a 1×4 unit are stored consecutively, and the data of multiple 1×4 units are

Table 6.1: Advantages and disadvantages of sparse matrix format categories.

Sparse Matrix Format Category	Advantages	Disadvantages
Diagonal-Based	No explicit column indices. Single-stride access on the matrix and vector.	Requires zero fillings on sparse diagonals.
Block-Based	Partial explicit column indices. Better reuse of the vector.	Requires zero fillings on sparse blocks.
Flat	No zero fillings.	Requires explicit column indices.

Table 6.2: Advantages and disadvantages of diagonal-based formats.

Diagonal-Based Format	Advantages	Disadvantages
DIA	Does not require the `BandPtr` array.	Cannot use shared memory to cache the vector.
BDIA	Can use shared memory to cache the vector.	Requires the `BandPtr` array.

stacked into a large array. Figure 6.10 shows the BCSR format of matrix C, with block size 2×4. There are several advantages and disadvantages to the BCSR format:

- Advantages: The required memory storage of the column indices is reduced with the BCSR format. If the block has a height larger than 1, the segment of the multiplied vector can be cached in registers and used across multiple block rows. This format does not need to pad zero blocks at the end of each super-row.
- Disadvantages: The BCSR format requires zero fillings in the sparse blocks. It might have unaligned access on the `Data` array. It also might have load balancing problem.

We summarize the advantages and disadvantages of the three sparse matrix format categories in Table 6.1, the two diagonal-based formats in Table 6.2, the four flat formats in Table 6.3, and the three block-based formats in Table 6.4.

The *clSpMV* Autotuner

Based on the *Cocktail Format*, every sparse matrix can be partitioned into many sub-matrices. However, it is challenging to find the best partitioning scheme of a given sparse matrix. Moreover, each sparse format can have many different parallelization strategies. Assume that the *Cocktail Format* is composed of k sparse matrix formats $f_1, f_2, \ldots, f_k$. Let F be the superset over all k sparse matrix formats, $\bigcup_{i=1}^{k} f_i = F$. For a matrix format f_i, assume there exist b_i implementations p_{i1}, p_{i2}, ..., p_{ib_i}. Let P_i be the superset over all implementations of format i, $\bigcup_{j=1}^{b_i} p_{ij} = P_i$. Let $t(A, f_i, p_{ij})$ be the execution time of the SpMV kernel using format f_i and implementation p_{ij} on matrix A. The matrix partitioning problem can then be formulated as follows:

Table 6.3: Advantages and disadvantages of flat formats.

Flat Format	Advantages	Disadvantages
ELL	Single-stride and aligned access to `Data` and `Col`. Does not require the `SlicePtr` array.	Requires zero paddings on short rows.
SELL	Single-stride and aligned access to `Data` and `Col`. Fewer zero paddings compared to ELL.	Requires zero paddings on short rows. Requires the `SlicePtr` array.
CSR	Very few zero paddings.	Unaligned access to `Data` and `Col`. Bad load balance.
COO	Very few zero paddings. Good load balance.	Threads might stall in reduction. Requires explicit row indices.

Table 6.4: Advantages and disadvantages of block-based formats.

Block-Based Format	Advantages	Disadvantages
BELL	Single-stride and aligned access to `Data`. Does not require the `SlicePtr` array.	Requires zero paddings on short super-rows.
SBELL	Single-stride and aligned access to `Data`. Fewer zero paddings compared to BELL.	Requires zero paddings on short super-rows. Requires the `SlicePtr` array.
BCSR	Very few zero paddings.	Unaligned access to `Data`. Bad load balance.

- Problem *CMP* (*Cocktail Matrix Partitioning*): Given sparse matrix A, the k sparse formats in the *Cocktail Format*, and the b_i implementations of format f_i, find k sub-matrices $A_1, A_2, \ldots, A_k$ and k implementations $L_1, L_2, \ldots, L_k$ such that $\sum_{i=1}^{k} A_i = A$, $L_1 \in P_1, L_2 \in P_2, \ldots, L_k \in P_k$, and the value of $\sum_{i=1}^{k} t(A_i, f_i, L_i)$ is minimized.

The CMP problem is an NP-complete problem. For a sparse matrix with n non-zeros, and the *Cocktail Format* with k formats, the size of the sample space is $O(k^n \times \max_{1 \le i \le k} b_i)$. If we allow the same non-zeros to be covered by multiple formats, the sample space is even larger. Moreover, function $t(A, f_i, p_{ij})$ is nonlinear. The actual execution time will depend on the thread scheduler, system load, cache behavior, and many other factors.

Overall Structure of the *clSpMV* Autotuner

In addition to the CMP problem, we must also compute the $t(A, f_i, p_{ij})$ function. When multiple implementations of a single sparse matrix format are available, most autotuners execute all implementations exhaustively to find the best implementation [92, 58, 63]. This

strategy gives us the exact $t(A, f_i, p_{ij})$ value, but is very time-consuming. For the *Cocktail Format*, the brute force search strategy involves expensive reformatting of the input matrix because the submatrices may need to be adjusted frequently. This overhead is unacceptable. Some autotuners develop models of specific architectures and predict performance based on these models [27]. This strategy is applicable, but requires significant effort – for each platform we support, we would need to develop its performance model, then apply that performance model on every implementation. When a new platform is released, we would need to undertake the entire procedure again. For portability concerns, this is not the best strategy.

Following the philosophy of OSKI [120], the *clSpMV* autotuner is composed of two stages: the offline benchmarking stage, and the online decision-making stage. The goal of offline benchmarking is to sample performance data with different sparse matrix settings, and to provide a way of estimating the value of $t(A, f_i, p_{ij})$. The online decision-making stage then solves the CMP problem.

The Offline Benchmarking Stage

The purpose of the offline benchmarking stage is to solve the performance approximation problem. Given a sparse matrix format f_i and a corresponding implementation p_{ij}, the offline benchmarking stage samples the execution time on different sparse matrix settings. These settings include matrix dimensions, total number of non-zeros, average number of non-zeros per row, variations in the number of non-zeros per row, and so forth. The sample density controls trade-offs between approximation accuracy and offline benchmarking time. More data points with wider matrix settings yield better approximation results, but require more offline benchmarking time. Given an arbitrary sparse matrix, the execution time can be approximated by interpolating nearby data points.

In our current implementation, we consider only matrix dimensions and average number of non-zeros per row, and we benchmark on dense banded matrices. When sampling on the matrix dimensions, we want representative data for the case that the processors are under-utilized most of the time, and for the case that all processors are saturated. Therefore, we employ an exponential scale: $2^{10}, 2^{11}, \ldots, 2^{21}$. When sampling on the number of non-zeros per row, we must cover from the case that the matrix is extremely sparse to the case that every row has enough work to saturate all processors. As will be discussed in Section 6.3.2, the parallelization strategies for different formats are different, so the sampling density of different formats is also different. If the parallelization strategy is based on having different work items working on independent rows, then having the number of non-zeros range from 1 to 64 should be enough. On the other hand, if the parallelization strategy is based on having multiple work items on the same row, then we need hundreds to thousands of non-zeros per row to saturate the processors.

For the nine sparse matrix formats, we have 75 implementations in total, and we must collect hundreds of sample points for each implementation. Including generating sparse matrices with different dimensions and numbers of non-zeros per row, the offline benchmarking stages on both Nvidia and AMD platform take about half a day. However, this is a one-time cost – we must only benchmark on a platform once, and the results can be applied to tune

the performance of SpMV on as many sparse matrices as we want.

The Online Decision-Making Stage

This stage solves the CMP problem by analyzing the input sparse matrix. We achieve this by collecting matrix features and enforcing partition policies.

Transforming a matrix from one format to another is both time and memory consuming. The *clSpMV* autotuner explores the design space of 30 different formats (block-based formats with different block dimensions are considered different formats here), so it is infeasible to analyze the structures of all the matrix formats by converting to those formats. Instead, we collect only statistical features that are representative of different matrix formats. When collecting features for diagonal formats, we maintain a histogram to count the number of non-zeros per diagonal. When collecting features for blocked formats, we maintain two histograms for each block dimension – one counts the number of blocks per super-row, and the other counts only the number of dense blocks per super-row. The definition of a dense block is given in the partition policies. The first histogram is used to estimate the execution time if we store the entire matrix using block-based formats; the second is used to estimate the execution time if we store only the dense blocks of the matrix. When collecting features for flat formats, we maintain a histogram to count the number of non-zeros per row. These feature histograms are used to capture the characteristics of a matrix under different formats. We also make our partition decisions based on the collected feature histograms.

The solution space of the CMP problem is enormous, so we use greedy policies to partition the sparse matrix. These policies are based on our analysis of the strengths and weaknesses of the formats. According to the nine supported sparse matrix formats, we apply the following policies:

- Priority of the categories: the priorities of the three categories (diagonal-based, flat, and block-based) are decided by the maximum estimated performance of each category according to the current matrix settings explored in the offline benchmarking stage.

- Dense Diagonals: let g_d be the maximum GFLOPS that the diagonal category can achieve under the current matrix settings, and let g_f be the maximum GFLOPS that the flat category can achieve under the current settings. A diagonal with dimension n_d is considered to be dense if its non-zero number e_d satisfies the following formula:

$$e_d > n_d \times \frac{g_f}{g_d}$$

- BDIA vs. DIA: choose the maximum achievable GFLOPS in the following three cases: using only BDIA, using only DIA, or using BDIA to represent thick bands and DIA to represent disjoint diagonals or thin bands.

- Dense Blocks: let g_d be the maximum GFLOPS that the block-based category can achieve under the current matrix settings and the given block size. A block with size n_b is considered to be dense if its non-zero number e_b satisfies the following formula:

$$e_b > n_b \times \frac{g_f}{g_b}$$

- SBELL vs. BELL vs. BCSR: choose the maximum achievable GFLOPS in the following three cases: using only SBELL, using only BELL, or using only BCSR.

- ELL and SELL vs. CSR and COO: let the maximum achievable GFLOPS of ELL, SELL, CSR, and COO be g_{ELL}, g_{SELL}, g_{CSR}, and g_{COO}, respectively. Use CSR and COO if $m_c = \max(g_{CSR}, g_{COO}) > m_e = \max(g_{ELL}, g_{SELL})$. Otherwise, extract the ELL and SELL portion first, then represent the remaining non-zeros using CSR or COO.

- Extract ELL: let w be the ELL width, let c be the number of columns of the matrix, let $z(w)$ be the zero paddings when the ELL width is w, and let $e(w)$ be the non-zeros covered by the ELL format with width w. w is then decided by solving the following:

$$\begin{array}{ll} \max & w \\ s.t. & (z(w) + e(w))/g_{ELL} < e(w)/m_c \\ & w \leq c \\ & w \in N \end{array}$$

  The possible values of w are bounded by c, so we apply the brute force method to solve this problem.

- Extract SELL: the idea is the same as extracting ELL – the only difference is considering the ELL width of each slice independently.

- ELL vs. SELL: choose the one that has the higher achievable GFLOPS value.

- CSR vs. COO: this decision is based on the load balancing issue of CSR. Assume that there are u work groups in the CSR implementation. Let $nnz(i)$ be the number of non-zeros computed by work group i. For a matrix with n non-zeros, use the CSR format if the following rule is satisfied:

$$\frac{u \times \max\limits_{1 \leq i \leq u} nnz(i)}{g_{CSR}} < \frac{n}{g_{COO}}$$

- Merge small submatrices: merge a submatrix into another existing submatrix if such behavior results in better estimated performance.

The *Cocktail Format* is a superset over many single sparse matrix representations. An input sparse matrix is partitioned into one or more submatrices, and each of which is represented by a specific format. Theoretically, the SpMV performance of the *Cocktail Format* should be at least the same as the SpMV performance of every format it covers. In practice, however, performance depends on the policies of the *clSpMV* autotuner and the accuracy of the estimated execution time (the value of the $t(A, f_i, p_{ij})$ function). This is a trade-off between analysis time and SpMV performance. If we apply more complicated policies and more accurate execution time estimates, we can find better matrix partitions and achieve higher SpMV performance, but this requires more analysis time. The analysis overhead of

OSKI is about 40 SpMVs. *clSpMV* takes 1 SpMV for diagonal analysis, 20 SpMVs for block analysis of one block size, and 4 SpMVs for flat analysis.

Because the matrix analysis overhead is not trivial, *clSpMV* is ideal for iterative methods that perform SpMV on a single sparse matrix hundreds or thousands of times. Moreover, if the autotuner user is dealing with matrices with similar structures, one can perform a full analysis on example matrices and then use the results to speed up the analysis of future matrices. For example, if the exemplar matrices do not have any dense blocks, one can advise *clSpMV* to skip the analysis of the block-based formats. To further reduce the analysis overhead, we can also follow the strategy used by Vuduc [121] – instead of analyzing the entire matrix, we can randomly sample the non-zeros of the matrix and make decisions based on these samples. Since most of the analysis is based on counting (tallying the number of non-zeros in a diagonal, a block, or a row), we can also parallelize the analysis procedure to further reduce the overhead.

6.3.2 Exploring the Design Space of Parallelization Strategies

The idea of the *Cocktail Format* is to take advantage of the strengths of a set of different sparse matrix formats. The *clSpMV* autotuner plugs in multiple implementations of many SpMV kernels, performs analysis on the matrices, and decides the best partition for the matrices. No matter what the final decomposition of the matrices is, SpMV performance depends fully on the implementations of all supported formats. Because different platforms have different characteristics, there is no one-size-fits-all solution. To achieve the best performance, the implementations of the SpMV kernels should be platform-dependent.

Because we target GPU platforms, we choose parallelization strategies that express data parallelism on GPUs. Moreover, we employ several parallelization strategies for each format. The *clSpMV* autotuner can select the best parallelization strategy based on offline benchmarking information. Here, we introduce the parallelization strategies for each sparse matrix format.

Diagonal-Based Formats

DIA Format The parallelization strategy of the DIA kernel is similar to [14] – each work item is responsible for one row of the matrix. Because AMD platforms favor explicit vectorization by using the *float4* data type [3], we also implement the kernel that each work item is responsible for four rows of the matrix. Moreover, we implement kernels that use texture memory and kernels that do not use texture memory to store the multiplied vector.

BDIA Format The parallelization strategy of BDIA is very similar to that of the DIA format. We have implementations where each work item is responsible for one matrix row, and implementations where each work item is responsible for four matrix rows using the *float4* data type. The major difference between the DIA and BDIA formats is that the diagonals in a band are consecutive, so we can predict the vector sections that each work item is accessing. For example, assume that the size of a work group is 128, and that work items r to $r + 127$ in this work group are responsible for row r to row $r + 127$, respectively.

Considering a band with d diagonals, and the offset of the first diagonal being o, work item i will access vector elements $o+i, o+i+1, \ldots, o+i+d-1$. The entire work group will access vector elements $o+r, o+r+1, \ldots, o+r+127+d-1$. The consecutive vector section can be cached into the shared local memory. We have implemented kernels that employ this caching strategy, and kernels that do not employ this caching strategy.

Flat Formats

ELL Format The parallelization strategy is similar to [14]. Each work item is responsible for one row of the matrix. Considering platforms that favor explicit vectorization such as AMD GPUs, we also have the implementation that each work item is responsible for four rows using the *float4* data type. Again, kernels using texture memory and not using texture memory to cache the multiplied vector are all included.

SELL Format The parallelization strategy is the same as that in the ELL format. The only difference is that we cache the `SlicePtr` array in the local shared memory.

CSR Format Bell and Garland [14] propose two parallelization strategies for the CSR format. The scalar strategy allows one work item working on one row of the matrix. The vector strategy allows one warp of work items working on one row of the matrix. According to [14], the scalar strategy outperforms the vector strategy only when the number of non-zeros per row is small. However, when the number of non-zeros per row is small, the ELL format is a better candidate. Therefore, we implement only the vector strategy in *clSpMV*. Again, implementations using texture memory and not using texture memory are both included.

COO Format The parallelization strategy is the same as [14]. We perform segmented reduction computation on the three arrays in the COO format. However, the implementation in [14] requires three kernel launches. By padding zeros at the end of three arrays to match the work group size, we need only two kernel launches in our OpenCL implementation.

Block-Based Formats

BELL Format The parallelization strategy is similar to the ELL format. However, instead of letting one work item work on a row, we let one work item work on a super-row. Because the smallest unit of all block sizes is 1×4, we employ the *float4* data type in our implementation.

SBELL Format The parallelization strategy is similar to the BELL format. Each work item is responsible for one super-row. The only difference is that we cache the `SlicePtr` array in the local shared memory.

BCSR Format The parallelization strategy is similar to the vector CSR strategy used in [14]. However, instead of letting one warp of work items work on one row, we now let one warp of work items work on one super-row.

6.3.3 Exploring the Design Space of Platform Parameters

Here, we explore the design space of platform parameters on a subset of sparse matrix formats supported by *clSpMV*.

CSR Format The parallelization strategy of the CSR format is to use a warp of work items working on one row. However, different GPUs can host different numbers of warps in a processor. If we create too many warps, the scheduling overhead grows; conversely, if we create too few, we cannot saturate the processors. Ideally, the number of warps should just match the maximum capacity of the underlying GPU. To find this optimal number, *clSpMV* evaluates the performance of different numbers of work groups in the offline benchmarking stage. The number of warps can then be computed from the number of work groups. Based on this offline benchmarking information, *clSpMV* selects the best configuration to optimize the CSR SpMV kernel.

COO Format The parallelization strategy of the COO format is to perform a segmented reduction on the three COO arrays. The basic idea of segmented reduction is to create a fixed number of work groups, then let the work groups go over the arrays iteratively until the arrays are fully processed. As a result, similar to the case of the CSR SpMV kernel, the performance of the COO SpMV kernel is related to the number of work groups created. *clSpMV* evaluates the performance of the COO SpMV kernel with different numbers of work groups in the offline benchmarking stage. In the online decision-making stage, *clSpMV* selects the COO SpMV kernel with the best work group configurations.

Block-Based Formats The block-based formats include BELL, SBELL, and BCSR. These formats are similar to their flat variations – the only difference is that the flat formats store one non-zero at a time, while the block-based formats store one block of non-zeros together. The common numeric configurations of these formats are the block sizes. Larger block sizes use less memory to store column indices, but might increase the number of zero-fillings; smaller block sizes use more memory to store column indices, but might decrease the number of zero-fillings. Using a larger or a smaller block size is therefore matrix dependent. *clSpMV* supports eight different block sizes: 1×4, 2×4, 4×4, 8×4, 1×8, 2×8, 4×8, and 8×8. The performance of these different configurations is collected during the offline benchmarking stage. In the online decision-making stage, *clSpMV* then selects the ideal block size based on this offline benchmarking information and the input matrix characteristics.

6.3.4 Experimental Results

We can evaluate the performance of *clSpMV* on different platforms, given the cross-platform capabilities of OpenCL. Nvidia and AMD are the major GPU vendors, so we evaluate the autotuner's performance on these two different platforms. Since both platforms achieve higher performance on the single precision floating point data type, we use such a data type in our experiments. In this section, we first introduce the matrices used for benchmarking, and then discuss the performance of *clSpMV* on an Nvidia GTX 480 and an AMD Radeon 6970.

The Benchmarking Matrices

We use the 14 matrices from Williams et al. [123] for our benchmarking. This same set is also used in other SpMV research [14, 92, 27]. The statistics of the matrices are summarized in Table 6.5. The # rows column, the # cols column, the # nnzs column, and the nnz/row column summarize the number of rows, number of columns, number of non-zeros, and average number of non-zeros per row, respectively. Unfortunately, this set contains mostly regular matrices that are well-suited for single-format representation. Although *clSpMV* is able to determine the best single format to represent the matrices, it is unlikely that the *Cocktail Format* will further improve performance. Therefore, we add six additional matrices from the University of Florida Sparse Matrix Collection [37] to our benchmarking suite. We choose matrices with balanced portions of diagonals, dense blocks, and random singular non-zeros; matrices with highly irregular distributions of non-zeros; and matrices with enough non-zeros that the overhead of launching multiple kernels will not be significant. The statistics of these six additional matrices are also summarized in Table 6.5.

Table 6.5: Overview of the sparse matrix benchmark.

Name	# rows	# cols	# nnzs	nnz/row
Dense	2K	2K	4M	2000
Protein	36K	36K	4.3M	119
FEM/Spheres	83K	83K	6M	72
FEM/Cantilever	62K	62K	4M	65
Wind Tunnel	218K	218K	11.6M	53
FEM/Harbor	47K	47K	2.37M	50
QCD	49K	49K	1.9M	39
FEM/Ship	141K	141K	3.98M	28
Economics	207K	207K	1.27M	6
Epidemiology	526K	526K	2.1M	4
FEM/Accelerator	121K	121K	2.62M	22
Circuit	171K	171K	959K	6
Webbase	1M	1M	3.1M	3
LP	4K	1.1M	11.3M	2825
circuit5M	5.56M	5.56M	59.5M	11
eu-2005	863K	863K	19M	22
Ga41As41H72	268k	268k	18M	67
in-2004	1.38M	1.38M	17M	12
mip1	66K	66K	10M	152
Si41Ge41H72	186k	186k	15M	81

Experimental Results on Nvidia GTX 480

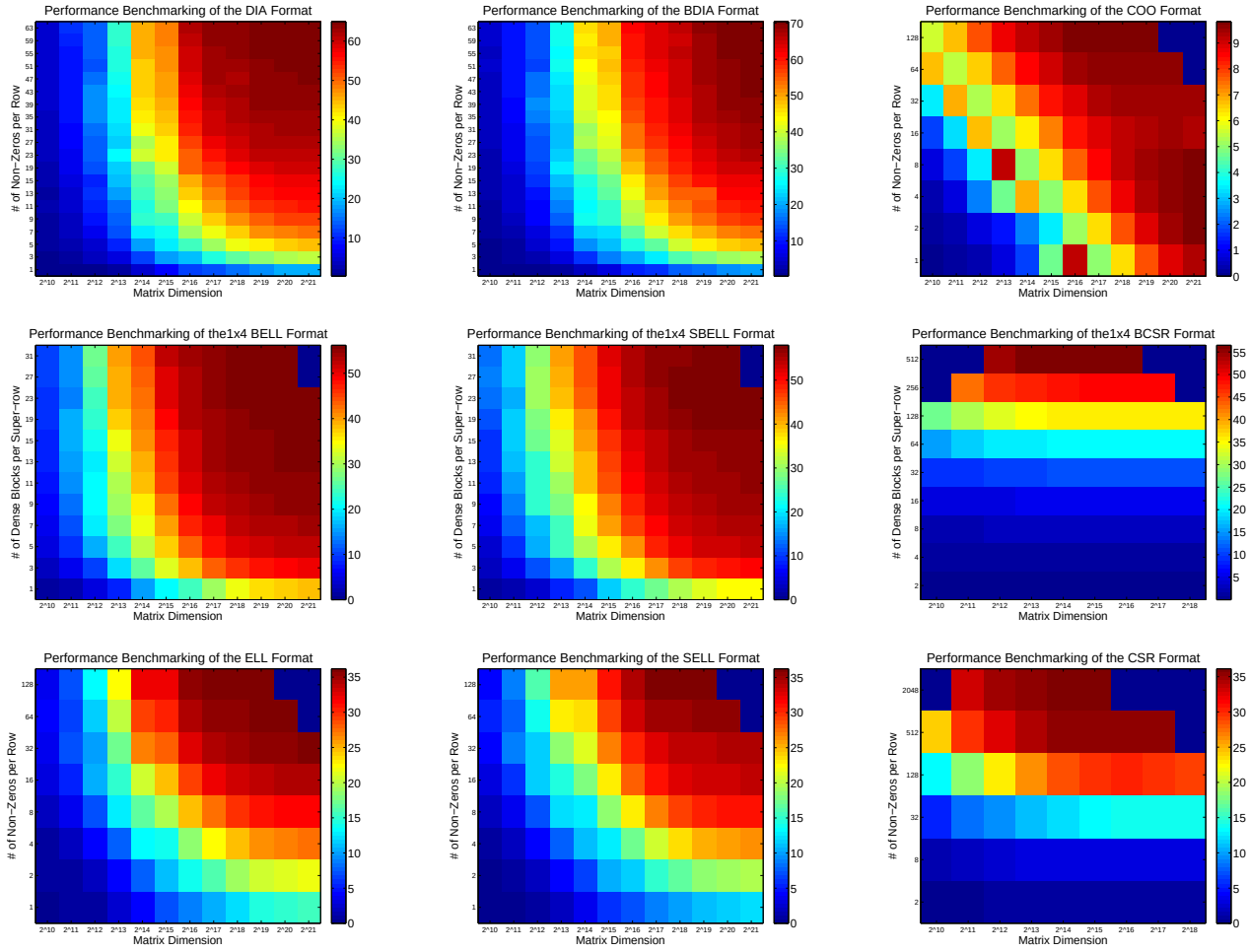

Figure 6.11: Performance benchmarking on the Nvidia GTX 480 platform. The x axis is the dimension of the matrix. On rows 1 and 3, the y axis is the number of non-zeros per row. On row 2, the y axis is the number of dense blocks per super-row. The unit of the color-bar is in GFLOPS.

We summarize our performance benchmarking results on the Nvidia GTX 480 platform in Figure 6.11. Some performance numbers at the top-right corners are missing because the matrix storage size is larger than a pre-defined upperbound. Some performance numbers at the top-left corners are missing because the matrix is too dense to be considered a sparse matrix. The performance of the diagonal-based formats is benchmarked on dense diagonal matrices. Although each diagonal format has multiple implementations, the heat-map shows only the best achievable performance among all implementations. As expected, performance increases with an increase in matrix dimension and number of non-zeros per row. The peak performance of the BDIA format is larger than that of the DIA format, but when the

number of non-zeros per row is very small, the DIA format will perform slightly better. The performance of the block-based formats is benchmarked on dense diagonal blocked matrices. Due to space limitations, the performances of only the 1×4 blocked matrices are included in this figure. However, blocked matrices with other dimensions follow a similar pattern. For the BELL and the SBELL formats, each work item works on a super-row, and we can achieve close to peak performance when there are 20 to 30 dense blocks per super-row. However, for the BCSR format, because a warp of work items is responsible for a super-row, we need more than 200 dense blocks per super-row to saturate the processors. Performance of the flat formats is benchmarked on dense diagonal matrices. These performance patterns are very close to their blocked variations, but their peak achievable performances are significantly reduced. The COO performance is very stable when the dimension of the matrix is large enough; however, the peak achievable performance is the lowest among the nine formats.

Table 6.6: *clSpMV* performance on Nvidia GTX 480 for the selected 20 matrices, compared to implementations in [14] and to all the single formats supported by *clSpMV*. The highest achieved performance for each matrix is in bold.

Benchmark Name	NV CUDA			Single All		*clSpMV*	
	NV HYB (GFLOPS)	Best NV (GFLOPS)	Best NV Format	Best Single (GFLOPS)	Best Single Format	*clSpMV* (GFLOPS)	*clSpMV* Format
Dense	8.38	32.63	CSR	**54.08**	BCSR	53.05	BCSR
Protein	15	23.17	CSR	35.84	SBELL	**35.86**	SBELL
FEM/Spheres	25.11	25.11	HYB	34.44	SBELL	**34.52**	SBELL
FEM/Cantilever	19.06	34.9	DIA	35.03	SBELL	**35.10**	SBELL
Wind Tunnel	25.07	25.07	HYB	**42.94**	SBELL	**42.94**	SBELL
FEM/Harbor	11.67	13.83	CSR	27.17	SBELL	**27.21**	SBELL
QCD	25.05	25.09	ELL	**30.93**	SELL	29.88	ELL
FEM/Ship	19.11	19.11	HYB	40.59	SBELL	**40.73**	SBELL
Economics	7.61	7.61	HYB	7.32	SELL	**10.59**	ELL(81%)COO(19%)
Epidemiology	24.02	24.02	ELL	25.36	SELL	**26.55**	ELL
FEM/Accelerator	9.35	9.35	HYB	**16.29**	SBELL	15.25	SELL
Circuit	7.35	7.35	HYB	7.72	SELL	**11.40**	ELL(84%)COO(16%)
Webbase	9.74	9.74	HYB	7.30	COO	**12.77**	ELL(64%)COO(36%)
LP	8.89	12.78	CSR	**12.99**	BCSR	12.98	BCSR
circuit5M	12.81	12.81	HYB	9.02	COO	**17.07**	DIA(9%)SELL(73%)COO(18%)
eu-2005	12.14	12.14	HYB	11.84	SBELL	**16.03**	SELL(85%)COO(15%)
Ga41As41H72	13.28	16.11	CSR	16.11	CSR	**16.80**	BDIA(18%)ELL(32%)CSR(50%)
in-2004	10.53	10.53	HYB	12.04	SBELL	**16.87**	SELL(79%)COO(21%)
mip1	10.8	18.92	CSR	18.92	CSR	**19.00**	SBELL(80%)SELL(17%)COO(3%)
Si41Ge41H72	12	17.68	CSR	17.68	CSR	**18.77**	BDIA(15%)ELL(27%)CSR(58%)

To evaluate the performance of *clSpMV*, we compare its performance to other implementations on the 20 benchmarking matrices. We first compare the performance to [14]. The released code of [14] is based on CUDA, and has SpMV kernels of the DIA, ELL, CSR, COO, and HYB formats. The HYB (Hybrid) format in [14] is composed of the ELL and COO formats; therefore, the HYB format is a subset of our *Cocktail Format*. Although we would like to compare the performance of *clSpMV* to the SELL format by Monakov et al. [92] and to the blocked formats by Choi et al. [27], they did not release their code. So, we instead compare *clSpMV* to our own OpenCL implementations of the SELL, BELL, SBELL, and BCSR formats.

Our experimental results are summarized in Table 6.6. Performance is measured by

Table 6.7: Improvement of *clSpMV* compared to the hybrid format in [14], to the best implementations in [14], and to the best single-format implementations supported by *clSpMV*.

Benchmark Name	*clSpMV* Improvement		
	NV HYB	Best NV	Best Single
Dense	533.1%	62.6%	-1.9%
Protein	139.1%	54.8%	0.1%
FEM/Spheres	37.5%	37.5%	0.2%
FEM/Cantilever	84.2%	0.6%	0.2%
Wind Tunnel	71.3%	71.3%	0.0%
FEM/Harbor	133.1%	96.7%	0.1%
QCD	19.3%	19.1%	-3.4%
FEM/Ship	113.1%	113.1%	0.3%
Economics	39.2%	39.2%	44.7%
Epidemiology	10.5%	10.5%	4.7%
FEM/Accelerator	63.1%	63.1%	-6.4%
Circuit	55.1%	55.1%	47.6%
Webbase	31.1%	31.1%	74.9%
LP	46.0%	1.5%	-0.1%
circuit5M	33.3%	33.3%	89.2%
eu-2005	32.1%	32.1%	35.5%
Ga41As41H72	26.5%	4.3%	4.3%
in-2004	60.2%	60.2%	40.1%
mip1	75.9%	0.4%	0.4%
Si41Ge41H72	56.4%	6.2%	6.2%
Average	83.0%	39.6%	16.8%

$\frac{2 \times nnz}{ExecutionTime}$ (GFLOPS). Sometimes using texture memory to cache the multiplied vector results in higher performance, sometimes not. We evaluate both cases and report only the highest number in the table. The NV HYB (Nvidia Hybrid) column lists the performance achieved by the CUDA code of the HYB format by Bell and Garland [14]. The Best NV (Best Nvidia) column lists the highest performance achieved among the five implementations supported in [14] – DIA, ELL, CSR, COO, and HYB. The Best NV Format column lists the format that achieves the highest performance. The Best Single column lists the highest performance achieved among all single formats. Among all single-format performances, DIA, ELL, CSR, and COO performance is measured using the CUDA implementations from [14]; BDIA, SELL, BELL, SBELL, and BCSR performance is measured using our own OpenCL implementations. Because we have multiple implementations of every single format introduced in Section 6.3.2 (such as different block sizes of the block-based formats), only the highest performance numbers among all implementations are reported. The Best Single Format column summarizes the format that achieves the highest performance. The *clSpMV* column lists the performance achieved by the *clSpMV* autotuner. The *clSpMV* Format column lists the decision made by *clSpMV*. The percentage numbers following the formats refer to the portions of non-zeros covered by the formats.

Table 6.7 summarizes the improvement ratios of *clSpMV* compared to other implementations based on the performance numbers in Table 6.6. On average, *clSpMV* is 83% faster than the CUDA implementation of the proposed HYB format in [14], 39.6% faster than all CUDA implementations in [14], and 16.8% faster than all single formats supported by *clSpMV*.

For the 14 matrices from Williams et al. [123], most have regular structures, and their total numbers of non-zeros are small. Therefore, they favor single-format representation. As

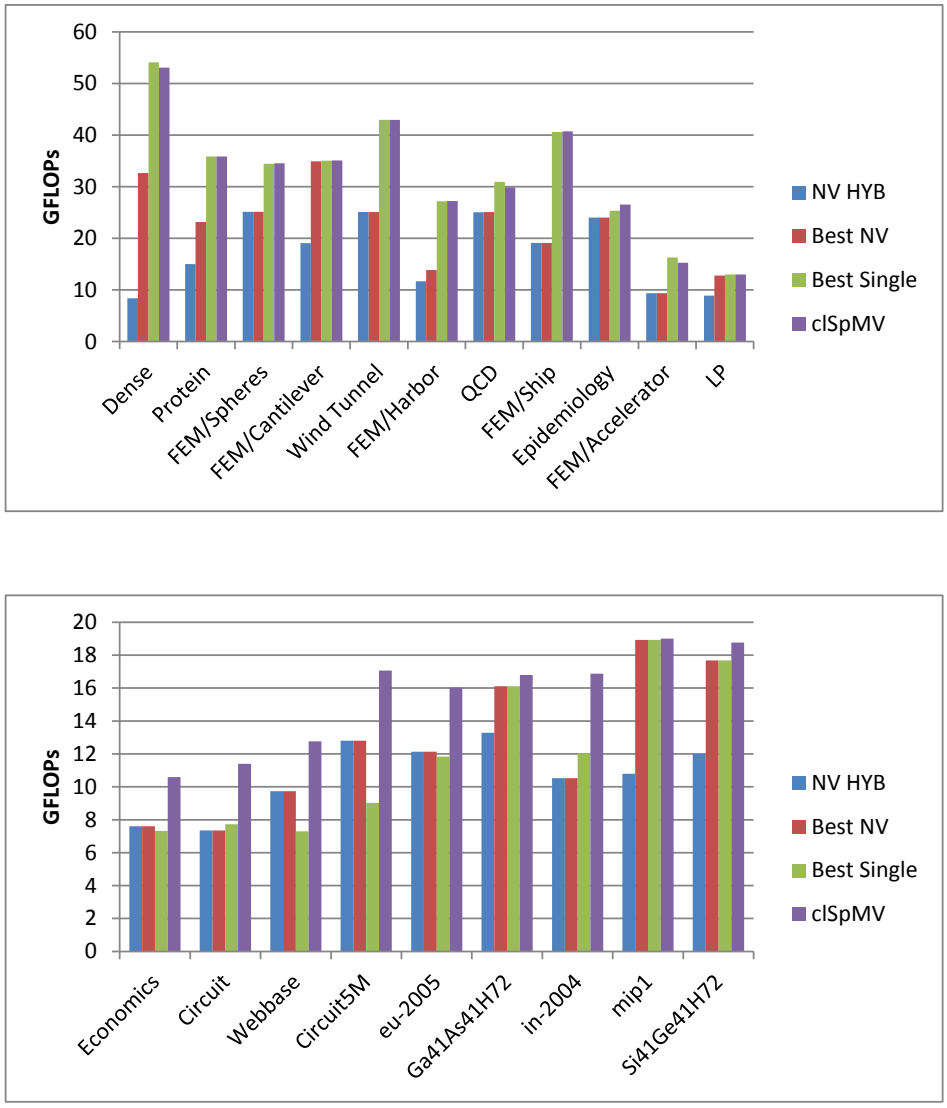

Figure 6.12: *clSpMV* performance on the 20 benchmarking matrices on GTX 480. Top: regular matrices that *clSpMV* suggests representing with one format. Bottom: irregular matrices that *clSpMV* suggests partitioning into multiple submatrices.

shown in Table 6.6 and Figure 6.12, most of the time *clSpMV* can successfully find the best single representation to match the results in the Best Single Format column. Even when the chosen format is not the same, the performance difference is very small. There are three matrices for which the *clSpMV* matches the NV HYB format in [14]. For these, *clSpMV* outperforms the CUDA implementation of the NV HYB format due to three factors. First, the NV HYB format in [14] assumes that the ELL format is three times faster than COO format. In contrast, *clSpMV* uses the more accurate offline benchmarking numbers. Second,

the COO implementation from [14] requires three kernel launches, but *clSpMV* requires only two. Third, the number of work groups (or thread blocks) used by the COO implementation from [14] is fixed; however, *clSpMV* selects the best work group size based on the offline benchmarking information.

For the six additional matrices from the University of Florida Sparse Matrix Collection [37], *clSpMV* partitions them into many submatrices. *clSpMV* achieves significant improvements on three matrices ($40\% - 90\%$ better performance), but small improvements on the other three ($0.4\% - 6\%$). This is because of texture memory. Texture memory boosts CSR performance from 10 GFLOPS to $16-18$ GFLOPS; therefore, the data access pattern of CSR has very high hit rate on the texture cache. Though CSR performance is good, *clSpMV* achieves even better performance. Theoretically, the *Cocktail Format* should outperform every single format. In practice, *clSpMV* uses good policies to find reasonable matrix partitions, represents them using the *Cocktail Format*, and does achieve better performance compared to all other single formats.

We can also discuss the results on regular matrices and irregular matrices separately. A matrix is considered regular if *clSpMV* suggests using one sparse matrix format to represent the entire matrix. A matrix is considered irregular if *clSpMV* suggests partitioning the matrix into many submatrices and then using different formats for them. The result is shown in Figure 6.12. The plot on the top shows performance on 11 regular matrices. On these matrices, the performance of *clSpMV* is 114% better than the NV HYB format in [14], 48% better than the best formats in [14], and 0.5% worse than the best single format. The plot on the bottom shows performance on the other nine irregular matrices. On these, the performance of *clSpMV* is 46% better than the NV HYB format in [14], 29% better than the best formats in [14], and 38% better than the best single format. As a result, if a given matrix is regular, *clSpMV* finds a single format that properly represents the matrix, and thus achieves comparable performance to the best single format. If a given matrix is irregular, *clSpMV* determines the proper *Cocktail Format* to represent the matrix, and outperforms all existing SpMV works.

Experimental Results on AMD Radeon 6970

The experimental settings on the AMD platform are very similar to those on the Nvidia. The performance benchmarking results are summarized in Figure 6.13. Additional performance numbers at the top-right corners are missing because the required matrix storage sizes of these sample points exceed 256 MB, the largest consecutive memory size allowed by the AMD OpenCL runtime. We are not aware of any SpMV project that targets AMD platforms, so we compare *clSpMV* to only the single-format implementations supported by *clSpMV*. The results are shown in Table 6.8. On average, the performance of *clSpMV* is 43.3% higher than any of the single format implementations. On the dense matrix and the LP matrix, *clSpMV* chooses the right single format, but the chosen block size is not optimal and so the performance is worse than the best single format. An offline benchmarking procedure with wider and denser sample points would supply better execution time estimates, and therefore enable *clSpMV* to determine the best block size.

Figure 6.14 shows our experimental results by separating the benchmarking matrices into

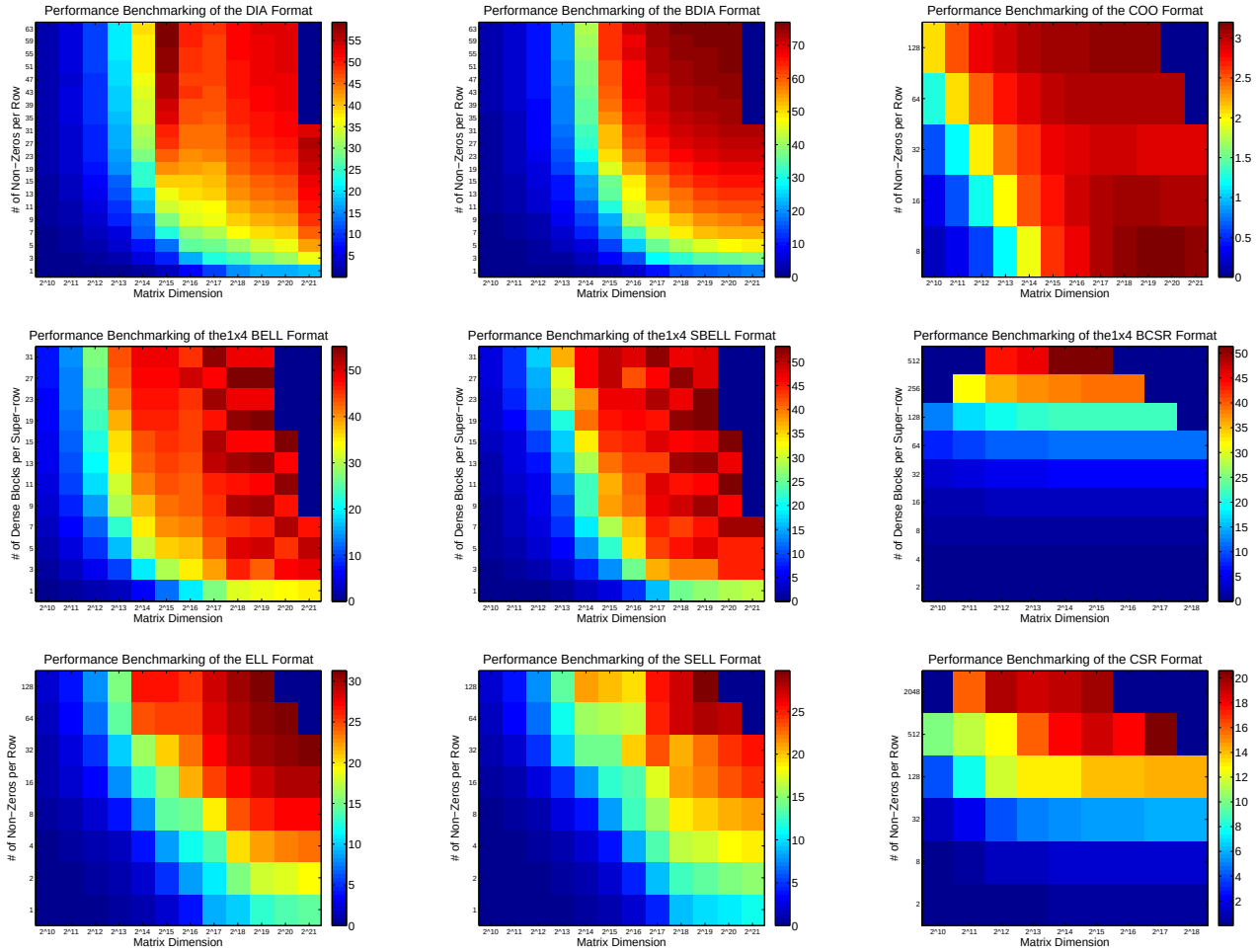

Figure 6.13: Performance benchmarking on the AMD Radeon 6970 platform. The x axis is the dimension of the matrix. On rows 1 and 3, the y axis is the number of non-zeros per row. On row 2, the y axis is the number of dense blocks per super-row. The unit of the color-bar is in GFLOPS.

regular and irregular. The plot on the top shows the performance on nine regular matrices. On these matrices, the performance of *clSpMV* is 2% worse on average than the best single format. The plot on the bottom shows the performance on the other 11 irregular matrices. On these, the performance of *clSpMV* is 80% better than the best single format.

When comparing Tables 6.6 and 6.8 and Figures 6.11, 6.12, 6.13, and 6.14, we see that *clSpMV* makes decisions based on platform strengths. Since the BDIA format achieves significantly higher performance than all other formats on the AMD platform, *clSpMV* favors the BDIA format whenever possible. For example, the Protein and Cantilever matrices are considered to be regular matrices on the GTX 480 platform, and are represented by only the SBELL format. Conversely, they are considered irregular matrices on the Radeon 6970 platform, and are partitioned into one BDIA matrix and one other matrix. Such partition-

Table 6.8: *clSpMV* performance on the selected 20 matrices, compared to all the single formats supported by *clSpMV* on AMD Radeon 6970. The highest achieved performance for each matrix is in bold.

Benchmark Name	Single All		*clSpMV*		Improvement
	Best Single (GFLOPS)	Best Single Format	*clSpMV* (GFLOPS)	*clSpMV* Format	
Dense	**46.85**	BCSR	41.85	BCSR	-10.7%
Protein	29.91	SBELL	**30.99**	BDIA(43%)SBELL(57%)	3.6%
FEM/Spheres	**31.85**	SBELL	31.44	SBELL	-1.3%
FEM/Cantilever	33.72	DIA	**35.93**	BDIA(90%)ELL(10%)	6.5%
Wind Tunnel	**35.23**	SBELL	34.51	SBELL	-2.0%
FEM/Harbor	**22.29**	SBELL	22.20	SBELL	-0.4%
QCD	24.84	SELL	**25.01**	BELL	0.7%
FEM/Ship	33.75	SBELL	**34.43**	SBELL	2.0%
Economics	4.87	SELL	**9.04**	ELL(88%)COO(12%)	85.9%
Epidemiology	22.51	ELL	**22.58**	ELL	0.3%
FEM/Accelerator	**15.83**	SELL	15.51	SELL	-2.0%
Circuit	3.06	COO	**8.40**	ELL(88%)COO(12%)	174.7%
Webbase	3.26	COO	**6.42**	ELL(70%)COO(30%)	97.0%
LP	**10.03**	BCSR	9.50	BCSR	-5.3%
circuit5M	3.21	COO	**8.06**	SELL(82%)COO(18%)	150.7%
eu-2005	3.01	COO	**8.19**	ELL(83%)COO(17%)	172.1%
Ga41As41H72	4.70	CSR	**6.93**	BDIA(18%)ELL(32%)CSR(50%)	47.5%
in-2004	3.04	COO	**7.42**	SBELL(28%)ELL(53%)COO(19%)	144.2%
mip1	8.27	BCSR	**8.28**	BDIA(20%)SBELL(62%)SELL(14%)COO(4%)	0.2%
Si41Ge41H72	10.81	SBELL	**11.10**	BDIA(15%)SBELL(85%)	2.7%
Average					43.3%

ing actually boosts performance because BDIA is significantly faster than all other formats. Moreover, the ELL performance on the AMD platform is significantly better than the COO performance, so the *clSpMV* increases the ratio of the ELL portion on the AMD platform. Taking the economics matrix as an example, *clSpMV* uses ELL to represent 81% of the non-zeros on the GTX 480 platform, but uses ELL to represent 88% of the non-zeros on the Radeon 6970 platform. These results demonstrate that *clSpMV* can automatically adjust the *Cocktail Format* representation of a given matrix based on the strengths of the platform.

Theoretically, the *Cocktail Format* is a superset over all single sparse matrix formats, so its performance should be better than (or at least equal to) all single formats. In practice, with the help of the *clSpMV* autotuner we have achieved 16.8% better performance than any single format on the Nvidia GTX 480 platform, and 43.3% better performance on the AMD Radeon 6970 platform. Although solutions that are portable across diverse platforms generally provide lower performance compared to solutions specialized to a particular platform, we have nonetheless achieved 83% better performance compared to the CUDA implementation of the proposed HYB format in [14], and 39.6% better performance compared to all CUDA implementations by Bell and Garland [14]. In conclusion, the *Cocktail Format* delivers better SpMV performance both theoretically and practically. *clSpMV* is a cross-platform autotuner that selects the best representation of any given matrices and delivers very high-performance SpMV kernels.

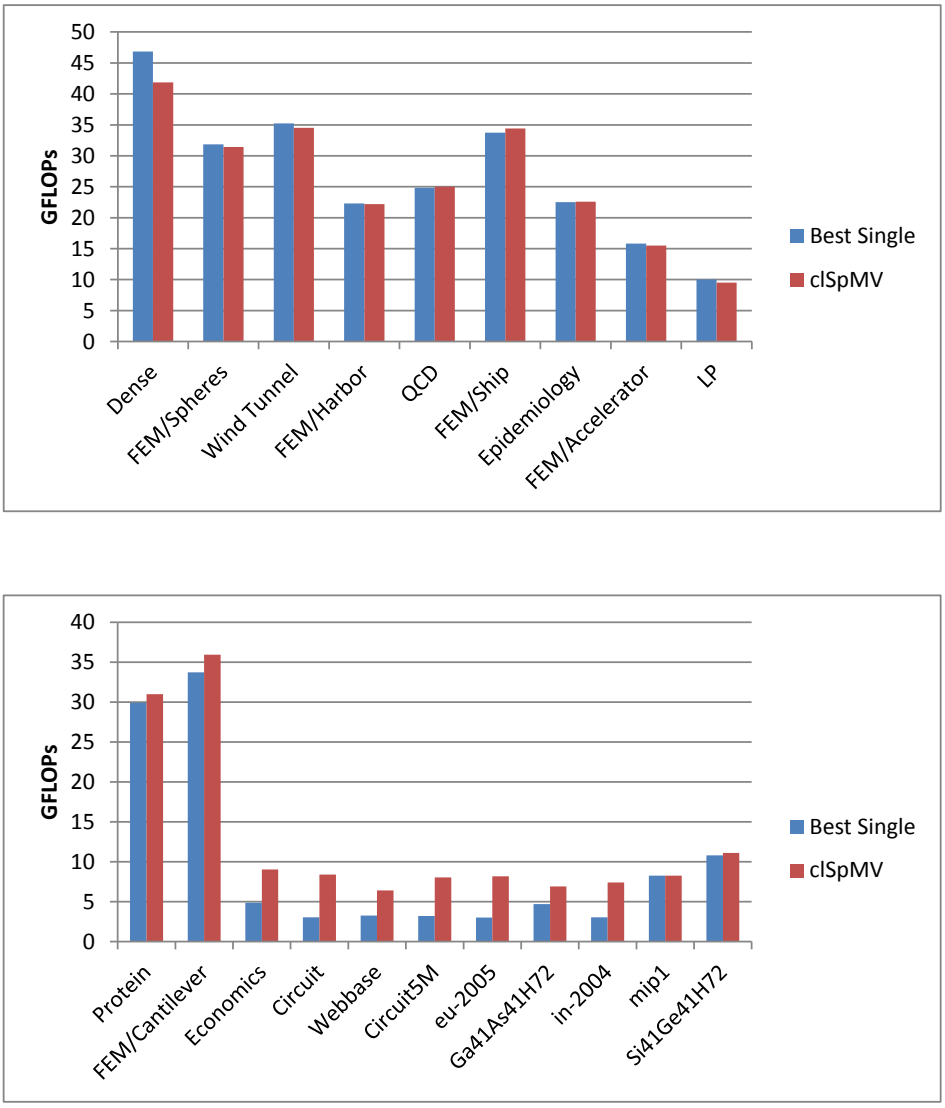

Figure 6.14: *clSpMV* performance on the 20 benchmarking matrices on Radeon 6970. Top: regular matrices that *clSpMV* suggests representing with one format. Bottom: irregular matrices that *clSpMV* suggests partitioning into multiple submatrices.

6.4 The Pair-Wise Distance Computation Autotuner

The `Vector Distance` application pattern in Table 3.1 is widely used in many classification algorithms. It computes the distance between two vectors of the same dimension, and is the basis of the k-nearest neighbor [8] algorithm – given a query vector, find k training samples with the smallest distances to the query vector, and ask these k neighbors to decide the category to which the query vector belongs. Variations of the k-nearest neighbor

algorithm are used in many state-of-the-art object recognition systems [133, 95, 18, 61].

The fundamental idea of the `Vector Distance` pattern is to compute a scalar value from two vectors with the same dimension. As such, this pattern can be generalized to many other classification algorithms. For example, the support vector machine (SVM) classification [31] is computed as follows:

$$c = sgn\{b + \sum_{i=1}^{l} y_i \alpha_i \Phi(s_i, x)\}, \tag{6.3}$$

where x is the query vector, s_i is the ith support vector, $\Phi(s_i, x)$ is the kernel function on s_i and x, α_i is the weight of s_i, y_i is the category of s_i, l is the total number of support vectors, b is the biased term, sgn is the sign function, and c is the category of x. If we define the distance between x and s_i to be $\Phi(s_i, x)$, we can employ the `Vector Distance` application pattern to compute the SVM classification algorithm. The same strategy can apply the `Vector Distance` application pattern to other classification algorithms that employ kernel tricks, such as discriminant analysis [91] and logistic regression [67].

In practice, instead of computing a distance value between two vectors, more often we compute distances between two vector sets all at once. Let $X = x_1, x_2, \ldots, x_n$ be n query vectors. In the k-nearest neighbor algorithm, let $Y = y_1, y_2, \ldots, y_m$ be m training samples, we need to compute the distances between each pair of $x_i \in X$ and $y_j \in Y$. In the SVM algorithm, let $S = s_1, s_2, \ldots, s_m$ be m support vectors, we also need to compute the distances between each pair of $x_i \in X$ and $s_j \in S$. We therefore define the pair-wise distance computation as follows:

- Computation *PaDi*(*Pair-Wise Distance*): Given two sets of vectors with dimension k, $X = \{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathbb{R}^k \ \forall x_i \in X$, and $Y = \{y_1, y_2, \ldots, y_m\}$, $y_j \in \mathbb{R}^k \ \forall y_j \in Y$, and a distance function $dist(x, y)$, compute a distance table D with dimension $n \times m$ such that $D_{ij} = dist(x_i, y_j) \ \forall x_i \in X, \ y_j \in Y$.

The computational complexity of the PaDi computation is $O(nmk)$, where n and m are the sizes of the two vector sets, and k is the dimension of the vectors. If k, m, and n are of the same magnitude, the computational complexity becomes $O(n^3)$. This is a very expensive computation, so we develop the *clPaDi* autotuner to optimize it.

There are many different distance functions, and researchers keep creating new ones. For *clPaDi* to be flexible enough to fit the diverse needs of researchers, we would like it to provide user customizations. By analyzing common distance functions, we have observed that most can be defined by three operators – an element operator, a reduction operator, and a post-processing operator. Given two vectors with dimension k, $u = [a_1, a_2, \ldots, a_k]$, and $v = [b_1, b_2, \ldots, b_k]$, an element operator $e(a, b)$ computes a value between a pair of elements in vectors u and v. By performing the same element operator on the k pairs, we get a vector z with dimension k, $z = [z_1, z_2, \ldots, z_k] = [e(a_1, b_1), e(a_2, b_2), \ldots, e(a_k, b_k)]$. The reduction operator $\bigoplus$ reduces vector z into a scalar value c, $c = z_1 \bigoplus z_2 \bigoplus \ldots \bigoplus z_k$. The post-processing operator $r(c)$ then computes the final distance value d based on c, $d = r(c)$.

Tables 6.9 and 6.10 show how we can define common distance functions and SVM kernels with only these three operators. The two tables cover all distance functions and SVM kernels

Table 6.9: Using element operators, reduction operators, and post-processing operators to define common distance functions.

Distance Name	Distance Computation	$e(a,b)$	$\bigoplus$	$r(c)$
Minkowski Distance	$(\sum_{i=1}^{k} \lvert a_i - b_i \rvert^p)^{1/p}$	$\lvert a-b \rvert^p$	$+$	$c^{1/p}$
Manhattan Distance	$\sum_{i=1}^{k} \lvert a_i - b_i \rvert$	$\lvert a-b \rvert$	$+$	c
Chebyshev Distance	$\max_{i=1}^{k} \lvert a_i - b_i \rvert$	$\lvert a-b \rvert$	max	c
Euclidean Distance	$\sqrt{\sum_{i=1}^{k} (a_i - b_i)^2}$	$(a-b)^2$	$+$	$\sqrt{c}$
Squared Euclidean Distance	$\sum_{i=1}^{k} (a_i - b_i)^2$	$(a-b)^2$	$+$	c
χ^2 Distance	$\sum_{i=1}^{k} \frac{(a_i-b_i)^2}{a_i+b_i}$	$\frac{(a-b)^2}{a+b}$	$+$	c
Canberra Distance	$\sum_{i=1}^{k} \frac{\lvert a_i-b_i \rvert}{\lvert a_i \rvert+\lvert b_i \rvert}$	$\frac{\lvert a-b \rvert}{\lvert a \rvert+\lvert b \rvert}$	$+$	c

Table 6.10: Using element operators, reduction operators, and post-processing operators to define common SVM kernel functions.

Kernel Name	Kernel Computation	$e(a,b)$	$\bigoplus$	$r(c)$
Linear Kernel	$\sum_{i=1}^{k} a_i b_i$	ab	$+$	c
Polynomial Kernel	$(\alpha \sum_{i=1}^{k} a_i b_i + \gamma)^p$	ab	$+$	$(\alpha c + \gamma)^p$
Gaussian Kernel	$\exp(-\gamma \sum_{i=1}^{k} (a_i - b_i)^2)$	$(a-b)^2$	$+$	$exp(-\gamma c)$
Sigmoid Kernel	$\tanh(\alpha \sum_{i=1}^{k} a_i b_i + \gamma)$	ab	$+$	$\tanh(\alpha c + \gamma)$
Intersection Kernel [82]	$\sum_{i=1}^{k} \min(a_i, b_i)$	$\min(a,b)$	$+$	c

employed by the 31 state-of-the-art papers in Section 3.3. By enabling users to customize these three operators, *clPaDi* is flexible enough to allow researchers to use their favorite distance functions.

6.4.1 Exploring the Design Space of Algorithms

The PaDi computation is very similar to the general matrix multiply (GEMM) computation. In fact, if we let $e(a,b) = ab$, $\bigoplus = +$, and $r(c) = c$, PaDi is exactly the same as GEMM. GEMM is a fundamental computation in dense linear algebra, and has been studied for decades. Here, we summarize algorithmic improvements for GEMM, and then explain how we apply these algorithms in *clPaDi*.

The naive GEMM algorithm is summarized in Figure 6.15. We perform a dot product computation between each row in A and each column in B. The total floating point operations (FLOPs) in this computation is $2nmk$. Assuming $n = m = k$, total FLOPs becomes $2n^3$. The memory operations required in GEMM are the summation of the sizes of the three matrices, $n^2 + n^2 + n^2 = 3n^2$. The arithmetic intensity of GEMM is $\frac{2n^3}{3n^2} = \frac{2n}{3}$. In other words, the number of reuses of a value in the memory is proportional to n. When n increases, elements in A, B, and C are reused more often. As a result, GEMM is a compute-bounded computation – that is, the performance of GEMM is bounded by the number of FLOPs a hardware platform can perform in one second, rather than by memory bandwidth.

However, although GEMM is theoretically compute-bounded, the naive algorithm in

```
Algorithm: GEMM
Input:   A (Matrix with dimension n × k)
         B (Matrix with dimension k × m)
Output:  C (Matrix with dimension n × m)
1   for i ← 1 to n
2       for j ← 1 to m
3           for s ← 1 to k
4               C[i][j] ← C[i][j] + A[i][s]B[s][j];
```

Figure 6.15: The naive GEMM algorithm.

Figure 6.15 is nonetheless still memory-bounded most of the time. If matrices A, B, and C do not fit in the register file of a processor, they must stay in other memory layers. The processor must therefore wait for the memory controller to load these matrices from other memory layers to the register file before it can execute the multiply and add instructions. If the processor spends a meaningful portion of its time waiting for data, performance will be bounded by memory bandwidth.

In order to prevent the GEMM computation from being memory-bounded, we need to reduce the number of memory operations and improve the utilization rate of data in every memory layer. The most effective strategy to achieve this is blocking – we read a submatrix $\hat{A}$ from A, a submatrix $\hat{B}$ from B, and update a submatrix $\hat{C}$ from C, such that $\hat{A}$, $\hat{B}$, and $\hat{C}$ will fit in the register file at the same time. The GEMM computation on $\hat{A}$, $\hat{B}$, and $\hat{C}$ thus do not rely on data from other memory layers, and the processor can achieve peak performance in computing $\hat{C} = \hat{C} + \hat{A}\hat{B}$.

Figure 6.16 shows the blocked GEMM algorithm on a system with two memory layers – a fast-but-small layer, and a slow-but-large layer. Assuming that the slow memory is able to store matrices A, B, and C, and the fast memory is able to store submatrices $\hat{A}$, $\hat{B}$, and $\hat{C}$, the blocked GEMM algorithm continuously partitions the original matrices into smaller submatrices and performs the GEMM computation on these smaller submatrices. The blocked GEMM algorithm partitions matrix A, B, and C into submatrices with dimensions $x \times z$, $z \times y$, and $x \times y$, respectively. Lines 1-2 iteratively go over submatrices $\hat{C}$ from C. Line 3 reads $\hat{C}$ from the slow memory to the fast memory. Line 4 then iteratively goes over submatrices $\hat{A}$ and $\hat{B}$ from A and B. Lines 5-6 read $\hat{A}$ and $\hat{B}$ from the slow memory to the fast memory. Lines 7-10 perform the naive GEMM computation on the submatrices $\hat{C} = \hat{C} + \hat{A}\hat{B}$. Line 11 stores submatrix $\hat{C}$ into the slow memory. Because the GEMM computation of $\hat{C} = \hat{C} + \hat{A}\hat{B}$ happens in the fast memory, the blocked GEMM algorithm is significantly faster than the naive GEMM algorithm, in which the entire computation happens in the slow memory.

This blocking strategy can be hierarchical according to the memory subsystem of the

```
Algorithm: Blocked GEMM
Input:  A (Matrix with dimension n × k)
        B (Matrix with dimension k × m)
Output: C (Matrix with dimension n × m)
1   for b_x ← 1 to n, step x
2       for b_y ← 1 to m, step y
3           Read Ĉ = C[b_x][b_y] ... C[b_x + x − 1][b_y + y − 1] into fast memory;
4           for b_z ← 1 to k, step z
5               Read Â = A[b_x][b_z] ... A[b_x + x − 1][b_z + z − 1] into fast memory;
6               Read B̂ = B[b_z][b_y] ... B[b_z + z − 1][b_y + y − 1] into fast memory;
7               for i ← b_x to b_x + x − 1
8                   for j ← b_y to b_y + y − 1
9                       for s ← b_z to b_z + z − 1
10                          C[i][j] ← C[i][j] + A[i][s]B[s][j];
11          Store Ĉ into slow memory;
```

Figure 6.16: The blocked GEMM algorithm.

underlying hardware platforms. For example, Intel Core i7 2600 has four cores, and its memory subsystem has four layers – an 8 MB L3 cache shared by all four cores, and a 256 KB L2 cache, a 32 KB data cache, and a 576 bytes floating point register file for each core. Based on these four memory layers, we can perform four hierarchies of blocking to improve the data utilization rate of each memory layer: 1) find submatrices $\hat{A}_1$, $\hat{B}_1$, $\hat{C}_1$ from A, B, C such that $\hat{A}_1$, $\hat{B}_1$, $\hat{C}_1$ can be stored in L3; 2) find submatrices $\hat{A}_2$, $\hat{B}_2$, $\hat{C}_2$ from $\hat{A}_1$, $\hat{B}_1$, $\hat{C}_1$ such that $\hat{A}_2$, $\hat{B}_2$, $\hat{C}_2$ can be stored in L2; 3) find submatrices $\hat{A}_3$, $\hat{B}_3$, $\hat{C}_3$ from $\hat{A}_2$, $\hat{B}_2$, $\hat{C}_2$ such that $\hat{A}_3$, $\hat{B}_3$, $\hat{C}_3$ can be stored in L1; and 4) find submatrices $\hat{A}_4$, $\hat{B}_4$, $\hat{C}_4$ from $\hat{A}_3$, $\hat{B}_3$, $\hat{C}_3$ such that $\hat{A}_4$, $\hat{B}_4$, $\hat{C}_4$ can be stored in the register file.

For the blocked GEMM algorithm in Figure 6.16, every element in $\hat{A}$, $\hat{B}$, and $\hat{C}$ is reused y, x, and k times in the fast memory, respectively. The larger the blocks, the more reuse of data in the fast memory. However, if the block sizes are too large, the slow memory might not be able to store all the blocks. Moreover, different hardware platforms have different memory subsystems. In order to optimize the GEMM computation on a specific platform, we must consider different block sizes and determine which works best on the memory subsystem. A number of autotuners have been developed to tune the block sizes on different platforms. For example, ATLAS [122] and PHiPAC [16] tune the block size on single-core platforms. GotoBLAS [56] tunes the block size on multicore platforms. Volkov and Demmel optimized GEMM on Nvidia Tesla architectures by blocking matrix B in shared local memory and blocking matrix C in registers [119]. Li et al. developed an autotuner to find the best blocking sizes on Nvidia Fermi architectures [79]. Du et al. proposed using autotuning to optimize GEMM on both Nvidia and AMD architectures using OpenCL [43].

Algorithm: Blocked PaDi
Input: A (Vector set with dimension $n \times k$)
B (Vector set with dimension $m \times k$)
Output: D (Distance Table with dimension $n \times m$)
1 **for** $b_x \leftarrow 1$ to n, step x
2 **for** $b_y \leftarrow 1$ to m, step y
3 Initialize $\hat{D}$ to be a zero matrix in registers;
4 **for** $b_z \leftarrow 1$ to k, step z
5 Read $\hat{A}_1 = \mathrm{A}[b_x][b_z] \ldots \mathrm{A}[b_x + x - 1][b_z + z - 1]$ into shared local memory;
6 Read $\hat{B}_1 = \mathrm{B}[b_y][b_z] \ldots \mathrm{B}[b_y + y - 1][b_z + z - 1]$ into shared local memory;
7 **for** $s \leftarrow 1$ to z
8 Read $\hat{A}_2 = \hat{A}_1[1][s] \ldots \hat{A}_1[x][s]$ into registers;
9 Read $\hat{B}_2 = \hat{B}_1[1][s] \ldots \hat{B}_1[y][s]$ into registers;
10 **for** $i \leftarrow 1$ to x
11 **for** $j \leftarrow 1$ to y
12 $\hat{D}[\mathrm{i}][\mathrm{j}] \leftarrow \hat{D}[\mathrm{i}][\mathrm{j}] \bigoplus \mathrm{e}(\hat{A}_2[\mathrm{i}], \hat{B}_2[\mathrm{j}])$;
13 $\hat{D} \leftarrow r(\hat{D})$;
14 $\mathrm{D}[b_x][b_y] \ldots \mathrm{D}[b_x + x - 1][b_y + y - 1] \leftarrow \hat{D}$;

Figure 6.17: The blocked PaDi algorithm.

Because the OLOV project targets GPU platforms, we use a strategy similar to that of Li et al. [79] to perform two hierarchies of blocking in the PaDi computation – shared memory blocking, and register blocking. The blocked PaDi algorithm is summarized in Figure 6.17. Input matrices A and B are the two vector sets, and output matrix D is the table that stores the distances between each vector pair of A and B. The dimension of vectors is k. Vector sets A and B have n and m vectors, respectively. In lines 1-6, we store $\hat{A}_1$ (a block of A with dimension $x \times z$) and $\hat{B}_1$ (a block of B with dimension $y \times z$) into shared local memory. This is the blocking in shared local memory. In lines 7-9, we iteratively read a column of $\hat{A}_1$ and $\hat{B}_1$ from shared local memory to registers. So, the dimensions of $\hat{A}_2$ and $\hat{B}_2$ are $x \times 1$ and $y \times 1$. This is the blocking in the registers. In lines 10-12, we update $\hat{D}$ (a block of D with dimension $x \times y$) in registers using the element and reduction operators. In lines 13-14, we apply the post-processing operator on $\hat{D}$ and store it into output distance table D. This is the algorithm used in *clPaDi*.

The *clPaDi* Autotuner

After a user specifies the three operators – the element, reduction, and post-processing operators, *clPaDi* automatically tunes and optimizes the PaDi computation with the distance function defined by the user. Similar to the *clSpMV* autotuner, *clPaDi* is composed

of two stages: the offline benchmarking stage, and the online decision-making stage. The offline benchmarking stage samples the performance of different blocking sizes on different vector set configurations. In the online decision-making stage *clPaDi* chooses the blocking size that delivers the best performance given the input configurations based on offline benchmarking data.

The configurations of vector sets involve three parameters: n, m, and k. n is the number of vectors in the first vector set, m is the number of vectors in the second vector set, and k is the dimension of the vectors.

The offline benchmarking stage samples the performance of different blocking sizes on different n, m, and k values. Sampling density is a trade-off between performance prediction accuracy and benchmarking time – collecting more samples in the configuration space produces more accurate performance predictions, but also takes more time. *clPaDi* samples the three parameters in an exponential scale – n from 2^9 to 2^{14}, m from 2^9 to 2^{14}, and k from 2^7 to 2^{13}. The offline benchmarking stage takes about half a day.

In the online decision-making stage, *clPaDi* computes the distance table between two vector sets. Based on the input configuration (the n, m, and k parameters of the input vector sets), *clPaDi* estimates the execution time of different blocking sizes, selects the optimal blocking size, and performs the PaDi computation with this optimal blocking size. The execution times of the different blocking sizes are estimated by interpolating between nearby samples in the offline benchmarking data. The overhead for determining the best blocking size is only 3.3 microseconds on an Intel Core i7 920 machine, and therefore negligible compared to the PaDi computation.

6.4.2 Exploring the Design Space of Parallelization Strategies

In Figure 6.17, we block distance table D with submatrices $\hat{D}$ of dimension $x \times y$. Let $d_x = n/x$, $d_y = m/y$, D is partitioned into $d_x \times d_y$ submatrices. Because the submatrices are independent, we can compute each in parallel. We create two-dimensional work groups wg$[d_x][d_y]$ in OpenCL, and assign each work group to compute a submatrix of D. That is, work group wg[i][j] computes the submatrix of D from D$[(i-1)x+1][(j-1)y+1]$ to D$[ix][jy]$.

We create 16×16 work items in each group. Therefore, each work item is responsible for computing $\frac{x}{16} \times \frac{y}{16}$ elements in the $x \times y$ submatrix. To ensure that every work item receives the same amount of work, we force x and y be multiples of 16. For example, if $(x, y) = (16, 16)$, every work item computes an element in the submatrix; if $(x, y) = (64, 80)$, every work item computes 4×5 elements in the submatrix.

Assuming that each work item is responsible for $w_x \times w_y$ elements in the submatrix, two different parallelization strategies can be applied for assigning work to the work items:

1. Contiguous Work Assignment: Every work item is assigned a contiguous block with dimension $w_x \times w_y$ from submatrix $\hat{D}$. The work item with index (i, j) computes the block from $\hat{D}[(i-1)w_x+1][(j-1)w_y+1]$ to $\hat{D}[iw_x][jw_y]$.

2. Interleaved Work Assignment: Every contiguous 16×16 block in submatrix $\hat{D}$ is

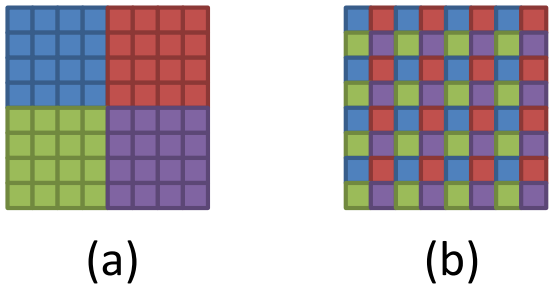

Figure 6.18: Two strategies for the work assignment of each work item: (a) Contiguous work assignment, and (b) Interleaved work assignment.

computed by 16×16 different work items, and every work item is assigned an element in each 16×16 block. The work item with index (i, j) computes $\hat{D}[i + 16a][j + 16b]$, $\forall 0 \leq a < \frac{x}{16}$, $\forall 0 \leq b < \frac{y}{16}$.

Figure 6.18 shows a simplified example of the two strategies, where each work group has 2×2 work items instead of 16×16 work items. For a submatrix $\hat{D}$ with dimension 8×8, every work item must compute 4×4 elements in the submatrix. For the contiguous work assignment strategy, $\hat{D}$ is partitioned into four blocks of dimension 4×4, and each work item works on one contiguous block. For example, the work item with index $(1, 1)$ computes the blue block – $\hat{D}[1][1]$ to $\hat{D}[4][4]$. For the interleaved work assignment strategy, a work item never works on continuous elements – every contiguous 2×2 block in $\hat{D}$ is computed by four different work items. For example, the work item with index $(1, 1)$ computes the blue elements – $\hat{D}[1 + 2a][1 + 2b]$ with $a = \{0, 1, 2, 3\}$ and $b = \{0, 1, 2, 3\}$.

On GPU platforms, multiple work items are grouped and scheduled together. For Nvidia platforms, a warp of 32 work items are scheduled together; for AMD platforms, a wavefront of 64 work items are scheduled together. If we apply the contiguous work assignment strategy, the scheduled work items will access disjoint data in the memory, resulting in redundant memory transactions. However, if we instead apply the interleaved work assignment strategy, consecutive work items access consecutive memory pieces and the number of memory transactions is minimized. As such, the second strategy is better than the first, and so we apply the second strategy in *clPaDi*.

6.4.3 Exploring the Design Space of Platform Parameters

There are three platform parameters we must tune in *clPaDi* – x, y, and z. These parameters describe the blocking sizes of A, B, and D, which are partitioned into $x \times z$, $y \times z$, and $x \times y$ blocks, respectively. As discussed in Section 6.4.2, we create 16×16 work items in each work group, and force x and y to be multiples of 16. This decision reduces parameter space.

The blocked PaDi algorithm in Figure 6.17 iteratively loads an $x \times z$ block from A and a $y \times z$ block from B to the shared local memory. The dimension of A is $n \times k$ – that is, n vectors with dimension k, and the elements in a vector are stored consecutively. Because every matrix is stored in linear memory in practice, the layout of A is $[a_{11}, a_{12}, \ldots, a_{1k}; a_{21}, a_{22}, \ldots, a_{2k}; \ldots, a_{n1}, a_{n2}, \ldots, a_{nk}]$, where a_{11} to a_{1k} is the first vector, a_{21} to a_{2k} the second vector, and a_{n1} to a_{nk} the nth vector. The layout of B is similar. On GPU platforms, we need to read consecutive data together in order to reduce the number of necessary memory transactions. On Nvidia platforms, a warp of 32 work items are scheduled together, but are executed in two clock cycles. So 16 work items are actually executed simultaneously. Similarly, on AMD platforms a wavefront of 64 work items are scheduled together, but are executed in four clock cycles – again, 16 work items are executed simultaneously. Based on this observation, we set z to be 16. In other words, when 16 work items are active, they read 16 consecutive elements from a vector with one memory transaction. If z is smaller than 16 we must still load the entire memory segment, so some memory bandwidth is wasted. If z is larger than 16, we might need to reduce the value of x and y to ensure that $\hat{A}$ and $\hat{B}$ can be stored in the shared local memory. However, reducing x and y deteriorates the effectiveness of register blocking because the size of register blocking is $x \times 1$ and $y \times 1$ as shown in Figure 6.17. As a result, we conclude that 16 is the best value of z.

In *clPaDi*, we explore the design space of $x = \{16, 32, 48, 64, 80, 96\}$ and $y = \{16, 32, 48, 64, 80, 96\}$. So, we implement $6 \times 6 = 36$ kernels with different blocking sizes.

6.4.4 Experimental Results

Here, we evaluate the performance of *clPaDi* on the Nvidia GTX 480 and AMD Radeon 6970 platforms. Because both platforms achieve the highest performance on single precision floating point data type, we perform experiments on vector sets with this data type.

Offline Benchmarking Performance

In the offline benchmarking stage, *clPaDi* collects the performance of different blocking sizes with different vector set configurations. As discussed in Section 6.4.1, the input can be characterized by parameters n, m, and k: n is the number of vectors in the first vector set, m is the number of vectors in the second vector set, and k is the dimension of the vectors. In the offline benchmarking stage, *clPaDi* samples n from 2^9 to 2^{14}, m from 2^9 to 2^{14}, and k from 2^7 to 2^{13}. Given a configuration (n, m, k), *clPaDi* collects the performance of different blocking sizes on that configuration. Since *clPaDi* supports blocking sizes from 16×16 to 96×96, it collects 36 performance results for each configuration.

Figure 6.19 shows heat maps of the 36 performance results with the configuration of $(n, m, k) = (1024, 1024, 128)$ on the Nvidia GTX 480 platform. The heat map on the left shows performance results when the distance function is defined as the dot product between two vectors – that is, this heat map shows the single precision GEMM performance. The heat map on the right instead shows the performance results when the distance function is defined as the χ^2 distance. The χ^2 distance is used in the region-based object recognition system by Gu et al. [61]. In the left heat map, the optimal performance is 561 giga floating

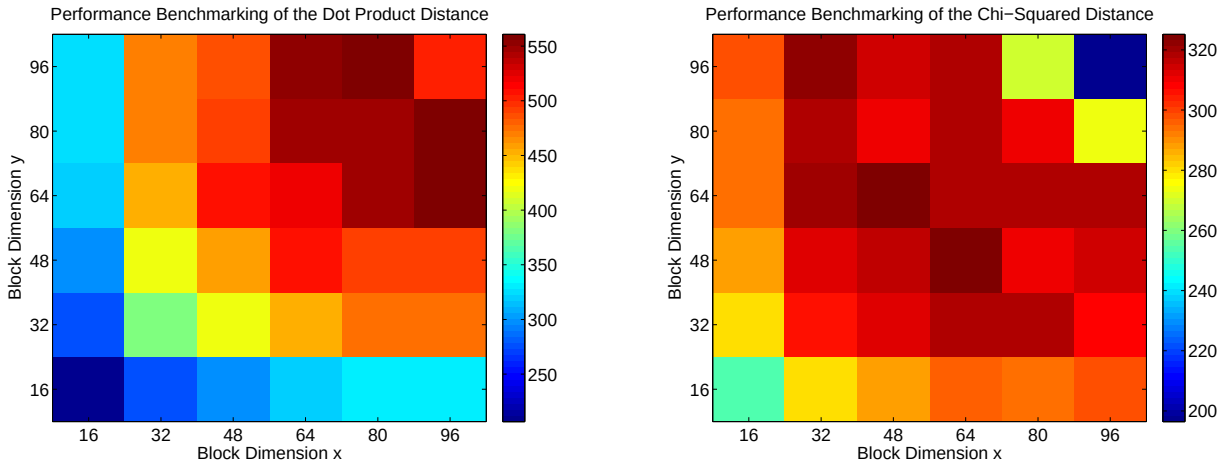

Figure 6.19: Performance benchmarking on the Nvidia GTX 480 platform. Left: Performance benchmarking on dot product distance. Right: Performance benchmarking on χ^2 distance.

point operations per second (GFLOPS), when the blocking size is 96×80. In the right heat map, the optimal performance is 325 GFLOPS, when the blocking size is 48×64. As shown in the maps, when the blocking size is too small, there is little benefit from the blocked algorithm; conversely, if the blocking size is too large, the registers cannot store all data and a portion must be stored in slower memory layers, which degrades performance. Without autotuning, it is very difficult to find the optimal blocking size.

When the distance function is defined as dot product, we need only multiply two elements from two vectors and add the results to a summation variable. Both Nvidia and AMD platforms have a MAD instruction that fuses multiply and add operations. So, the computation requires one instruction and no additional registers to store any intermediate results. On the other hand, when the χ^2 distance function is used, we need to compute $\frac{(a-b)^2}{(a+b)}$ on two elements from two vectors. At least five instructions are required – add $(a+b)$, subtract $(a-b)$, multiply $(a-b) \times (a-b)$, divide $(a-b)^2/(a+b)$, and add (add the results to a summation variable). This requires many additional registers to store the intermediate results, such that the total number of available registers for blocking is reduced. Therefore, the optimal blocking size for the χ^2 distance is smaller than that of the dot product distance. Further, because the division instruction is more expensive than other instructions, the computation time for the χ^2 distance is also longer than for the dot product distance. As a result, the optimal performance of the χ^2 distance is smaller than that of the dot product distance.

Figure 6.20 shows heat maps of the 36 performance results with the same configuration on the AMD Radeon 6970 platform. In the left heat map, the optimal performance is 388 GFLOPS, when the blocking size is 48×96. In the right heat map, the optimal performance is 369 GFLOPS, when the blocking size is 64×64. The difference between

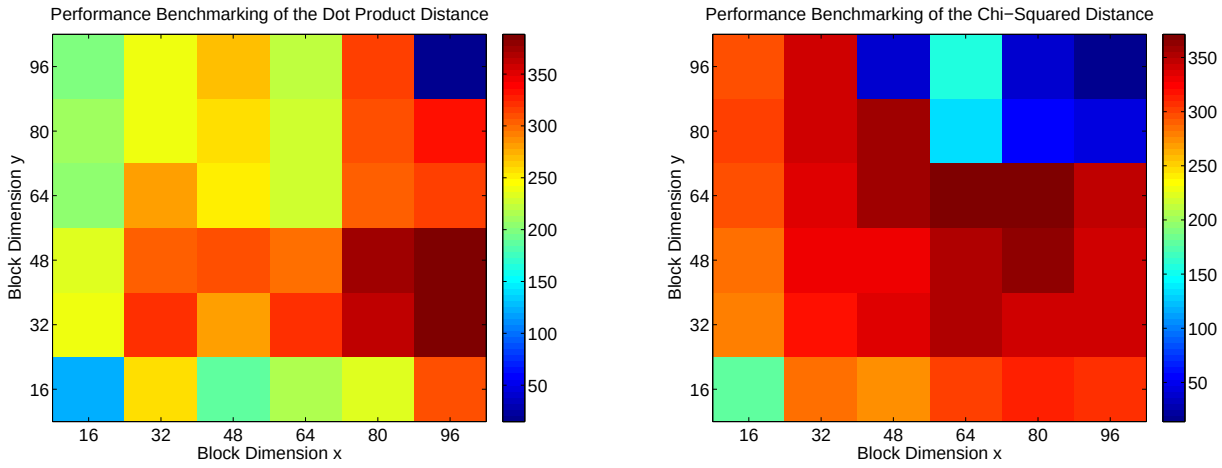

Figure 6.20: Performance benchmarking on the AMD Radeon 6970 platform. Left: Performance benchmarking on dot product distance. Right: Performance benchmarking on χ^2 distance.

the two optimal blocking sizes is small, and the performance difference is also small. This is because the compiler performs different optimizations for the two distance functions. When the blocking size is 64×64, each work item stores four elements from both vector set A and B, and uses these eight values to update a 4×4 block in distance table D. On AMD Radeon 6970, one work item is mapped to a VLIW unit with width four – in other words, one work item can issue a VLIW instruction on four floating point numbers. The χ^2 distance is more complicated, so more operations are performed on the input data. As a result, the compiler successfully identifies the opportunity to issue width-four VLIW instructions for some operations. Conversely, when the distance function is dot product, only one instruction is needed, and the compiler does not convert that operation into a VLIW instruction. Therefore, the performance difference between the two distance functions is not very significant.

Online Decision-Making Performance

When the distance function is dot product between two vectors, the PaDi computation is the same as the GEMM computation. Both Nvidia and AMD release vendor-optimized code for the GEMM computation. CUBLAS [97] is the Nvidia optimized linear algebra library, while clAmdBLAS [4] is the optimized library from AMD. Both libraries offer a highly-optimized GEMM routine. Here, we compare the performance of *clPaDi* to these two libraries in order to understand the effectiveness of the autotuning procedure. Moreover, in Chapter 7, since we use *clPaDi* to optimize the χ^2 distance computation in the region-based object recognition system by Gu et al. [61], we also evaluate the performance when the distance function is set to be the χ^2 distance. We perform these experiments on vector sets with different sizes. Let the two vector sets be A and B, we set the number of vectors in

both sets to be the same, ranging from 512 to 8192. The dimension of the vectors is set to 128, because many feature vectors have the dimension of 128, including the contour feature by Gu et al. [61] and SIFT [80].

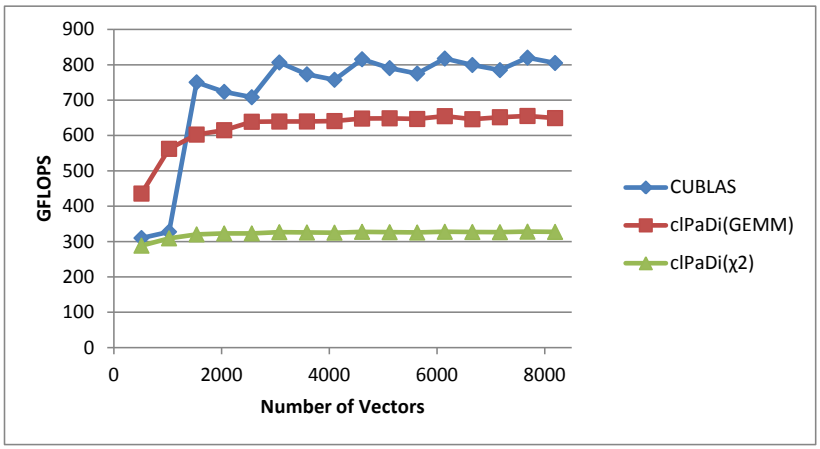

Figure 6.21: Performance of CUBLAS [97], *clPaDi* with dot product distance, and *clPaDi* with χ^2 distance on Nvidia GTX 480.

Figure 6.21 shows the experimental results on an Nvidia GTX 480. CUBLAS is a highly-optimized library, and it achieves 800 GFLOPS on the GEMM computation, while *clPaDi* achieves 650 GFLOPS. This performance gap results from two factors. First, we tune the blocking sizes of only two dimensions. The GEMM autotuner by Li et al. [79] slightly outperforms CUBLAS because they tune the GEMM computation on nine parameters. In addition to blocking sizes, it also tunes the number of threads per thread block, and the reshaping parameters to read submatrices of A and B into shared local memory. Second, we do not use texture memory to cache matrices A and B. As such, the performance of *clPaDi* could be further improved by more detailed autotuning and employing texture memory to read matrices A and B. However, CUBLAS is optimized only when matrices are large; when the matrices are small, *clPaDi* outperforms it. For example, when the number of vectors in A and B are both 1024, *clPaDi* achieves 562 GFLOPS, while CUBLAS achieves only 328 GFLOPS. The performance of *clPaDi* with χ^2 distance is steady at 320 GFLOPS, regardless of matrix size. This is the performance we can expect when applying *clPaDi* to optimize region-based object recognition in Chapter 7.

Figure 6.22 shows the experimental results on AMD Radeon 6970. *clPaDi* outperforms the clAmdBLAS library most of the time. *clPaDi* can achieve 400 to 500 GFLOPS, while clAmdBLAS reaches only 300 GFLOPS. However, clAmdBLAS is a relatively new library, and AMD is still optimizing it. We can expect this library to achieve better performance in future releases. But, for now, we can nonetheless claim that *clPaDi* achieves better performance than the vendor-optimized clAmdBLAS library version 1.7. The *clPaDi* performance with χ^2 distance is similar to the dot product distance, for the same reason as in the offline benchmarking stage – the compiler uses the VLIW unit more effectively with the χ^2 distance

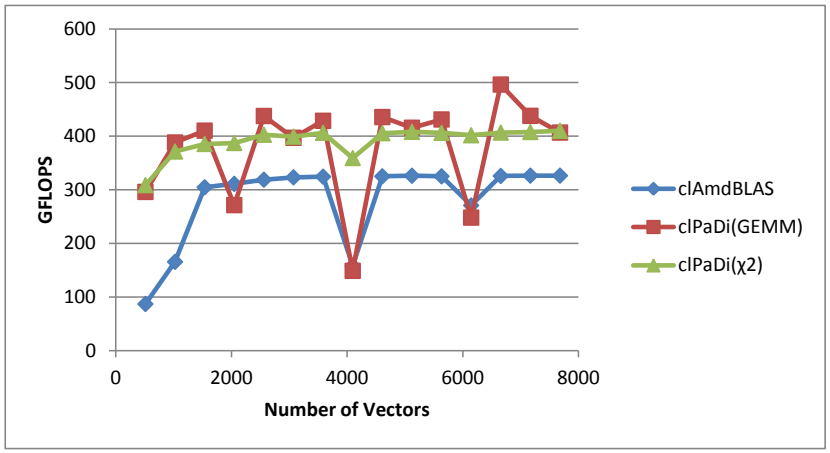

Figure 6.22: Performance of clAmdBLAS [4], *clPaDi* with dot product distance, and *clPaDi* with χ^2 distance on AMD Radeon 6970.

function. If we manually employ vector types in *clPaDi*, such as *float4*, we might be able to further boost performance of the GEMM computation.

Overall, *clPaDi* is a flexible autotuner that allows end users to customize distance functions and optimize the PaDi computation with these customized functions. It achieves near-peak performance on both Nvidia and AMD platforms.

6.5 Summary

We propose the OLOV project to develop a collection of autotuners for computationally-intensive application patterns on GPU platforms using OpenCL. We have developed two autotuners in this project: *clSpMV* and *clPaDi*.

clSpMV is an autotuner for sparse matrix vector multiplication (SpMV) computation. To represent sparse matrices, we propose using the *Cocktail Format* – a collection of sparse matrix formats. A matrix is partitioned into many submatrices, and each submatrix is represented by a specialized format. *clSpMV* is an autotuner that tunes the *Cocktail Format* of a sparse matrix and the implementation of the corresponding SpMV computation. *clSpMV* supports nine sparse matrix formats: DIA, BDIA, ELL, SELL, CSR, COO, BELL, SBELL, and BCSR. It achieves 83% better performance compared to the HYB format in [14], and 39.6% better performance compared to all formats in [14]. Compared to all single formats, *clSpMV* is 16.8% better on the Nvidia GTX 480, and 43.3% better on the AMD Radeon 6970.

clPaDi is an autotuner for the pair-wise distance (PaDi) computation. It allows an end user to customize the distance function between two vectors by specifying three operators – the element, reduction, and post-processing operators. *clPaDi* tunes the performance of the PaDi computation with different blocking sizes. When the distance function is dot

product, the PaDi computation is equivalent to the general matrix multiply GEMM computation. *clPaDi* performs slightly worse than CUBLAS [97], but significantly better than clAmdBLAS [4]. Overall, *clPaDi* offers excellent efficiency, productivity, portability, and flexibility.

clSpMV and *clPaDi* illustrate how effectively autotuners can be used to optimize computationally-intensive application patterns. We plan to cover more application patterns in the future, and in particular expensive optimization patterns.

Chapter 7

Developing Parallel Applications Using the Parallel Application Library

We perform three case studies on three application patterns in Chapter 5, and develop two autotuners for expensive computations in Chapter 6. Although these functions cover only a subset of application patterns in Table 3.1, they are sufficient to resolve bottlenecks in the region-based object recognition system developed by Gu et al. [61]. This chapter demonstrate how we can employ our parallel application library to parallelize and optimize the region-based object recognition system.

7.1 The Region-Based Object Recognition System

Multi-scale scanning algorithms have traditionally been dominant in object recognition research [33, 47, 75, 20, 49, 83]. Although region-based approaches have been used by some researchers [61, 85, 106], their use in the computer vision field is not as prevalent. Identification of objects in images by region may appear similar to how humans identify objects, but this approach relies heavily on the accuracy of the segmentation algorithms used for extracting regions. Without the ability to identify accurate segmentations, resulting object recognition accuracy of the region-based approaches will be worse than brute-force scanning algorithms. Fortunately, with advances in image contour detection [81] and segmentation [7], we can generate very accurate segmentations on images. Having accurate regions leads to a significant reduction in the size of the search space compared to multi-scale scanning algorithms, and defines better separation between foreground objects and the background – the background noise problem in multi-scale scanning algorithms can be reduced. As such, region-based analysis will become more important in future object recognition research efforts. Based on the gPb contour detection algorithm [81] and the UCM segmentation algorithm [7], the object recognition system developed by Gu et al. [61] uses regions to identify objects. This system achieves very high performance on the false positive per image (FPPI) curve of the ETHZ [50] benchmark, and competitive performance on the Caltech 101 [46]

benchmark, thus closing the performance gap between multi-scale scanning algorithms and region-based algorithms.

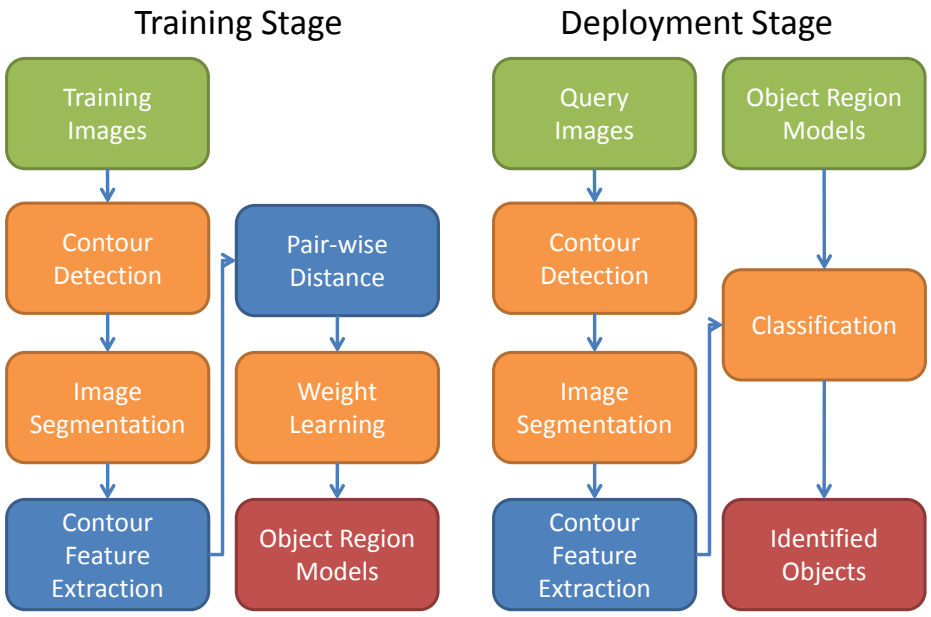

Figure 7.1: The application-level software architecture of the object recognition system in [61].

The region-based object recognition system by Gu et al. is summarized in Figure 7.1. The basic idea is to identify objects by region matches. In the training stage, gPb contours of training images are detected, and then images are segmented into regions using the UCM algorithm. Contour features are collected to represent the shapes of the regions, and the χ^2 distances between each pair of feature vectors are computed. Although each object is segmented into multiple regions, not every region is equally important. Discriminative regions should reappear on multiple instances of the same object, and have long distances compared to regions from other objects or the background. Based on this assumption, a quadratic programming problem is formulated to compute the weight of each region. Similarly, in the deployment stage contours of a query image are extracted by the gPb algorithm [81] and the query image is segmented by the UCM algorithm [7]. Contour features of the regions are extracted from the segmentation results, and the Hough Voting algorithm scores matches between the query regions and training regions.

Although the region-based system by Gu et al. [61] is a high-quality object recognition system, it is nonetheless computationally intensive. During the training stage, for example, the training procedure on 127 images with an average size of 0.15M pixels takes 32,393 seconds. Even if training images are segmented in advance, feature extraction and region model building still takes 2,332 seconds. Similarly, during the deployment stage, object recognition takes 331 seconds on a 0.15M pixel image. As a result, it is essential to parallelize

and optimize this application if we want to employ it in real life.

7.2 Parallelizing the Object Recognition System

Figure 7.1 shows the topmost application-level software architecture of the object recognition system. In order to parallelize and optimize this system, we conduct a more detailed study on lower hierarchies of the software architecture, replace application patterns with library functions we have developed, and evaluate the resulting performance.

7.2.1 The Software Architecture of the Object Recognition System

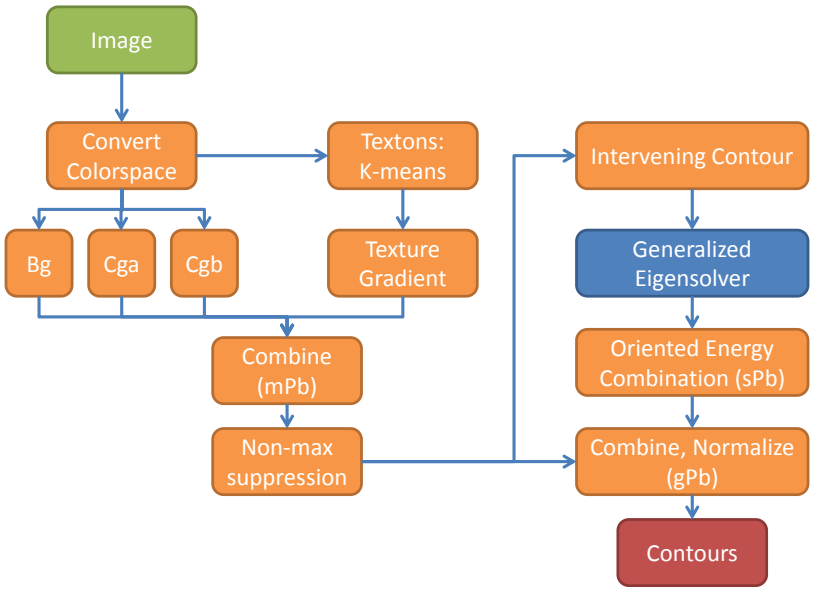

Figure 7.2: The software architecture of the gPb contour detection algorithm [81].

The first three steps of both the training and deployment stages are the same: detecting contours from the images, segmenting the images, and then collecting contour features for the regions.

Currently, the highest-quality image contour detector (as measured by the Berkeley Segmentation Dataset [87]) is the gPb detector [81], which is used in the object recognition system. The application-level software architecture of the gPb algorithm is summarized in Figure 7.2. The gPb detector consists of many modules that can be grouped into two main components: mPb, a detector based on local image analysis at multiple scales; and sPb, a detector based on the normalized cuts criterion.

The mPb detector is constructed from brightness, color, and texture cues at multiple scales. For each cue, the detector from [86] is employed, estimating the probability of boundary $Pb_{C,\sigma}(x, y, \theta)$ for a given image channel, scale, pixel, and orientation by measuring the difference in image channel C between two halves of a disc of radius σ centered at (x, y) and oriented at angle θ. The cues are computed over four channels: the CIELAB 1976 L channel, which measures brightness, and A, B channels, which measure color, as well as a texture channel derived from texton labels [84]. The cues are also computed over three different scales $[\frac{\sigma}{2}, \sigma, 2\sigma]$ and eight orientations in the interval $[0, \pi)$. The mPb detector is then constructed as a linear combination of the local cues, where the weights α_{ij} are learned by training on an image database:

$$mPb(x, y, \theta) = \sum_{i=1}^{4} \sum_{j=1}^{3} \alpha_{ij} Pb_{C_i,\sigma_j}(x, y, \theta). \tag{7.1}$$

The mPb detector is then reduced to a pixel affinity matrix W, whose elements W_{ij} estimate the similarity between pixel i and pixel j by measuring the intervening contour [78] between pixels i and j. Due to computational concerns, W_{ij} is not computed between all pixels i and j, but for only those pixels that are near to each other. In this case, we use Euclidean distance as the constraint, meaning that we compute only $W_{ij}\ \forall i, j\ s.t.\ ||(x_i, y_i) - (x_j, y_j)|| \leq r$; otherwise, we set $W_{ij} = 0$. In this case, we set $r = 5$.

Once W has been constructed, sPb follows the normalized cuts approach [108], which approximates the NP-hard normalized cuts graph partitioning problem by solving a generalized eigensystem. Only the $k + 1$ eigenvectors v_j with smallest eigenvalues are useful in image segmentation and need to be extracted. In this case, we use $k = 8$. The smallest eigenvalue of this system is known to be 0, and its eigenvector is not used in image segmentation, so we extract $k + 1$ eigenvectors. After computing the eigenvectors, we extract their contours using Gaussian directional derivatives at multiple orientations θ, to create an oriented contour signal $sPb_{v_j}(x, y, \theta)$. We combine the oriented contour signals based on their corresponding eigenvalues:

$$sPb(x, y, \theta) = \sum_{j=2}^{k+1} \frac{1}{\sqrt{\lambda_j}} sPb_{v_j}(x, y, \theta). \tag{7.2}$$

The final gPb detector is then constructed through linear combination of the local cue information and the sPb cue:

$$gPb(x, y, \theta) = \gamma \cdot sPb(x, y, \theta) + \sum_{i=1}^{4} \sum_{j=1}^{3} \beta_{ij} Pb_{C_i,\sigma_j}(x, y, \theta), \tag{7.3}$$

where the weights γ and β_{ij} are also learned via training. To derive the final $gPb(x, y)$ signal, we maximize over θ, threshold to remove pixels with very low probability of being a contour pixel, skeletonize, and then renormalize.

The gPb signal is then used to segment images following the UCM algorithm [7]. The application-level software architecture of the UCM algorithm is summarized in Figure 7.3.

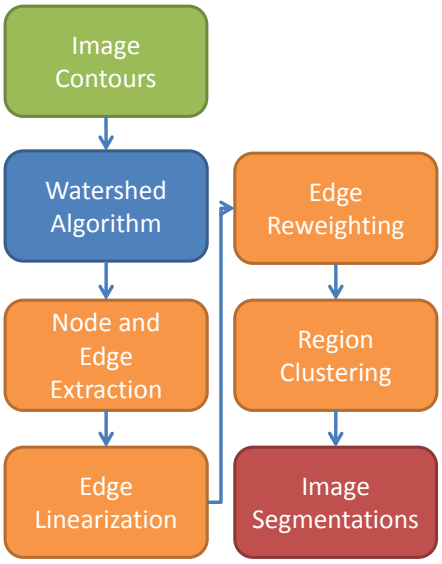

Figure 7.3: The software architecture of the UCM segmentation algorithm [7].

Given the contours (the gPb signal) of an input image, the watershed algorithm [90] is applied on the contours, which results in an over-segmentation of the image. The watershed boundaries are represented as connected graphs – nodes are used to represent intersections across multiple boundaries, and edges are used to represent boundaries between nodes. An edge between two nodes can follow an arbitrary path. The UCM algorithm approximates each edge by a set of linear segments. This linearization step makes it possible to compute an orientation value for every edge segment. The accurate contours generated by the gPb algorithm are used to associate weight with the watershed boundaries. A linear edge segment on the watershed boundary, matching the orientation of the contours generated by the gPb algorithm, will have a higher weight. Otherwise, it will be given a lower weight. The over-segmentation effect caused by the watershed algorithm is alleviated by reducing the weights of redundant boundaries. The final step of the segmentation algorithm is to further refine the weight assignment on the watershed boundaries, using a region-clustering algorithm.

After segmenting an image, the contour features for regions on the image are collected. The computation is the same as the contour histogram computation introduced in Section 5.3.

The training stage uses the contour features from the training images to develop a region model that identifies discriminative regions on objects. The χ^2 distances between each pair of feature vectors are computed, and are used in the following weight-learning step. The weight-learning step is formulated as a quadratic programming problem. Given an image I, we need to compute the weight assignment of all its regions. Let J be the set of images in the same object category as image I, and K be the set of images in the different object categories. We select T pairs of images $(J_1, K_1), (J_2, K_2), \ldots, (J_T, K_T)$ from J and K, and

use these T pairs to train the weights of regions in image I. Let $d(X, Y)$ be the distance between image X and image Y. We use quadratic programming to find discriminative regions, which make $d(I, K_t) > d(I, J_t), \forall t \in T$. The generalized simplex algorithm is used to solve this problem.

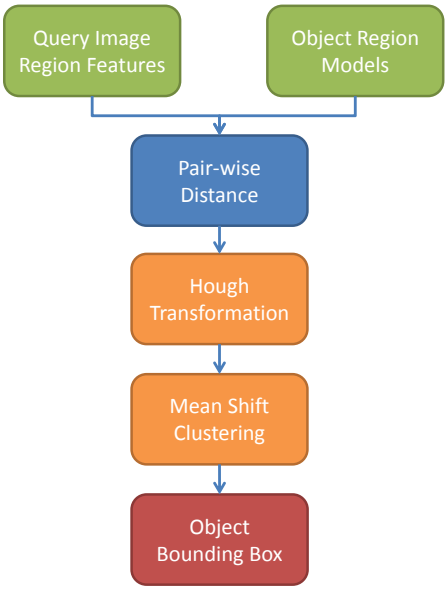

Figure 7.4: The software architecture of the classification step in [61].

The deployment stage uses the contour features from a query image to recognize objects. The application-level software architecture of the classification step is summarized in Figure 7.4. The χ^2 distances between the query features and the training features are computed – if the distance is small enough, a region match is reported. Because a matched region might represent only a portion of an object, the Hough transformation algorithm is used to compute the bounding box of the overall object based on the matched region. Multiple region matches of the same object generate multiple conjectures of the bounding boxes of the object. Therefore, the mean-shift clustering algorithm is used to group all conjectures into a single conjecture, with weight equal to the summation over the weights of all conjectures. The weights of conjectures come from the region weights learned during the training stage.

7.2.2 Using the Parallel Application Library to Parallelize and Optimize the Object Recognition System

The application-level software architecture of the object recognition system by Gu et al. [61] is shown in Figures 7.1, 7.2, 7.3, and 7.4. We have developed many functions for the parallel object application library in Chapters 5 and 6, and these highly parallelized and optimized functions can be employed in the object recognition system to speed up the

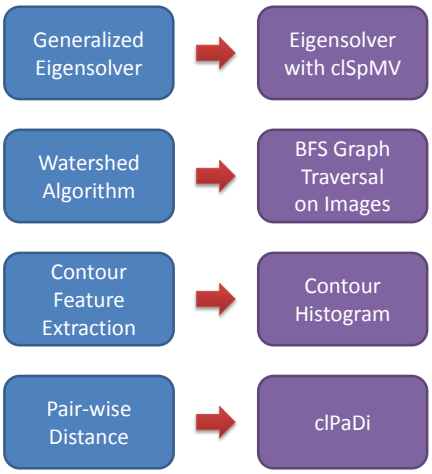

Figure 7.5: Computations with the corresponding parallel library functions.

application. To do so, we simply replace the application patterns in the software architecture with these functions. The replaceable application patterns are in blue boxes in the software architecture, and are summarized in Figure 7.5.

The generalized eigensolver pattern in the gPb contour detection algorithm can be replaced by the eigensolver function discussed in Section 5.1. We use the *clSpMV* autotuner in Section 6.3 to optimize the sparse matrix vector multiplication (SpMV) kernel in the eigensolver. The final results suggest using only the diagonal format to represent the affinity matrix. This matches our observations from Section 5.1. Therefore, we employ our original eigensolver without any changes on the SpMV kernel. The watershed algorithm [90] is composed of three steps: identifying local minimums in an image, assigning unique labels to these local minimums, and propagating labels from the smallest to the largest gray level. These steps can be implemented using BFS graph traversal. We replace the watershed computation with our BFS graph traversal function in Section 5.2. The contour feature extraction step is the same as the contour histogram function in Section 5.3. The pair-wise χ^2 distance computations in both the training and deployment stages are replaced by the *clPaDi* autotuner.

Although we do not develop library functions for the remaining application patterns in the software architecture, we still parallelize these computations with reasonable efforts. The mPb and sPb signals in the contour detection are parallelized using data parallelism strategies on a GPU. The quadratic programming computation in the weight learning step is parallelized using a task parallelism strategy on a CPU.

7.2.3 Experimental Results

In this section, we examine the overall performance of the parallel object recognition system we have developed, and compare our system with the original serial implementation

Table 7.1: Performance of the parallel object recognition system: deployment stage.

Computation	Computation time (s)		Speedup
	Serial(double-precision)	Parallel(single-precision)	
Contour Detection	236.7	1.58	150 ×
Image Segmentation	2.27	0.357	6.36 ×
Feature Extraction	7.97	0.065	123 ×
Classification	84.13	0.779	108 ×
Total	331.07	2.781	119 ×

Table 7.2: Performance of the parallel object recognition system: training stage.

Computation	Computation time (s)		Speedup
	Serial(double-precision)	Parallel(single-precision)	
Feature Extraction	543	15.97	34 ×
χ^2 Distance	1732	2.9	597 ×
Weight Learning	57	1.41	40 ×
Total	2332	20.28	115 ×

in [61]. In the original serial implementation, the gPb contour detector [81] is implemented mostly in C++. The UCM segmentation algorithm [7] is implemented half in C++ and half in MATLAB. The remaining parts are implemented in MATLAB. The original serial implementation uses double-precision floating point algebra, and is executed on an Intel Core i7 920 (2.66 GHz) machine.

Our parallel library is implemented with single-precision floating point algebra because Nvidia cards with CUDA computability prior to 1.2 do not support double-precision floating point operations. In order to support earlier Nvidia cards, we decide to implement our library in single-precision floating point algebra. If we re-implement our parallel library in double-precision algebra, very likely the execution time will be doubled. The deployment stage is executed on an Nvidia GTX 480 GPU. For the training stage, the contour detection and image segmentation computations are the same as in the deployment stage. The performance improvement is also the same as in the deployment stage. To avoid repetition, we assume that training images are already segmented and only report the execution times for feature collection and model building. The χ^2 distance computation in the training stage is executed on an Nvidia Tesla C1060 GPU, because the distance table in the training stage exceeds the 1.5G memory on a GTX 480, but fits in the 4G memory on a Tesla C1060. The parallel weight-learning computation is executed on an Intel Core i7 920 (2.66 GHz) machine with 8 threads. The performance of the parallel object recognition system is summarized in Tables 7.1 and 7.2. The deployment stage is executed on an image with 0.15M pixels. The training stage is executed on 127 images with 0.15M pixels in average. Moving from a double-precision MATLAB implementation to a single-precision C++ implementation results in 3-6× speedup. The remaining 20-40× speedup is gained by replacing the original serial implementations with our highly-optimized parallel application library functions. We obtain only 6× speedup on the segmentation computation, since only the watershed algorithm is

parallelized, and the computation is in integer so we do not get the extra 2× speedup by replacing double-precision algebra with single-precision algebra.

In addition to evaluating execution time, we also evaluate the detection quality of the parallelized object recognition system on the ETHZ shape benchmark [50]. The detection quality is shown in Figure 7.6, using the detection rate of the False Positive Per Image (FPPI) metric. Compared to the serial implementation, the detection quality of the parallel implementation is slightly worse. The difference in quality results primarily from numerical errors. MATLAB uses double precision floating point operations, while we use only single precision floating point operations in order to achieve the best utilization rate of GPU computing resources. Numerical errors influence the thresholding operations on the χ^2 distance table, forcing the parallel implementation to choose different image pairs for computing the region weights of a training image. Detection quality is then influenced by the discrepancy of region weights in training images.

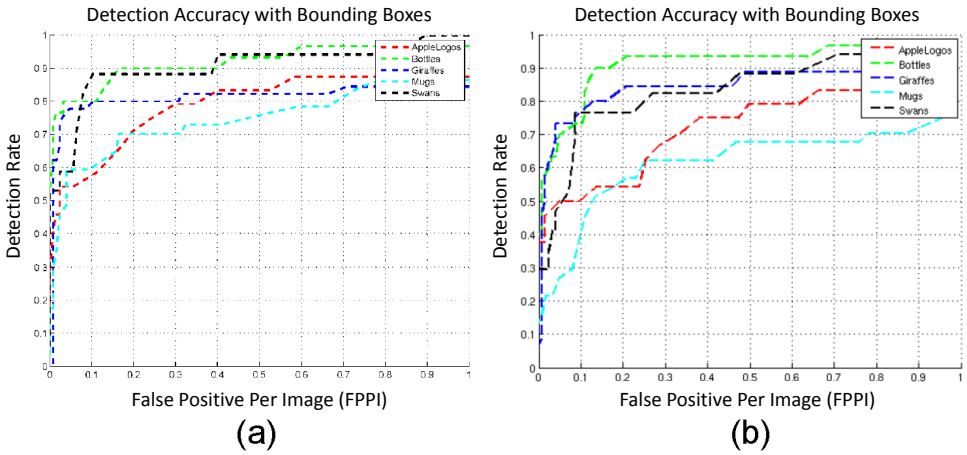

Figure 7.6: (a) The detection rate versus FPPI curve of the original serial implementation. (b) The detection rate versus FPPI curve of the parallel object recognition system.

In summary, we demonstrate how employing the parallel application library can help transform computationally-intensive object recognition applications into near real-time applications. We achieve 110-120× speedup on both the deployment stage and the training stage, compared to [61]. Using this parallel object recognition system, we are able to train hundreds of images in minutes, and classify an image in only three seconds.

7.3 Summary

We demonstrate using the functions of the parallel application library to accelerate the object recognition system by Gu et al. [61]. This system is very computationally intensive – the training stage with training images segmented in advance takes 2332 seconds, while the deployment stages takes 331 seconds on a 0.15 mega pixel image. Although this is a

highly-accurate system, the execution time is too long to be employed in real life. We study the application-level software architecture to understand the essential computations, then deploy the parallel application library by replacing computations in the software architecture with the highly parallelized and optimized functions introduced in Chapters 5 and 6. The functions we support cover only a subset of the computations, but these are the bottlenecks of the system. We achieve 115× speedup in the training stage, and 119× speedup in the deployment stage with comparable detection accuracy. As a result, we are able to detect objects in a 0.15 mega pixel image in only three seconds. The technology we have developed makes it possible and practical to employ this object recognition system in real life.

Chapter 8

Conclusions and Future Work

Object recognition is a major branch in the field of computer vision. If we can enable computers to identify objects in images, we will be better able to make computers understand the contents of images as well. However, state-of-the-art object recognition algorithms are computationally-intensive. Worse, this computational cost grows with the number of object categories to be recognized, the number of images for training and classification, and the resolution of those images. For example, if we apply the object recognition system by Gu et al. [61] on the PASCAL VOC benchmark [44], the training stage on 5011 images requires 529 tera floating point operations (FLOPs), and the classification stage on 4952 images consumes 164 tera FLOPs. It takes years to finish these computations on even the best available single-core desktop CPU system. This expensive computational cost becomes a barrier to deploying these highly-accurate object recognition systems in real life. Therefore, we must explore parallelism in these systems, and accelerate them on parallel platforms.

However, special expertise is required to parallelize and optimize a given computation on a given parallel platform. Application developers understand the algorithms and computations used in their applications, but lack the knowledge of parallelizing and optimizing their code on parallel platforms. On the other hand, expert parallel programmers master mapping computations to parallel platforms, but don't understand the algorithms used in the applications. As a result, an implementation gap exists between application developers and expert parallel programmers. In order to bridge this gap, we propose developing a parallel application library for object recognition systems.

8.1 Contributions

Here, we summarize the major contributions of this dissertation.

8.1.1 Application Patterns for Object Recognition

We perform pattern mining on 31 state-of-the-art object recognition papers selected from CVPR 2007 through CVPR 2011, and conclude that only 15 application patterns are required to develop these systems. These 15 application patterns are convolution, histogram

accumulation, vector distance, quadratic optimization, graph traversal, eigen decomposition, k-means clustering, Hough transform, nonlinear optimization, meanshift clustering, fast Fourier transform, singular value decomposition, convex optimization, k-medoids clustering, and agglomerative clustering. These patterns therefore become the feature list of the proposed parallel application library for object recognition. As long as we support these patterns in our library, we are able to parallelize and optimize all 31 object recognition systems under consideration.

8.1.2 Parallelizing and Optimizing Application Patterns

Parallelizing and optimizing a computation on a parallel platform is not trivial – a systematic procedure is necessary to achieve this goal. We propose using Our Pattern Language (OPL) [73] to architect a computation, and then employing this software architecture to guide design space exploration. A design space can be divided into three layers: algorithms, parallelization strategies, and platform parameters. The algorithm layer is the superset over all valid transformations from input data to output data. The parallelization strategy layer is the superset over all valid mappings from an algorithm to the underlying parallel platform. The platform parameter layer is the superset over all valid configurations of parameters used in a parallelization strategy.

We employ the exhaustive search strategy to optimize three application patterns. The first is an eigensolver for the normalized cut algorithm, for which we employ the Lanczos algorithm [12]. There are three major bottlenecks in the Lanczos algorithm, which we have resolved through design space exploration. Sparse matrix vector multiplication (SpMV) computation is optimized by representing the matrix using the diagonal format. Reorthogonalization overhead is conquered by employing the no-reorthogonalization strategy. Convergence checking overhead is reduced by performing that operation infrequently. With these methods, we have achieved 280× speedup compared to the original serial MATLAB solver. The second application pattern is breadth-first-search (BFS) graph traversal on images. We propose using structured grids to replace the original BFS graph traversal computation. This strategy results in $12 - 33\times$ speedup compared to a serial implementation. The third application pattern is contour histogram. We have experimented with different parallelization strategies, and determined that parallel reduction is the best strategy for this application pattern. With this, we have achieved $5 - 30\times$ speedup compared to a serial implementation.

To improve the portability of the proposed parallel application library, we initiate the OpenCL for OpenCV (OLOV) project. The goal of this project is to develop a collection of autotuners to optimize computationally-intensive application patterns on GPU platforms. This dissertation specifically contributes *clSpMV* and *clPaDi* autotuners to the project.

clSpMV is an autotuner for sparse matrix vector multiplication (SpMV) computation. We propose using the *Cocktail Format* to represent sparse matrices. The *Cocktail Format* is itself a collection of sparse matrix formats. A sparse matrix is partitioned into many submatrices, each of which is represented by a specialized format. Currently, *clSpMV* supports nine sparse matrix formats: DIA, BDIA, ELL, SELL, CSR, COO, BELL, SBELL,

and BCSR. *clPaDi* determines the best representation of a sparse matrix, and optimizes the SpMV kernel on the special representation. It achieves 40% better performance compared to the Nvidia vendor-optimized code. It also performs 16.8% and 43.3% better compared to all single formats on Nvidia and AMD platforms, respectively.

clPaDi is an autotuner for pair-wise distance computation. It allows the user to customize a distance function using three operators – the element, reduction, and post-processing operators. Based on the customized distance function, it finds the best blocking size on the underlying platform. *clPaDi* achieves 650 giga floating point operations per second (GFLOPS) with dot product distance and 320 GFLOPS with χ^2 distance on an Nvidia platform. It also achieves 450 GFLOPS and 405 GFLOPS for the two distance functions on an AMD platform.

From these case studies, we illustrate the effectiveness of parallelizing and optimizing application patterns by exploring the design space. We have achieved significant speedups compared to serial implementations, and even compared to state-of-the-art parallel implementations. Following the same strategy to fully develop the parallel application library will result in an optimized library for object recognition computations.

8.1.3 Developing a Parallel Object Recognition System Using the Application Library

We employ the library functions we developed to optimize the region-based object recognition system by Gu et al. [61], which is highly-accurate, but computationally very expensive. We decompose the object recognition system with patterns to understand its software architecture, and identify application patterns that match the functions in the parallel application library. We then replace these patterns with our own highly-optimized parallel functions. Experimental results show that we achieve 115× speedup in the training stage, and 119× speedup in the deployment stage with comparable detection accuracy to [61]. As a result, we are able to recognize objects in only three seconds. These improvements make it practical to deploy this system in real life.

8.2 Future Work

In this dissertation, we demonstrate two different methods of library development. The first optimizes specific computations on a specific platform. The three application patterns discussed in Chapter 5 employ this strategy. This method is more narrowly-focused – no user customization is available, and the underlying hardware platform is fixed. However, there are still advantages to this development philosophy. For example, the design space is significantly reduced because we need not worry about architectural differences between different platforms. Moreover, because we do not provide user customizations, we also do not need to create an abstract programming model that covers all variations of an application pattern and enables end users to instantiate any subset of the application pattern. As a result, the application patterns are simplified, and the library development procedure can be accelerated. For a short-term goal, we would like to follow this method to expand

the parallel library with more optimized functions until we have full support for the 15 application patterns in Section 3.3.

The second method of library development uses autotuners to provide portable and flexible solutions. The OLOV introduced in Chapter 6 employs this method. However, the function optimization problem is more difficult. We need to understand the architectures of different platforms, implement different optimization strategies for each, and design autotuners to make dynamic decisions based on the underlying platform. Moreover, we need to create an abstract programming model that enables users to instantiate any subset of the application pattern. For example, in *clPaDi* we provide the abstract programming model of three operators with which an end user can define arbitrary distance functions. This development philosophy is more challenging, but also more general-purpose. An end user can experiment with different ideas instead of using fixed functions. For our long-term goal, we would like to expand the OLOV project by developing autotuners for all 15 application patterns in Section 3.3.

In conclusion, for our short-term goal, we would like to expand our library to the point that we can parallelize all 31 object recognition systems. Our long term goal is to provide autotuners with which users can develop their own parallel object recognition systems on any parallel platform.

8.3 Summary

We see the computational challenge in object recognition research, and believe the problem will only continue to grow. It is therefore essential for computer vision researchers to take advantage of massively-parallel hardware platforms to accelerate their applications. To overcome the difficulty of parallel programming, this dissertation proposes a parallel application library to help computer vision researchers parallelize their applications. Our initial results are encouraging: we have used our library to accelerate a computationally-intensive object recognition by two orders of magnitude. By expanding the parallel application library to provide more functionalities, we are confident that it will become a key development toolkit for all existing and future object recognition systems.

Bibliography

[1] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. GpuCV: an opensource GPU-accelerated framework for image processing and computer vision. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pages 1089–1092, New York, NY, USA, 2008. ACM.

[2] AMD. ATI Stream Computing User Guide, 2008.

[3] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, 2011. `http://developer.amd.com/zones/OpenCLZone`.

[4] AMD. AMD Accelerated Parallel Processing BLAS Library, 2012. `http://developer.amd.com/libraries/appmathlibs/Pages/default.aspx`.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] M. Andriluka, S. Roth, and B. Schiele. Pictorial structures revisited: People detection and articulated pose estimation. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1014 –1021, June 2009.

[7] P. Arbeláez, M. Maire, Charles Fowlkes, and J. Malik. From contours to regions: An empirical evaluation. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[8] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[10] S. Baden, N. Chrisochoides, D. Gannon, and M. Norman. *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*. Springer, London, UK, 1999.

[11] D.A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP*, 2006.

[12] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, 2000.

[13] O. Barinova, V. Lempitsky, and P. Kohli. On detection of multiple object instances using hough transforms. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2233 –2240, June 2010.

[14] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 18:1–18:11, New York, USA, 2009.

[15] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. In *Journal of Computational Physics*, 1984.

[16] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. Technical report, 1996.

[17] Guy Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38:1526–1538, 1987.

[18] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1 –8, June 2008.

[19] Rajesh Bordawekar and Muthu Manikandan Baskaran. Optimizing sparse matrix-vector multiplication on GPUs. *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[20] Lubomir Bourdev and Jitendra Malik. Poselets: Body part detectors trained using 3D human pose annotations. In *International Conference on Computer Vision (ICCV)*, 2009.

[21] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[22] E. Breen and R. Jones. Attribute openings, thinnings, and granulometries. In *CVIU*, volume 64, pages 337–389, 1999.

[23] A. Buluc, and, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 721–733, may 2011.

[24] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[25] Bryan Catanzaro. *Compilation Techniques for Embedded Data Parallel Languages*. PhD thesis, EECS Department, University of California, Berkeley, May 2011.

[26] Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, and Kurt Keutzer. Efficient high quality image contour detection. In *International Conference on Computer Vision*, September 2009.

[27] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 115–126, New York, USA, 2010.

[28] Jike Chong. *Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2010.

[29] S. Barry Cooper. *Computability Theory*. CRC Press, 2004.

[30] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[31] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[32] Jane K. Cullum and Ralph A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Vol. I: Theory. SIAM, 2002.

[33] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *International Conference on Computer Vision and Pattern Recognition*, volume 2, pages 886–893, June 2005.

[34] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[35] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. In *SIREV*, volume 51 of *1*, pages 129–159, 2009.

[36] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[37] T. A. Davis and Y. Hu. University of florida sparse matrix collection. 38(1), 2011. `http://www.cise.ufl.edu/research/sparse/matrices`.

[38] Hongli Deng, Wei Zhang, E. Mortensen, T. Dietterich, and L. Shapiro. Principal curvature-based region detector for object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1 –8, June 2007.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[40] T. Deselaers and V. Ferrari. Global and efficient self-similarity for object classification and detection. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1633 –1640, June 2010.

[41] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[42] Ulrich Drepper. What every programmer should know about memory, 2007.

[43] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[44] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html, 2007.

[45] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

[46] L. Fei-Fei, F. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach testing on 101 object categories. In *Workshop on Generative-Model Based Vision, CVPR*, 2004.

[47] P. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *CVPR*, 2008.

[48] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2), September 2004.

[49] V. Ferrari, M. Marin-Jimenez, and A. Zisserman. Progressive search space reduction for human pose estimation. In *CVPR*, 2008.

[50] V. Ferrari, T. Tuytelaars, and L. Van Gool. Object detection by contour segment networks. In *ECCV*, May 2006.

[51] Joseph A. Fisher. Retrospective: very long instruction word architectures and the eli-512. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 34–36, New York, NY, USA, 1998. ACM.

[52] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the fft. pages 1381–1384. IEEE, 1998.

[53] James Fung, Steve Mann, and Chris Aimone. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of the ACM Multimedia*, pages 849–852, November 2005.

[54] J. Gall and V. Lempitsky. Class-specific hough forests for object detection. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1022 –1029, June 2009.

[55] Google Inc. Google Goggles, 2011. `http://www.google.com/mobile/goggles/`.

[56] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[57] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *POOSC*, 2005.

[58] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 12:1–12:8, New York, USA, 2011.

[59] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[60] R. G Grimes, D. R. Kincaid, and D. M. Young. Itpack 2.0 user's guide. Technical Report CNA-150, University of Texas, Austin, TX, USA, August 1979.

[61] C. Gu, J. Lim, P. Arbeláez, and J. Malik. Recognition using regions. In *Proc. International Conference on Computer Vision and Pattern Recognition*, 2009.

[62] M. Guillaumin, J. Verbeek, and C. Schmid. Multimodal semi-supervised learning for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 902 –909, June 2010.

[63] Ping Guo and Liqiang Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In *International Conference on Computational and Information Sciences (ICCIS)*, pages 1154–1157, 2010.

[64] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[65] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[66] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[67] Steven C. H. Hoi, Michael R. Lyu, and Edward Y. Chang. Learning the unified kernel machines for classification. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 187–196, New York, NY, USA, 2006. ACM.

[68] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, pages 18:135–18:158, February 2004.

[69] Intel. Intel core i7-920 processor, November 2008. `http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor`.

[70] Intel. Intel Advanced Vector Extensions Programming Reference. 2009. `http://software.intel.com/en-us/avx`.

[71] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

[72] Capers Jones. *Programming Productivity*. Mcgraw-Hill, 1986.

[73] Kurt Keutzer and Tim Mattson. A Design Pattern Language for Engineering (Parallel) Software. *Intel Technology Journal, Addressing the Challenges of Tera-scale Computing*, 13(4), 2010.

[74] Gunhee Kim, C. Faloutsos, and M. Hebert. Unsupervised modeling of object categories using link analysis techniques. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1 –8, June 2008.

[75] C. H. Lampert, M. B. Blaschko, and T. Hofmann. Beyond sliding windows: Object localization by efficient subwindow search. In *CVPR*, 2008.

[76] C.H. Lampert. Partitioning of image datasets using discriminative context information. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1 –8, June 2008.

[77] Y.J. Lee and K. Grauman. Object-graphs for context-aware category discovery. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1 –8, June 2010.

[78] Thomas Leung and Jitendra Malik. Contour continuity in region based image segmentation. In *Proc. European Conference on Computer Vision*, pages 544–559. Springer-Verlag, 1998.

[79] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning gemm for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[80] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[81] M. Maire, P. Arbeláez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. *Proc. International Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[82] S. Maji, A.C. Berg, and J. Malik. Classification using intersection kernel support vector machines is efficient. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1 –8, June 2008.

[83] S. Maji and J. Malik. Object detection using a max-margin hough transform. In *CVPR*, 2009.

[84] Jitendra Malik, Serge Belongie, Jianbo Shi, and Thomas Leung. Textons, contours and regions: Cue integration in image segmentation. In *Proc. International Conference on Computer Vision*, page 918, Washington, DC, USA, 1999. IEEE Computer Society.

[85] T. Malisiewicz and A. Efros. Recognition by association via learning per-exemplar distances. In *CVPR*, 2008.

[86] D. Martin, C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using brightness and texture. In *Advances in Neural Information Processing Systems*, volume 14, 2002.

[87] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *ICCV*, volume 2, pages 416–423, July 2001.

[88] David R. Martin, Charless C. Fowlkes, and Jitendra Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:530–549, 2004.

[89] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[90] F. Meyer. Topographic distance and watershed lines. In *Signal Processing*, 1994.

[91] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K.R. Mullers. Fisher discriminant analysis with kernels. In *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop*, pages 41 –48, aug 1999.

[92] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. *High Performance Embedded Architectures and Compilers*, pages 111–125, 2010.

[93] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[94] Neeraj Kumar et al. Leafsnap: An Electronic Field Guide, 2011. `http://leafsnap.com/`.

[95] E. Nowak and F. Jurie. Learning visual similarity measures for comparing never seen objects. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1 –8, June 2007.

[96] Nvidia. Compute Unified Device Architecture Programming Guide, 2007. `http://nvidia.com/cuda`.

[97] Nvidia. CUDA Basic Linear Algebra Subroutines, 2012. `http://developer.nvidia.com/cuda/cublas`.

[98] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. In *IJCV*, pages 145–175, 2001.

[99] OpenMP. OpenMP API specification for parallel programs. `http://openmp.org`.

[100] David A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.

[101] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[102] William Lester Plishker. *Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2006.

[103] Qualcomm Inc. Vuforia — Augmented Reality — Qualcomm, 2012. `http://www.qualcomm.com/solutions/augmented-reality`.

[104] Linda Rising. *The Patterns Handbook: Techniques, Strategies, and Applications*. Cambridge University Press, 1998.

[105] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[106] B. C. Russell, A. A. Efros, J. Sivic, W. T. Freeman, and A. Zisserman. Using multiple segmentations to discover objects and their extent in image collections. In *CVPR*, 2006.

[107] D. P. Scarpazza, O. Villa, and F. Petrini. Efficient breadth-first search on the Cell/BE processor. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1381–1395, 2008.

[108] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, Aug 2000.

[109] Youtube Statistics. Youtube statistics, May 2012. `http://www.youtube.com/t/press_statistics`.

[110] Bor-Yiing Su, Tasneem Brutch, and Kurt Keutzer. Parallel BFS graph traversal on images using structured grid. In *Proc. International Conference on Image Processing*, pages 4489–4492, September 2010.

[111] Bor-Yiing Su, T.G. Brutch, and K. Keutzer. A parallel region based object recognition system. In *IEEE Workshop on Applications of Computer Vision (WACV)*, pages 81 –88, jan. 2011.

[112] Bor-Yiing Su and Kurt Keutzer. clspmv: A cross-platform opencl spmv framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012. ACM.

[113] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2011.

[114] TBB. Threading Building Blocks for open source. `http://threadingbuildingblocks.org`.

[115] S. Thakkur and T. Huff. Internet streaming simd extensions. *Intel Technology Journal Q2*, 32(12):26–34, dec 1999.

[116] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems, 2011. `http://www.khronos.org/opencl`.

[117] R. Urquhart. Graph theoretical clustering based on limited neighborhood sets. In *Pattern Recognition*, volume 15, pages 173–187, 1982.

[118] F. Vázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 1146–1151, 2010.

[119] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[120] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, June 2005.

[121] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[122] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[123] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 38:1–38:12, New York, USA, 2007.

[124] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[125] S.A.J. Winder and M. Brown. Learning local image descriptors. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1 –8, June 2007.

[126] Kesheng Wu and Horst Simon. Thick-restart lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, pages Vol.22, No. 2, pp. 602–616, 2001.

[127] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 11, pages 1101–1113, 1993.

[128] Y. Xia and V. K. Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. In *PDCS*, 2009.

[129] Liu Yang, Rong Jin, R. Sukthankar, and F. Jurie. Unifying discriminative visual codebook generation with classifier training for object category recognition. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1 –8, June 2008.

[130] Shulin Yang, Mei Chen, D. Pomerleau, and R. Sukthankar. Food recognition using statistics of pairwise local features. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2249 –2256, June 2010.

[131] Saad Yousef. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[132] C.T. Zahn. Graph-theoretic methods for detecting and describing gestalt clusters. In *IEEE Transactions on Computing*, volume 20, pages 68–86, 1971.

[133] Hao Zhang, A.C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2126 – 2136, 2006.