TRADEOFFS IN DESIGNING MASSIVELY PARALLEL ACCELERATOR ARCHITECTURES

BY

AQEEL A. MAHESRI

B.S., University of California at Berkeley, 2002
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Associate Professor Sanjay J. Patel, Chair
Professor Josep Torrellas
Professor Wen-Mei Hwu
Assistant Professor Craig Zilles
ABSTRACT

There is a large, emerging, and commercially relevant class of applications which stands to be enabled by a significant increase in parallel computing throughput. Moreover, continued scaling of semiconductor technology allows us the creation of architectures with tremendous throughput on a single chip. In this thesis, we examine the confluence of these emerging single-chip accelerators and the applications they enable. We examine the tradeoffs associated with accelerator architectures, working our way down the abstraction hierarchy of computing starting at the application level and concluding with the physical design of the circuits.

Research into accelerator architectures is hampered by the lack of standardized, readily available benchmarks. Among these applications is what we refer to as visualization, interaction, and simulation (VIS). These applications are ideally suited for accelerators because of their parallelizability and demand for high throughput. We present VISBench, a benchmark suite to serve as an experimental proxy for VIS applications. VISBench contains a sampling of applications and application kernels from traditional visual computing areas such as graphics rendering and video encoding. It also contains a sampling of emerging application areas, such as computer vision and physics simulation, which are expected to drive the development of future accelerator architectures.

We use VISBench to examine some important high level decisions for an accelerator architecture. We propose a methodology to evaluate performance tradeoffs against chip area. We propose a memory system based on a cache incoherent...
shared address space along with mechanisms to provide synchronization and communication. We also examine GPU-style SIMD execution and find that a MIMD architecture is necessary to provide strong performance per area for some applications.

We analyze area versus performance tradeoffs in architecting the individual cores. We find that a design made of small, simple cores achieves much higher throughput than a general purpose uniprocessor. Further, we find that a limited amount of support for ILP within each core aids overall performance. We find that fine-grained multithreading improves performance, but only up to a point. We find that vector ALUs for SIMD instruction sets provide a poor performance to area ratio.

We propose a methodology for performing an integrated optimization of both the micro-architecture and the physical circuit design of the cores and caches. In our approach, we use statistical sampling of the design space for evaluating the performance of the micro-architecture and RTL synthesis to characterize the area-power-delay of the underlying circuits. This integrated methodology enables a much more powerful analysis of the performance-area and performance-power tradeoffs for the low level micro-architecture. We use this methodology to find the optimal design points for an accelerator architecture under area constraints and power constraints. Our results indicate that more complex architectures scale well in terms of performance per area, but that the addition of a power constraint favors simpler architectures.
To my parents, for their love and support.
ACKNOWLEDGMENTS

A great deal of appreciation is due to those individuals who have contributed directly to the results presented in this thesis. I would first like to thank the members of the Rigel group at the University of Illinois. In particular, John Kelm, Daniel Johnson, and Neal Crago have contributed greatly in developing the ideas presented here, and in developing the infrastructure used to explore those ideas. I would like to thank Tom Yeh and Glenn Reinmann of UCLA, who generously contributed physics simulation code. I would like to thank Sam Stone and Chris Rodrigues, who also contributed code kernels for use as benchmarks. I would like to thank Omid Azizi and Mark Horowitz of Stanford University, who developed the co-optimization framework used in this work.

I would like to thank all the past members of the Advanced Computing Systems group. I would like to thank in particular Francesco Spadini, Greg Muthler, Joseph Grzywacz, and Brian Slechta, for their work on the Joshua simulation infrastructure. The continued usability of their work is a testament to their software engineering! I would like to thank Brian Fahs for his valuable advice and assistance during my graduate school career. And I would especially like to thank Nick Wang for his friendship and his collaboration over the years.

I would like to thank all of the faculty members at the University of Illinois with whom I have had the pleasure of working over the years. I would like to thank the members of my thesis committee, including Craig Zilles, Wen-mei Hwu, and Josep Torrellas. I would like to thank Steve Lumetta and Matt Frank for their
assistance and collaboration over the years. Finally, I would like to thank my advisor, Sanjay J. Patel. Professor Patel brings to the architecture community and to my research one of the most valuable traits a researcher can possess, an ability to “see around corners” to find new opportunities.

This research was supported by the Focus Center for Circuit & System Solutions (C2S2), one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td></td>
<td>xiv</td>
</tr>
<tr>
<td>CHAPTER 1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2</td>
<td>BENCHMARKS FOR VISUAL COMPUTING</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>VISbench Applications</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Scanline rendering: Blender renderer</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Ray tracing: POVRay</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Video encoding: H.264 SAD-based motion estimation kernel</td>
<td>11</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Dynamics simulation: ODE PhysicsBench</td>
<td>13</td>
</tr>
<tr>
<td>2.1.5</td>
<td>High quality MRI reconstruction kernel</td>
<td>14</td>
</tr>
<tr>
<td>2.1.6</td>
<td>Computer vision: OpenCV based face detection</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallelization Model</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Characterization</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Instruction mix</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Performance scaling</td>
<td>22</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Thread length</td>
<td>25</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Data communication</td>
<td>31</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>ACCELERATOR META-ARCHITECTURE</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Acceleration Model</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Basic Architecture</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Design Space</td>
<td>36</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>METHODOLOGY</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Optimization Problem</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Area Modeling Methodology</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance Modeling Methodology</td>
<td>48</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Mean thread lengths for our benchmarks. .......................... 26
3.1 Summary of the design space. ....................................... 40
4.1 Area breakdown by pipeline component in mm$^2$. ............... 47
4.2 Area Comparison of Commercial Cores (Normalized to 65nm). .. 47
6.1 Example of stack-based mechanism for handling control flow divergence. Threads marked with an x are executing in the given cycle. ....................................................... 68
6.2 Count of divergence at branches in Facedetect loop. ............ 75
7.1 Baseline XPU architecture. ............................................. 83
7.2 Study parameters. .................................................... 83
7.3 Core counts with multithreading. ................................... 88
7.4 Frequency (per 1000 instructions) of trig and transcendental operations. .................................................. 89
8.1 Co-optimization design space ........................................ 100
8.2 Optimal configuration of five design points on the area-efficient frontier. ................................................. 106
10.1 Important architectural features for each benchmark. ........... 132
## LIST OF FIGURES

2.1 Rendered image used for Blender. ........................................ 10  
2.2 Rendered image used for POVRay. ................................. 11  
2.3 Phases of the physics computation. The parallel phases are shaded. 15  
2.4 Parallelization model used for studies. Each iteration of a parallel loop is one thread. .......................... 18  
2.5 Breakdown of operations in VISBench applications. ........... 21  
2.6 Breakdown of floating point operations. .......................... 21  
2.7 Dynamic instructions in parallel and sequential regions of the code. ............................ 23  
2.8 Performance scaling of the parallel sequences in VISBench. ... 24  
2.9 Estimated performance scaling of the applications in VISBench from applying Amdahl’s Law. .......................... 24  
2.10 CDF of thread length distribution for Blender. ................. 26  
2.11 CDF of thread length distribution for MRI reconstruction. .... 27  
2.12 CDF of thread length distribution for Facedetect. ............. 27  
2.13 CDF of thread length distribution for the different phases of ODE. .......................... 28  
2.14 CDF of thread length distribution for POVRay. ............... 29  
2.15 CDF of thread length distribution for the H.264 SAD kernel. .... 29  
2.16 Proportion of memory reads to shared versus private data. ... 31  
2.17 Proportion of memory writes to shared versus private data. ... 32  
3.1 The CPU/accelerator co-processing model. ..................... 34  
3.2 Basic architecture of the accelerator. ......................... 37  
5.1 Block of parallel code showing possible data sharing. ........ 51  
5.2 Frequency of shared memory accesses. .......................... 52  
5.3 Breakdown of input reads by number of sharers. .............. 53  
5.4 Performance versus gcache bandwidth (normalized to maximum). .......................... 56  
5.5 Performance versus off-chip memory bandwidth (normalized to maximum). .......................... 57  
5.6 Bulk synchronous parallelism model. ........................... 58  
5.7 Number of sharers for benchmarks that have within-barrier data sharing. .......................... 60  
6.1 A MIMD 4-core cluster. ......................................... 64  
6.2 A SIMD 4-core cluster. ........................................ 64
6.3 Example code for stack-based mechanism for handling control flow divergence. 67
6.4 SIMD efficiency versus warp size. 70
6.5 SIMD performance per area versus warp size. 71
6.6 Facedetect parallel loop code. Some details have been removed. 74
6.7 Facedetect first pass loop, version 2. 76
6.8 Facedetect later passes loop, version 2. 77
6.9 Facedetect first pass loop, version 3. 78
6.10 SIMD efficiency for various versions of Facedetect. 79
6.11 SIMD performance per area for various versions of Facedetect. 79

7.1 Performance for different pipeline configurations. 83
7.2 Performance comparison of baseline in1 vs. in2 vs. out2. 85
7.3 Relative performance of chip with SIMD instructions. 87
7.4 Performance of 2-wide in-order configuration with multithreading. 89
7.5 Performance of 2-wide out-of-order configuration with multithreading. 89
7.6 Relative performance of chip with hardware trig unit. 90
7.7 Area per core versus performance per area while sweeping the cache sizes from the smallest to the largest. 91
7.8 Performance versus L1 cache size for the in2 configuration, by benchmark. 92

8.1 Area-performance tradeoffs for various circuits. 96
8.2 Area-performance design space. Each point is a possible design. Those points which lie on the frontier are optimal area-efficient designs. 97
8.3 Overview of the optimization framework. The architectural simulator and model fitting generate the architectural design space. The circuit tradeoffs define the circuit design space. The link is made through a higher level model which then feeds to the optimizer. 99
8.4 Performance-area tradeoff space for based on an average of all six benchmarks. Area is in $\mu m^2$. Five area-efficient designs are annotated, for which we detail the micro-architectural and circuit parameters in Table 8.3. 105
8.5 Performance per area versus area per core tradeoff space based on an average of all six benchmarks. Area is in $\mu m^2$. The same five area-efficient designs are annotated as before. 108
8.6 Performance-area tradeoff space for each individual benchmark. Area is in $\mu m^2$. 110

9.1 Overview of the optimization framework with additional infrastructure for power. In order to determine dynamic power consumption, we need to count the number of operations going through each logic unit and SRAM. 114
9.2 Performance-power tradeoff space for based on an average of all six benchmarks. Power is in mW. ............................... 121
9.3 Performance-power versus power consumption. Power is in mW. 121
9.4 Energy per instruction versus performance. ............................. 122
9.5 Performance and power per core for pareto optimal configurations. Power is in mW. ................................. 123
9.6 Area vs. performance vs. power for all configurations. ............. 124
9.7 Performance versus area under varying power constraints. ......... 125
9.8 Performance versus power under varying area constraints. .......... 126
9.9 Power versus area under varying performance constraints. ......... 128
10.1 Opteron performance .................................................. 130
10.2 xPU performance ....................................................... 130
10.3 xPU performance versus 2.2GHz Opteron .......................... 131
10.4 Total CPU+xPU speedup over CPU only ............................ 131
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>BSP</td>
<td>Bulk Synchronous Parallelism</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multi-Processor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>FP</td>
<td>Floating Point</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple-Instruction, Multiple-Data</td>
</tr>
<tr>
<td>MRI</td>
<td>Magnetic Resonance Imaging</td>
</tr>
<tr>
<td>ODE</td>
<td>Open Dynamics Engine</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SAD</td>
<td>Sum of Absolute Differences</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single-Instruction, Multiple-Data</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>VIS</td>
<td>Visualization, Interaction, Simulation</td>
</tr>
<tr>
<td>XPU</td>
<td>Flexible Processing Unit</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Trends in semiconductor manufacturing technology provide new opportunities. The increased transistor counts due to Moore’s Law scaling enable computer architects to build chip with extremely high compute density, potentially providing GFLOPS of raw performance for every square milimeter of die area. In turn, the availability of availability of a dense computing substrate enables researchers and computer programmers in myriad fields to create new applications. These applications come from large, commercially relevant domains such as graphics and physics for gaming and interactive simulation, to data analysis for oil and gas exploration, scientific computing, 3D modeling for CAD, signal processing, digital content creation, and financial analytics. Applications in these domains can benefit tremendously from architectural approaches that provide higher performance through parallelism.

The new opportunities, however, are matched by the challenges of existing computer architectures. Traditional CPU architectures, even including multicore architectures, are restrained by the legacy of single-threaded applications. By market necessity, they require large, complex-cores to provide high performance for sequential applications. As a result, they cannot be architected to maximize throughput for emerging application domains. On the flip side, traditional accelerator architectures such as GPUs are heavily optimized for specific applications. They include specialized hardware that is deemed essential for an existing market, such as real-time graphics, but inhibits the ability to provide maximal performance
on emerging applications. Furthermore, the emerging applications tend to utilize algorithms that are far less structured than those targeted by traditional accelerators, such as graphics. As a result, traditional accelerators lack the support for features such as efficient synchronization which are necessary to enable the emerging applications.

Computer architects have responded to the challenges by introducing various accelerator architectures which seek to target emerging applications. The emerging class of accelerators is an evolution of traditional fixed-function, special purpose accelerators. Examples include the Cell processor from IBM, PhysX processors from AGEIA, TILERA, and numerous others. Even GPUs themselves are gaining greater support for general purpose programmability. These are not true general purpose processors, yet neither are they special purpose, but rather a class in between. We refer to the notion of a general class of accelerators as an “xPU” (as do others [1]).

Because of its emerging nature, the computer architecture community lacks a unified, broad-based understanding of the class of xPU accelerators. In this thesis, we seek to provide this understanding. We examine the design tradeoffs for this emerging class of xPU accelerators. We examine the tradeoffs between cost and performance, in terms of area and power. We examine these tradeoffs at all levels of the abstraction hierarchy of computing, ranging from the applications down to physical design of the hardware.

We first examine the application space for accelerator architectures. In this thesis, we focus on applications broadly associated with visual computing. Visual computing deals with the processing, rendering, and modeling of visual information. It includes graphics rendering, as well as video processing, computer vision, imaging, tracking, and physics simulation. To support our exploration, we de-
develop a benchmarking suite called VISBench (Visual, Interactive, Simulation Benchmarks) to serve as an experimental proxy for the visual computing domain. VISBench consists of applications from both traditional and emerging areas of the visual computing domain. We start with applications that are tuned to run on existing machines, and develop a parallelization model for evaluating their performance on massively parallel architectures. We provide a characterization of the applications. VISBench is explored in detail in Chapter 2.

From the characterization of the applications, we propose a highly parallel, throughput-oriented meta-architecture for a visual computing accelerator. An accelerator is a co-processor that allows the CPU to off-load and accelerate compute intensive work. In the accelerator model, the initial portion of the application consisting of I/O and branchy, sequential code is executed by the CPU. When the application reaches a parallelizable, compute intensive phase, the work is transferred over to the accelerator. Our proposed accelerator is built around a large array of simple compute cores, connected by a minimal on-chip network to a shared cache and a high bandwidth memory system. This meta-architecture is suitable for a far more general application space than existing accelerators such as GPUs, yet is not optimized to support the full range of workloads general purpose architecture. The meta-architecture is further described in Chapter 3.

Given the meta-architecture, we must still deal with a number of important architectural issues. To evaluate tradeoffs between different design points, we develop a model to estimate the chip area consumed by different architectural design points. In addition, we develop a simulation methodology to determine the performance achievable by those design points. We combine these models in order to solve an optimization problem: given a fixed area budget for a chip,

---

1Our VISBench should not be confused with NCSA VisBench, an unrelated application for analyzing remote CFD simulations.
determine the architecture that maximizes performance. Our methodology is detailed in Chapter 4.

An important high-level architectural issue is the memory model. In Chapter 5, we propose a hierarchical, cached memory model. We examine the data communication and synchronization patterns in VISBench, and propose mechanisms to handle them. We determine that VISBench, unlike raster graphics, does need support for fine-grained synchronization, but that the applications are insensitive to its performance. We propose a model with a non-coherent shared address space, and primitives to handle synchronization using the shared cache.\(^1\)

We examine the execution model for the cores within the accelerator. We examine the tradeoff between the area savings offered by a SIMD execution model versus the flexible control flow enabled by a MIMD model. We find that for some applications, SIMD execution provides a major performance per area benefit, but other applications’ control flow divergence results in an even greater performance loss. We perform a further analysis of one of our benchmark applications to evaluate ways to mitigate the performance losses due to control flow. The SIMD execution model is examined in Chapter 6.

In Chapter 7 we examine the performance and area tradeoffs of the core architecture. We examine design considerations such as superscalar execution, dynamic scheduling, multi-threading, and vector instructions. Furthermore, we examine the tradeoffs involved in sizing the private caches for each core. Subject to the constraints of the code we are able to generate, our major conclusions are as follows: There exists enough low-hanging instruction-level parallelism to profitably support dual-issue compute cores. Given the significant area overhead of word-level SIMD execution, supporting these instructions is not worthwhile considering their low frequency in most VISbench apps. 2-way multi-threading is often worth

\(^{1}\)This work was done in conjunction with the Rigel Project.
the extra area overhead; 4-way is not.

In Chapter 8, we further extend our study to examine the lower level micro-architectural tradeoffs in the core design. We extend our area model with a more detailed one that takes into account the effects of physical design through a synthesis flow. We also extend our performance model to take advantage of statistical sampling, in order to efficiently evaluate the large space of low-level design choices. We use the extended area model and statistical sampling to evaluate the effect of pipeline organization on performance, considering both IPC and clock speed, and co-optimizing architecture, microarchitecture, and physical design.\footnote{This work was done in conjunction with Omid Azizi and Mark Horowitz at the Stanford University department of Electrical Engineering.}

Given a model that accounts for the effects of physical design, we can evaluate tradeoffs not only between area and performance but also between power and performance. Power consumption is a first-order design constraint for large chips, including accelerators. In Chapter 9, we examine the effect of power constraints on the design space.

The discussion in Chapter 10 pulls together our results, and examines the limitations of our study. Our studies point toward a particular architecture that make sense for an xPU targeting the application areas represented by VISBench for any given set of power and area constraints. Clearly, like most experimental studies, our results must be viewed in light of our assumptions; due to the wide variation in the overall design space, one can potentially arrive at different conclusions with different assumptions. Nonetheless, the results in this paper serve as a starting point for investigation of accelerator tradeoffs and relative trends are likely to hold even with different assumptions.

Chapter 11 examines prior work in the area. It examines previous efforts at benchmarking visual computing and related application domains. It examines...
previous proposals for accelerator architectures. It also examines prior efforts at design space exploration, including early work on area-efficient and power-efficient parallel architectures, as well as related attempts at co-optimization of architecture and circuit-level design.

Finally, Chapter 12 provides concluding remarks and summarizes our contribution. This thesis aims to provide 3 enduring contributions:

1. The VISBench benchmark suite of applications from visual computing domains.

2. A process for evaluating design tradeoffs at multiple levels of the abstraction hierarchy.

3. An xPU architecture lying between special and general purpose architectures that can provide high performance for a broad set of visual computing applications.
We broadly define Visual Computing as application domains associated with the processing, rendering, and modeling of visual information. Applications in this domain represent a class of performance drivers for consumer and high-end computing. Higher baseline performance on such applications results in greater functionality and value to the end user. In many cases, the desire is to achieve interactive rates on certain applications (for example, video games require rendering and simulation rates of 30 frames per second and above).

Due to their visual nature, these applications tend to have considerable data-level parallelism. One can view visual computing applications as those that naturally map onto the GPU roadmap. As GPUs become more programmable, questions arise of what other applications can be mapped onto GPU architectures and how GPUs should be rearchitected to address the needs of the broader application base. We refer to this general acceleration architecture as an xPU.

In this Chapter, we present a benchmark suite to serve as an experimental proxy for visual applications. We present a description of each of the applications we include. We describe how the applications were parallelized for use in our studies. Finally, we present a characterization of the applications.
2.1 VISbench Applications

To address the question of xPU architecture, we create an experimental benchmark suite consisting of a sampling of visual computing applications. We selected open-source applications that have some relevant deployment in commercial products.

In VISBench (Visual, Interactive, Simulation Benchmark Suite) we cover classic visualization application areas, such as high-quality graphics rendering (Blender) and lighting (POVRay), and also video encoding (H.264), applications that are in commercial use today but which also have a continual need for improved throughput. We also cover emergent applications, such as interactive dynamics simulation (Open Dynamics Engine), computer vision (OpenCV) and high quality medical imaging (MRI), applications that are made possible by the widespread availability of low-cost, high-performance xPU architectures. VISBench applications are described below.

The space of visual computing is large and diverse. Naturally, our selection of open source benchmarks is only a sampling, and serves as a proxy for the commercial applications in this space. Moreover, we are considering applications that were not originally written with massively parallel accelerators in mind, and as such do not cover the whole set of architecture-specific optimizations proposed in the GPGPU community. Nonetheless, VISBench allows us to analyze a broad and relevant range of visual applications in order to provide insight into design choices for future accelerator architectures.

2.1.1 Scanline rendering: Blender renderer

Blender [2] is a free-software animation and 3D modeling program. We include Blender’s internal 3D renderer as a VISBench application.
Blender implements solid scanline rendering, similar to rasterization algorithms used for real-time 3D graphics, though it is far more flexible and aimed at higher quality off-line rendering. Blender’s rendering algorithm is broadly similar to algorithms such as REYES [3], used by RenderMan, which is commonly used for cinematic-quality rendering of movies, though the latter is more complex and offers far greater programmability. Blender can also perform ray tracing, which we do not use.

The scanline rendering algorithm is embarrassingly parallel. The shading of each pixel can be computed independently of other pixels. Blender supports multithreaded rendering. Each render can be broken up into tiles, with each tile rendered in a separate thread. Each tile can be as small as a pixel quad (4 pixels).

The basic rendering pipeline for Blender consists of a hierarchy, from the full image to tiles to pixels and finally sub-pixels. The core of the rendering pipeline is the Tile Processor. Within each tile, the renders go over pixels per scanline and gather visibility information. Sub-pixel samples are generated, and shading is performed for each sample. The shader determines the color value for each sample, through passes for computing textures, lighting, shadows, and a variety of other effects.

For our benchmark input, we use a complex image of a hairball with a room as the background, rendered into a 640 × 480 image. The rendered image is show in Figure 2.1. Using the parallelized version of Blender, this image can be rendered by up to 76,800 threads, ranging in length from 500K to 2.4M instructions. The complete render takes roughly 60B instructions.
2.1.2 Ray tracing: POVRay

An alternative to scanline rendering is ray tracing. Ray tracing is one of a number of highly compute-intensive global illumination methods; that is, methods that render not only direct lighting but also reflection, refraction, and diffusion. Global illumination methods can produce highly realistic lighting effects, but at a much greater computational cost than scanline algorithms that do not consider global illumination.

The ray tracing algorithm mimics optics by tracing the path taken by rays of light. One or more rays is cast out per pixel, away from the eye. Whenever the ray intersects an object, one or more recursive rays are sent out away from the point of intersection, up until some limit or the ray reaches a light source. This recursive process is used to generate a color value for each pixel.

The ray tracing algorithm is embarrassingly parallel. Each pixel calculation is independent of every other pixel. Although different pixels may intersect with the same object, requiring shared data structures, they need not write to any shared data.

POVRay is a widely used ray tracer, and is also an FP benchmark in SPEC.
CPU2006. We use a modified version of POVRay 3.6.1 which is single-threaded. Versions of POVRay from 3.7 (not released at the time of writing) support multithreaded rendering.

As input we use a scene containing a chessboard with a number of glass pieces. The rendered scene is shown in Figure 2.2. This input, which is included as part of the POVRay release, contains a large number of complex reflections and refractions. The scene is rendered at $384 \times 384$.

In VISBench, POVRay is decomposed hierarchically, with one thread for rendering each row, which in turn spawns off one thread for rendering each pixel in the row. A full render produces 147,456 threads, ranging from 98K to 4M instructions.

2.1.3 Video encoding: H.264 SAD-based motion estimation kernel

Modern video encoders take a sequence of raw images and compress them, typically using a motion compensation-based encoding technique. There are a variety
of video codecs currently in use, including H.262, H.263, and MPEG-2. An emerging codec for high definition video content is H.264, also known as MPEG-4 AVC.

The compute-intensive portion of the encoding pipeline is motion prediction, accounting for over 90% of execution time even for standard definition content. Motion compensation is a technique for describing a macroblock of pixels within an video frame relative to a block of pixels in a reference frame. The motion prediction process consists of determining a suitable motion vector for each macroblock in the current frame into a set of reference frames.

The process of finding motion vectors is called motion estimation. Motion estimation involves a search among the space of possible motion vectors for the optimal one, the one that maximizes compression. Maximum compression is achieved by balancing the number of bits needed to represent the motion vector itself versus the amount of block prediction error. While there is no algorithm for finding optimal motion vectors in polynomial time, there are fast algorithms to find near-optimal vectors.

The most suitable motion vector is one that minimizes residual encoding information, and this is often done by computing the sum of the absolute differences (SAD) between each pixel in the translated macroblock in the reference frame versus the corresponding block in the current frame. The SAD computation is fast and simple, and easily parallelized. Hence, using SAD to compute errors allows a large search space for finding good motion vectors. VISBench includes a motion prediction kernel that computes SAD values between macroblocks in two video frames and then checks the results.

The SAD kernel is embarrassingly parallel and can be parallelized in a number of ways. We use a standard CPU version of the kernel, written in C. We handle each macroblock in parallel, and within each macroblock all of the SAD comparisons for a particular row. One HD image has 3600 macroblocks, and a vertical
search range of 33, so we have 118,800 threads all roughly 75K instructions long, for a total of 8.9B instructions.

Note that commercial versions of the H.264 encoder perform motion searches that are optimized and predictive, and perform less work than the full-search technique we model, with a slight loss of compression [4].

2.1.4 Dynamics simulation: ODE PhysicsBench

Simulation of rigid body physics is useful in a variety of contexts, including scientific applications, engineering mechanical systems, and for adding interactivity and realism to video games.

Open Dynamics Engine (ODE) is a free library for simulating articulated rigid body dynamics. PhysicsBench is a suite of physical simulations built using the ODE library. A parallel implementation of PhysicsBench and ODE is described in Yeh et. al. [5].

The structure of computation in ODE is very different from graphics rendering and SAD computation, which are embarrassingly parallel if properly coded. Physics computation exhibits distinct phases with distinct levels of available parallelism. Broadly, one step of physics simulation consists of collision detection, followed by a solver which determines the forces and motion of interacting objects.

Collision detection is divided into two phases. The first phase, *broad-phase*, culls the O(n^2) space of possible collisions by determining whether bounding boxes around objects intersect. This phase can be parallelized, but only minimally so because it requires updates to shared data structures. The second phase, *narrow-phase*, determines which pairs of possible collisions is an actual collision. This phase is massively parallel since each pair is independent.

After determining which pairs of objects are interacting, the next phase, is-
land generation, groups the objects into islands of interacting objects. This phase is essentially serial. The next phase is island processing, where the motion of each object in each island is calculated. Island processing is a parallel process, where each island is independent, and further the constraint solver for each island contains independent iterations of work. Finally, PhysicsBench contains a cloth phase, which is highly parallelizable. The cloth objects are processed independently, and each cloth object is itself processed in parallel.

Figure 2.3 summarizes the phases of computation in our test simulation.

Our test simulation from PhysicsBench contains 1608 objects, 933 of them as boxes in breakable walls. Collision detection generates 61 threads ranging from 390K to 2.3M instructions, the constraint solver generates 192 threads ranging from 127K to 282K instructions, and cloth simulation generates 7000 threads ranging from 12K to 525K instructions. One timestep takes roughly 900M instructions.

2.1.5 High quality MRI reconstruction kernel

In magnetic resonance imaging (MRI), data from a magnetic resonance scan is reconstructed from a set of data points in the spatial frequency domain into a 3D image of the scan target. Conventionally, this is done using an FFT, which requires the data points to be on a Cartesian grid.

However, due to the physics involved, better images can be obtained by using a non-Cartesian scan trajectory; that is, sampled data points are not taken on a Cartesian grid. Reconstructing images from non-Cartesian data points is a difficult computational problem. Typically, these non-Cartesian data points are interpolated onto a Cartesian grid, and the image is then reconstructed using a fast Fourier transform (FFT). A better image can be obtained by applying a
Figure 2.3: Phases of the physics computation. The parallel phases are shaded.
non-uniform Fourier transform (non FFT) directly to the original data points. However, on current machines this process can take many hours. A CUDA implementation of non-Cartesian MRI reconstruction is described in Stone et. al. [6]. We base our study on the CPU version.

The main computation in an image reconstruction from the non-Cartesian data consists of computing two vectors, Q, given by

\[
Q(x_n) = \sum_{m=1}^{M} |\phi(k_m)|^2 e^{(j2\pi k_m x_n)}
\]

and \(F^H d\), given by

\[
[F^H d]_n = \sum_{m=1}^{M} \phi^*(k_m)d(k_m)e^{(j2\pi k_m x_n)}
\]

The computation of Q is performed offline, based only on the position of each sample. The computation of \(F^H d\) depends on the actual data values obtained from the scanner. Both computations are nearly identical; we use the \(F^H d\) computation as a benchmark. We compute all \(F^H d\) values in parallel, with each independent thread computing the value for different elements. A full MRI image requires a square convolution with several hundred thousand data points. In our study we simulate a 4096 by 4096 convolution which produces 4096 threads, each 240K instructions long.

2.1.6 Computer vision: OpenCV based face detection

Computer vision (CV) deals with systems for obtaining information from images. CV algorithms analyze images or video in order to recognize objects and reconstruct models of the scene. One use of CV is detection and recognition of human faces, important in such varied applications as autofocus in digital cameras, bio-
metrics, and video surveillance.

OpenCV is an open-source library containing many basic algorithms used in computer vision. FacePerf [7] is a set of benchmarks for evaluating face recognition performance. As part of VISBench we use a modified version of the the OpenCV Haar-based face detector that is included in FacePerf.

Haar Classifier-based face detection scans an image and returns a set of locations believed to be faces. It works by searching for facial features within rectangular subimages of the original. The algorithm uses an adaptive boosting technique to prune the search space. The algorithm is detailed in Viola and Jones [8].

The face detector can easily be parallelized, with subimages being processed in parallel. Pruning the search space results in some irregularity between the parallel tasks, but still allows for a large number of tasks. For this paper we run the face detector on a 2 megapixel image with over a dozen faces at varying angles and distances.

The face detector runs in repeated phases, each of which in turn has 3 parallel sub-phases: a short phase to set up the classifier, with 22 threads ranging from 1.3K to 95K instructions, and two phases running the classifier, each with roughly 450 threads ranging from 43K to 975K instructions.

2.2 Parallelization Model

The VISBench applications are all written for CPU execution. Some of the applications are written as single-threaded code, albeit implementing parallelizable algorithms. Other applications are multithreaded using Pthreads or OpenMP, in order to take advantage of dual or quad core machines. In each case, we need to expose the parallelism in the application in a manner that is appropriate to a
Figure 2.4: Parallelization model used for studies. Each iteration of a parallel loop is one thread.

design space exploration examining massively parallel architectures.

To enable our exploration, we first remove any existing parallelization from the application. We then parallelize the applications by placing annotations around the parallel loops. These annotations, used in the same manner as OpenMP pragmas, consist of dummy function calls to mark the beginning and end of parallel sections, and of individual threads. The annotations do not affect the execution of the program if it is run on an actual machine. However, they are detected by our simulation infrastructure and used to model the performance of the parallel implementation.

Figure 2.4 shows how we use annotations to indicate parallel sections of code. In our exploration, we will assume that sequential portions of the application
will execute on a CPU. When we reach a section of parallel code, we call the `start_outer_parallel_loop` function. Within the parallel loop, the beginning and end of each iteration (i.e. thread) is marked with calls to `start_outer_parallel_iter` and `end_outer_parallel_iter`, respectively. These iterations can be run in parallel.

Some of our benchmarks, in particular POVRay, ODE, and the H.264 SAD kernel, have nested parallel loops. In this case, we gain additional parallelism at the inner loop level as well. We indicate parallel inner loops in a similar manner to outer parallel loops, as shown in Figure 2.4.

Our parallelization model is fundamentally similar to that of OpenMP. Our model, like OpenMP, is based on the parallel execution of loop iterations. Furthermore, like OpenMP our model uses annotations. Whereas OpenMP uses pragmas, our model uses dummy function calls. The major difference between our model and OpenMP is that OpenMP is a fully featured programming model for parallelizing code on real machines, while our model is used to perform an abstract parallelization of the code to enable architectural studies through simulation.

Other parallelization models support a variety of parallelization styles besides loop parallelism. Pthreads is a low level threading interface for Unix. Using Pthreads, a skilled programmer can parallelize a program into threads with support for nearly an arbitrary parallelization. Cilk [9] is a higher level threading interface that can similarly support arbitrary parallelization. Other programming interfaces provide support for much more constrained parallelization. OpenCL [10] and CUDA [11] are extensions of C for programming GPUs. They support a much more restrictive model where the communication and synchronization between parallel tasks is very limited, and where parallel tasks cannot spawn additional parallel tasks. The arbitrary threading afforded by models such as Pthreads and Cilk is unnecessary for our studies using VISBench, while the myriad restrictions of the memory model in CUDA and OpenCL would not be suitable for our appli-
2.3 Characterization

In this section we provide a characterization of the benchmarks in VISBench. We want to examine characteristics of the benchmarks that have architectural implications. We examine the types of operations that each of the benchmarks uses most frequently. In addition, as we are focusing on parallel architectures, we examine characteristics of the benchmarks that will affect parallel performance scaling.

2.3.1 Instruction mix

An important consideration in designing an architecture is support for the specific operations that are frequently used by the target applications. In Figure 2.5, we break down the dynamic operations executed by each benchmark into categories. We observe a few trends. First, all of the applications are dominated by ALU and memory operations, with less control flow. However, control operations (branches, jumps) make up a larger proportion of the instruction stream than we find in traditional numerical workloads.

We note that every application except the H.264 SAD kernel makes use of floating point (FP) operations. In fact, 4 of the applications rely very heavily on floating point operations. This is not surprising for visual workloads. In Figure 2.6, we categorize the floating point operations. Add/sub includes simple floating point arithmetic operations. These dominate all of the benchmarks except MRI. All of the benchmarks also make frequent use of FP multiplies, especially MRI. The category of complex FP includes operations such as divide, reciprocal, square root, exponentiation, and trigonometric functions. In practice,
Figure 2.5: Breakdown of operations in VISBench applications.

Figure 2.6: Breakdown of floating point operations.
these operations can be implemented as multiple simple operations. Although complex operations make up a small fraction of the total FP operations, their complexity means that they will take up a much larger fraction of execution time. MRI in particular makes heavy use of trigonometric functions to compute $F^b d$. ODE, Blender, and POVRay also use trigonometric operations to deal with round objects and to compute transformations in 3 dimensions.

2.3.2 Performance scaling

In visual computing applications, there is a correlation between the complexity of the application data set and the user experience. For instance, the quality of a rendered image can be increased by rendering more polygons, and the fidelity of a physical simulation can be improved by modeling more objects. In this sense, VISBench applications are appealing targets for vendors of high-performance hardware, as these applications can take advantage of rapidly scaling computer performance.

We anticipate that future computer architectures will provide a much greater speedup through parallelism than through serial performance. Each of the VISBench applications contains short section(s) of serial code in addition to the time-intensive parallel portions. Under Amdahl’s law, the speedup of a parallel application is given by:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{parallel}) + \frac{Fraction_{parallel}}{Speedup_{parallel}}}$$

which is limited to $\frac{1}{1 - Fraction_{parallel}}$ even when the parallel speedup is infinite. Figure 2.7 shows the fraction of dynamic instructions occurring in startup code, parallel code, and in serial code phases in between parallel sections (found in ODE and Facedetect). In all VISBench applications, we find that at least 84%
of execution time on a serial machine would be spent in parallel portions. In most of the applications, the parallel code regions would consume more than 95% of the execution time, demonstrating that large speedups through parallelism are achievable.

In Figure 2.8, we examine the performance scaling of the parallel code sections of the VISBench applications. These scaling figures were generated by simulating the parallel portions of VISBench on accelerator architecture configurations with varying numbers of single-issue in-order cores. From the figure, we can see that all the benchmarks scale perfectly linearly up to 64 cores, where Facedetect’s scaling slows to a lower rate, while other benchmarks scale linearly up to 256 or even 1024 cores. Combining the scaling of the parallel sections with Amdahl’s law leads to the estimated scaling shown in Figure 2.9, assuming no performance enhancement in the sequential code.

Furthermore, and perhaps more important, all of these parallel sections scale linearly, or faster, with the data set. The results of Figure 2.9 are generated using
Figure 2.8: Performance scaling of the parallel sequences in VISBench.

Figure 2.9: Estimated performance scaling of the applications in VISBench from applying Amdahl’s Law.
the simulated VISBench inputs, which are small. However, as the complexity of the visual simulation grows, as is the desire from generation to generation, the parallel workload grows rapidly. For instance, in graphics rendering (either scanline or ray traced), the total computation time of the render scales with both the number of pixels being rendered and with the number of objects in the scene. There is a similar growth in computational requirement for other areas represented by VISBench. In fact, parallel sections are either $O(n^2)$ or quasi-$O(n^2)$\textsuperscript{1} in the number of primitives being processed (such as pixels, polygons, objects, etc). Moreover, the serial code sections scale at most linearly with the data set. Hence, these workloads benefit from Gustafson’s Law, which states that any sufficiently large problem can be efficiently parallelized. This important to note as it implies that an accelerator architecture can be scaled with Moore’s Law through parallelism and still provide value to the same application.

2.3.3 Thread length

From Figure 2.7, we see that parallel sections of code make up the vast majority of the dynamic instruction stream, and hence satisfy Amdahl’s law condition for attaining high parallel speedups. However, in practice the amount of parallel speedup that can be achieved may also be limited by the number of threads within the parallel sections and the length of those threads.

Table 2.1 shows the mean thread lengths for each of our benchmarks. From the table, we see that the mean thread lengths for different benchmarks range from the tens to the hundreds of thousands of instructions. In comparison to threads run on multicore CPUs, the threads in VISBench applications are fine grained, enough so that each parallel phase can have several hundred or even several thousand tasks running in parallel, even on the moderately sized inputs.

\textsuperscript{1}$O(n)$ with significant constants or worst case $O(n^2)$
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Thread Length (K instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blender</td>
<td>525</td>
</tr>
<tr>
<td>Open Dynamics Engine (ODE)</td>
<td>44.7</td>
</tr>
<tr>
<td>MRI</td>
<td>237</td>
</tr>
<tr>
<td>POVRay</td>
<td>88.3</td>
</tr>
<tr>
<td>Facedetect</td>
<td>405</td>
</tr>
<tr>
<td>H.264</td>
<td>77.2</td>
</tr>
</tbody>
</table>

Table 2.1: Mean thread lengths for our benchmarks.

To gain further insight, we examine the distribution of thread lengths. Figures 2.10 through 2.15 show the CDF of the thread lengths for each of the benchmarks. In Figure 2.11, we see that all threads are the same length of roughly 240K instructions. In Figure 2.10 and Figure 2.15, we see that most of the threads in Blender and the H.264 SAD kernel are the same length, with a few threads that run longer. In the case of H.264, this is due to threads handling macroblocks near the edges of the frame, which handle the special case of searches that may go outside the frame by padding the image. In the case of Blender, the threads rendering the hairball take roughly 4 times as long as threads rendering the background.

The CDF of thread lengths for Facedetect is shown in Figure 2.12. From the Figure, we can see that the length of a thread varies widely, a direct result of the underlying algorithm. Subimages that are filtered out by early stages of the Haar

Figure 2.10: CDF of thread length distribution for Blender.

used for our studies.
Figure 2.11: CDF of thread length distribution for MRI reconstruction.

Figure 2.12: CDF of thread length distribution for Facedetect.
Figure 2.13: CDF of thread length distribution for the different phases of ODE.
Figure 2.14: CDF of thread length distribution for POVRay.

Figure 2.15: CDF of thread length distribution for the H.264 SAD kernel.
cascade are processed by short-running threads, while subimages that are filtered out by later stages of the cascade or that return positive (i.e. contain a face) are processed by much longer running threads.

The different phases of ODE show very different behavior in terms of thread length. In the collision detection phase (narrow phase), thread lengths are spread fairly evenly throughout the range. In the solver (island processing) phase, most of the threads are roughly 170K instructions long, with a few threads varying somewhat. In the outer loop of the cloth phase, each thread is basically the serial portion of the solution for one piece of cloth. Our scene has cloth pieces of two different sizes, and hence the two possible thread lengths. The inner loop of the cloth phase is the parallel solver for each piece of cloth; here most threads are short but a few run very long.

POVRay shows the same behavior as the cloth solver, with most of the threads running 25K to 35K instructions but a few threads taking much longer. In this case threads computing background pixels and pixels with just a single object intersection are computed quickly, while pixels hitting complex reflective and refractive objects take an order of magnitude more time to compute.

The examination of thread lengths reveals that the VISBench applications have sufficiently short threads to allow us to exploit a massive amount of parallelism, and further that even the longer threads (a few million instructions) can be computed within a timestep. However, the distribution of thread lengths suggests that some applications have a great deal of data-dependent control flow. The implications of this observation will be discussed in Chapter 6.
2.3.4 Data communication

Another important factor in determining the scalability of an application is the amount of data communication between threads. Shared memory accesses in a parallel program need to use shared resources on the chip, and may need to be synchronized. This can place an upper bound on the achievable parallelism.

In Figure 2.16 we break down all of the dynamic loads into private and shared. Private loads are loads to data which has been generated within the same thread. In most of the VISBench applications such loads account for the majority. The second category is shared reads across barriers. These are loads to input data, that is, data generated prior to the start of a parallel phase of computation. In this case, there is a synchronization barrier (or perhaps several barriers) between the store that generated the data and the load that consumes it. As a result, though the value may need to be propagated across the chip’s shared resources, the loads themselves do not need to be individually synchronized with locks. All of the benchmarks have a large number of such loads, especially Facedetect. The final category is shared loads within a barrier. These are loads to data generated
Figure 2.17: Proportion of memory writes to shared versus private data.

by a store that may be in a thread running in parallel to the load. Such loads may require synchronization, but in VISBench they occur rarely if at all.

In Figure 2.17 we break down the dynamic stores into private and shared. Private stores are stores which generate values that are never accessed outside the same thread. Private writes account for the vast majority of stores. Shared writes across a barrier, analagous to shared reads across a barrier, are stores that produce an output value. Shared writes within a barrier are the stores which correspond to shared reads within a barrier, and may require synchronization with locks.

Chapter 5 revisits the data communication and synchronization needs of VIS-Bench and proposes mechanisms to address them.
In this chapter, we introduce an accelerator meta-architecture, a high-level architectural framework that defines the design space. We describe the accelerator model of computing. We describe the major, high-level components of the xPU. Finally, we break down the design space that we will explore in later chapters.

3.1 Acceleration Model

This study examines the accelerator co-processor execution model. In this model, we are assuming a heterogeneous computing system, containing different types of computing units for different portions of the computation. More specifically, we are assuming that the system includes a CPU, which contains compute cores designed for high performance on single-threaded, general-purpose code, and an accelerator (xPU) designed for high performance on certain long-running, compute-intensive sections of code. The general notion of the accelerator model is shown in Figure 3.1.

The CPU in Figure 3.1 is used to run system code and to start up the application. It may also be used to run sequential phases of code which do not execute efficiently on an xPU. The CPU controls the operation of the xPU, and maps onto the xPU the long-running, compute intensive code sections which are highly amenable to parallelization.

In this thesis, we envision the xPU as separate chip from the CPU, connected
Figure 3.1: The CPU/accelerator co-processing model.

```plaintext
init_code()
input_data()
accelerator_kernel()
... ...
execute_kernel()
... ...
output_data()
... ...
```
to the system via a commodity interconnect such as PCI Express or HyperTransport. We envision the CPU host and the xPU having disjoint memories from the programmer’s perspective and must be managed via explicit DMAs. Further, we envision the CPU and xPU as having potentially disjoint instruction set architectures. This view of the xPU and CPU as disjoint gives the system architecture a clean division and allows us to focus on the design of one without extensive consideration of the other. In thesis, we do not deal with the issues involved in managing transfers between the CPU and xPU, or of the tradeoffs involved in a close versus a distant coupling between the two. A variety of strategies have been deployed over time on GPUs and on Cell for amortizing the latency, reducing the bandwidth, and reducing programmer burden [12] associated with the separated memories. Nevertheless, our insights into the design of the xPU retain their validity even in the case where the CPU and accelerator are closely coupled.

3.2 Basic Architecture

Given that we have defined the role of the xPU in the acceleration model, the question remains as to what needs to go into the chip. At the most basic level, the xPU needs to contain the elements of any computer architecture, in other words a compute portion, a memory system, and in our case a mechanism for communicating with the rest of the system.

Figure 3.2 expands on Figure 3.1, and shows the basic architecture of the xPU. The xPU consists of a compute array with a large number of cores, connected to a cache. The cache is in turn connected to a high-bandwidth memory system. Finally, the xPU contains a controller that manages communication with the rest of the system as well as manages the initiation of tasks on the accelerator.

As pointed out, we envision the xPU as disjoint from the CPU. To make the
notion concrete, Figure 3.1 shows the xPU being connected to the rest of the system via a standard interface such as PCIe or HyperTransport, using DMAs to transfer data between the host and accelerator, although the workings of this interface are beyond the scope of this thesis. The xPU uses the controller core to handle data transfers and initiate execution on the accelerator.

The compute fabric of the xPU consists of an array of small cores. Each core consists of an execution pipeline capable of executing an instruction stream. Because we are focused on floating-point intensive visual computing applications, we assume each core has a floating point execution unit, along with integer units, register files, and small instruction and data caches. These cores are linked into a global on-chip interconnect.

The on-chip interconnect connects the compute array to the global cache (gcache). The gcache is a large, multi-banked cache accessible to all the cores in the array. This arrangement allows for a very large gcache bandwidth. The purpose of the gcache, in this case, is to multiply the bandwidth of the main memory, with the multiplier being the inverse of the miss rate of the cache.

The gcache is connected to a high-bandwidth interface to off-chip memory. Several banks of gcache are connected to a single memory controller, which controls a single high-bandwidth DRAM channel, striping the memory space across the DRAM channels.

3.3 Design Space

Given the accelerator model and the basic architecture, we can now fill in the details of the xPU design. The meta-architecture provides us a framework for exploring a broad design space. This space includes high-level design issues, such as the memory and execution model. It includes the architecture and microarchi-
Figure 3.2: Basic architecture of the accelerator.
tecture of the compute array, the global cache, and the memory system. Finally, it includes the circuit and physical design of the chip.

Certain important design considerations are beyond the scope of the studies in this thesis. For instance, we do not address any details on the controller core. Moreover, we do not deal with issues due to the connection between the accelerator and its memory with the rest of the system. These items are fundamentally performance overheads, and we assume they can be designed in a way as to minimize their impact. Similarly, we do not explore specific methods for doing thread scheduling, assuming that it can be done with minimal overhead. Previous work [12] [13] [14] has shown that this assumption is justified. We return to some of these issues in Chapter 10.

An important design consideration is the memory model. Multicore CPUs typically use a single, shared memory space. On the other hand, a number of existing accelerator architectures, including Cell and GPUs, use multiple memory spaces and scratchpad memories. Design decisions regarding the memory model include single versus multiple address spaces, the design of the memory hierarchy, and the bandwidth requirements. A related issue is the mechanisms provided for synchronization and communication. The xPU could include zero support for any sort of synchronization all the way up to support for fine-grained locking at a global level. Similarly, it could include support for absolutely no inter-thread communication (except through the CPU host), minimal support through memory, or extensive, high-bandwidth communication mechanisms on-chip. In Chapter 5, we examine the tradeoffs all of these design decision make on the execution of the VISBench applications.

The execution model of the compute array is another portion of the design space. In GPUs, the compute array is arranged in clusters of pipelines which execute in SIMD lockstep. In other architectures, such as TILERA or multicore
CPUs, the compute array consists of discrete cores which execute in a MIMD fashion. The SIMD approach allows the architecture to provide a higher density of peak throughput, at the cost of a loss in flexibility in execution between threads. The tradeoffs of these approaches are examined in Chapter 6.

A further component of the design space for the compute array is the high-level architecture of the cores. The high-level, or *macro*-architecture, defines the *algorithm* of the core. It affects the way the core executes its instruction stream. Elements of the macro-architecture include features of the overall pipeline design, such as superscalar issue and dynamic scheduling. They include decisions that are exposed to the programmer, such as multithreading, or instruction set changes like support for vector ALUs via SIMD instructions, as well as special floating point instructions for the complex functions that occur frequently in visual computing workloads. The tradeoffs in the core-level macro-architecture are examined in Chapter 7.

A level beneath the macro-architecture is the core *micro*-architecture, those aspects of the core design that may affect performance but not the overall mechanism by which the instruction stream is executed. Microarchitectural design choices include pipeline depth and functional unit latencies. They include predictor sizing. Also, as we consider the lowest level caches to be part of the core, they include the size and latency of the private first level caches. The micro-architecture is closely coupled to the implementation. The implementation includes the logical design of the micro-architecture. It also includes logic styles, and the circuit design. Finally, it includes the physical design, including circuit sizing and latency. Chapters 8 and 9 examine the tradeoffs of micro-architecture and implementation, including IPC, clock speed, area, and power consumption.

Table 3.1 summarizes the design space for the xPU.
<table>
<thead>
<tr>
<th>Design level</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory model</td>
<td>memory hierarchy, caching, synchronization mechanisms, communication support</td>
</tr>
<tr>
<td>execution model</td>
<td>SIMD/MIMD</td>
</tr>
<tr>
<td>chip-wide architecture</td>
<td>global cache bandwidth, memory bandwidth</td>
</tr>
<tr>
<td>core macroarchitecture</td>
<td>superscalar issue, dynamic scheduling, multithreading, instruction set</td>
</tr>
<tr>
<td>core microarchitecture</td>
<td>pipeline latency, functional unit latency, predictor sizing, L1 cache sizing, L1 cache latency</td>
</tr>
<tr>
<td>implementation</td>
<td>logical design, logic style, circuit design, circuit sizing, circuit latency</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of the design space.
In Chapters 5 to 7, we will be dealing with the macro-architectural optimization of the accelerator. This will include higher-level issues of memory and execution model, as well as the more concrete architectural design of the core and memory system. We want to search the architectural design space of the accelerator to maximize performance given an area constraint.

In this Chapter, we describe our methodology for evaluating macro-architectural tradeoffs. As part of our methodology, we have developed a performance measurement methodology based on simulation. In addition we have developed a model to compute the area cost of a variety of architectural features by mapping out the required hardware at a micro-architecture and then logical level and then determining the area cost of each component.

4.1 Optimization Problem

The architectural design of the xPU can be seen as an optimization problem. In an optimization problem, we want to maximize an objective function given some set of constraints. In a high-performance architecture, the objective is performance, and the constraint is to build a chip with as large an area and power budget as can be built and sold profitably in the given market segment. As a result, area is often the most important first-order constraint. Hence, we will focus on maximizing performance given an area constraint.
Power consumption is of course also a first order constraint. However, it is more difficult to model power consumption as we are exploring only the high-level macro-architecture, and as a result we will defer the consideration of power until after we have dived into the low-level micro-architecture and physical design. As we will ultimately show, power and area constraints are closely related.

In semiconductor manufacturing, the cost of a defect-free die is a function of the die size. The cost of a wafer is typically fixed, and as the die size increases, the number of chips that can be produced on a wafer decreases. Hence, the cost of a die increases. Moreover, due to the random distribution of defects, the cost of the die increases faster than linearly with the size. As a result, any given price target leads to a fairly sharp constraint in die size.

We want to study the optimization problem as it pertains to a modern manufacturing process and a typical price target for a high-end accelerator chip. Thus, we evaluate area based on circuits printed using a 65nm process, the most recent process generation for which extensive information was readily available. Furthermore, we set the area budget at 400mm$^2$, roughly the size of a high-end GPU or server CPU, and close to the maximum die size that can be supported by the consumer market.

The resultant design problem becomes: given a chip area budget of 400mm$^2$ in a 65nm process, how should we architect that area in such a way as to maximize performance.

4.2 Area Modeling Methodology

Modeling area is a challenge for architecture researchers. The most accurate way to model the area of a core is to design the core in RTL and synthesize, place, route, and optimize for speed, power, and area constraints. Unfortunately, this
method requires developing RTL for all of the various design possibilities, a time consuming endeavor, and is not ideal for a high level design space exploration.

A much simpler method used by previous researchers is to measure components of real designs using die photos or published values [15]. [16] extends this method by using analytical formulas. However, this method is restricted to the design assumptions made by the vendor of the baseline design.

We use a hybrid approach that allows us to evaluate a larger design space. We divide the components of the area cost into three groups: SRAMs, functional units, and pipeline logic. For most SRAMs, we compute area estimates using CACTI-6 [17]; however, CACTI is designed to evaluate large SRAM arrays, and so for some of the smallest SRAMs we count the number of bit cells and gates needed to implement the SRAM and multiply by published figures for the sizes of these structures [18].

In any area model, we need to base the area estimate on a specific micro-architectural design. The purpose of the hybrid area model developed here is to evaluate macro-architectural tradeoffs, so we implement a somewhat generic pipeline micro-architecture for each macro design point. Furthermore, at a macro level the performance metric is instruction throughput per clock cycle, while clock speed is more closely tied to the micro-architecture. We implement each of the micro-architectures in this section to achieve comparable clock rates around 1GHz. In the 65nm process technology, this is achieved by restricting each pipeline stage to 30-40 FO4 delays.

For ALU and FPU components, we use published synthesis results ([19], [20]). When synthesis results are not available, we obtain area estimates by designing the structures at the logic gate level and counting the number of gates, flops, and SRAM bits required in the implementation.

Finally, for remaining pipeline logic (e.g. muxes, PC selection logic) we imple-
ment the components at the logic gate level and again count the number of gates, flops, and SRAM bits.

We compose these components to model three different basic pipeline configurations: A 1-wide in-order pipeline, a 2-wide in-order pipeline, and a 2-wide out-of-order pipeline. These pipeline configurations represent the sort of architectures that are often considered for many-core designs because of their small area and high ratio of performance to area.

The 1-wide configuration consists of a standard 5-stage fully-bypassed, in-order pipeline with a static BTFN predictor. The stages are fetch, decode/register read, execute, dcache access, and writeback. We assume a non-blocking cache and a multicycle FP unit (5 cycle), and allow instructions to writeback out of order. We assume separate register files for integer and FP, each 32 entries times 32 bits.

The 2-wide in-order pipeline consists of 6 stages (the stages from the 1-wide plus an arbitration stage before execute) with a bimodal predictor. A second simple ALU is added to the execution stage (adder and logical operations, no shifter or multiplier). As in the 1-wide, there is only one FP unit.

The out-of-order pipeline consists of a 6-stage, 2-wide, out-of-order pipeline with a bimodal predictor. The stages are fetch, decode/rename, ROB issue, scheduling, execute, and retire. The pipeline is modeled roughly after the P6 microarchitecture. Execution resources remain the same as the 2-wide in-order. The ROB contains 24 entries, the scheduler 12 entries. We assume all of these cores run at 1GHz.

For each basic configuration, we examine the effects of additional performance enhancing features such as SIMD instructions with vector ALUs and fine-grained multithreading. We model the addition of 2-way and 4-way SIMD, 2-way and 4-way multithreading, and special purpose functional units for sine and cosine.

The incremental cost for multithreading is modeled by replicating portions of
the front-end stages (fetch, decode, rename, rob insert). In the front-end stages of the pipeline, we replicate the pipeline registers for each stage for each thread. Within each stage, muxes select the pipeline registers which will be read and updated, so we add a mux at the beginning and end of each stage. In addition, we replicate the register file, RAT, and PC for each thread.

The additional cost of word-level SIMD (a la SSE) is modeled by replicating execution units (to form a vector execution unit) as well as accounting for the increase in pipeline registers. No additional register files are added for SIMD, but read and write ports are scaled appropriately.

A hardware implementation of trigonometric and transcendental functions is described in [21]. The unit works by performing a Taylor Series approximation of the function, through a combination of table lookups, adds, and multiplies. We model this unit as consisting of a ROM for the lookup table, plus adders and multipliers, and a shifter. We model this unit having a delay of 6 cycles, pipelined with 2 stages such that operations can begin once every 3 cycles.

Table 4.1 shows area by major hardware function for each of the basic configurations. Frontend includes fetch and decode as well as scheduling and renaming. Other includes remaining logic and pipeline registers.

We make the simplifying assumption in this study that adding cores does not affect the incremental area consumed by the interconnect. Because we want to focus on the core and cache architecture in this study, we want to minimize the impact of the interconnect. Hence, we model the bandwidth limitations at the global cache level and assume that we can build a network, perhaps somewhat overprovisioned, that is capable of maximizing the utilization of the global cache. Because we hold the global cache bandwidth fixed, we would not need to significantly increase the area consumed by the interconnect in order to supply maximal gcach utilization to a larger number of cores.
Area, of course, is affected by more than just architecture. Physical design itself involves a large design space exploration, and a given microarchitecture can be implemented by widely varying layouts. Assuming that we use synthesis to generate the layout, sources of variation include wiring overhead due to place and route (since optimal place and route is NP-hard), choice of libraries, choice of logic gates, and padding due to DFM. We used a 33% overhead to encompass all of these, but in real designs the overhead can vary dramatically. In Chapter 7 we will assume a confidence interval for our area numbers of ±20%. This number was the empirical variation in area for layouts generated by the Synopsys Design Compiler in [22]. We will examine the design space for physical design in Chapter 8 and later, as we evaluate the lower level microarchitecture and circuit-level tradeoffs and also evaluate the effect of power constraints.

Of course, another potential source of inaccuracy in our model is error: a mismatch between the architectural and logic level design, uncertainty in the results from tools such as CACTI, and potentially inaccurate area numbers for our components. While we cannot provide a quantitative measure of these errors, we have made our best effort to ensure the correctness of our model. We cross-checked our area results against existing designs such as the MIPS 74K [23] and Tensilica 570T [24]. These commercial cores contain additional logic for features that our cores lack - such as bus interface units, debug ports, interrupt support, and FPUs that fully support IEEE 754 double precision, and are targeted towards synthesis and configurability. However, they show that the core areas shown in Table 4.1 are achievable. Designs such as Nvidia’s G80 are essentially chip multiprocessors with significant amounts of area allocated to application-specific hardware (such as texture units) for graphics support. Intel’s 80-core Teraflop Research Chip [25] and IBM’s Cell [26] target high clock frequencies and so can expect a penalty in area. Table 4.2 shows area data for existing designs.
Table 4.1: Area breakdown by pipeline component in mm$^2$.

<table>
<thead>
<tr>
<th>Pipeline Component</th>
<th>1W in-order</th>
<th>2W in-order</th>
<th>2W out-of-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontend</td>
<td>0.016</td>
<td>0.066</td>
<td>0.200</td>
</tr>
<tr>
<td>Execution</td>
<td>0.068</td>
<td>0.092</td>
<td>0.092</td>
</tr>
<tr>
<td>Caches (4KI/8KD)</td>
<td>0.140</td>
<td>0.150</td>
<td>0.150</td>
</tr>
<tr>
<td>Other</td>
<td>0.009</td>
<td>0.026</td>
<td>0.057</td>
</tr>
<tr>
<td>Core (total)</td>
<td>0.330</td>
<td>0.420</td>
<td>0.590</td>
</tr>
<tr>
<td>+SIMDx4</td>
<td>0.200</td>
<td>0.250</td>
<td>0.250</td>
</tr>
<tr>
<td>+MTx2</td>
<td>0.032</td>
<td>0.074</td>
<td>0.120</td>
</tr>
<tr>
<td>+MTx4</td>
<td>0.096</td>
<td>0.220</td>
<td>0.360</td>
</tr>
<tr>
<td>+Trancendentals</td>
<td>0.019</td>
<td>0.019</td>
<td>0.019</td>
</tr>
</tbody>
</table>

Table 4.2: Area Comparison of Commercial Cores (Normalized to 65nm).

<table>
<thead>
<tr>
<th>Core</th>
<th>Area in mm$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensilica 108Mini</td>
<td>.143</td>
</tr>
<tr>
<td>Tensilica 570T</td>
<td>.349</td>
</tr>
<tr>
<td>MIPS 74k</td>
<td>1.7 to 2.5</td>
</tr>
<tr>
<td>Nvidia G80</td>
<td>1.87</td>
</tr>
<tr>
<td>IBM Cell 4x Vector FPU</td>
<td>.65</td>
</tr>
</tbody>
</table>
4.3 Performance Modeling Methodology

We use hand parallelized versions of the VISBench applications, as described in Chapter 2. As described in the parallelization model section of Chapter 2, we replace Pthread and OpenMP calls surrounding parallel loops with annotations for marking the beginning and end of parallel sections as well as the boundaries of individual threads.

We compile our benchmarks using gcc 4.1.2, with the -O3, -ftree-vectorize, -ffast-math, -mfpmath=sse, and -march=pentium4 optimization flags. We compile POVRay with additional flags, -malign-double and -minline-all-stringops. We run each benchmark either until completion or until 2 billion instructions. We fast-forward through initial sequential code ranging from 20M instructions for the SAD kernel to 450M instructions for Facedetect. As discussed in Chapter 3, we are assuming an accelerator model where the host CPU is responsible for executing startup code.

We run our annotated binaries sequentially through a functional simulation frontend that simulates x86 code. This frontend fast-forwards the sequential code to reach the parallel portions that would run on our accelerator, indicated by the start_outer_parallel_loop annotation. The frontend detects the dummy function calls that serve as thread boundaries and generates an instruction trace for each thread. A cycle-based timing model simulates the performance of all cores, running them in parallel to capture the interleaving of memory accesses across threads.

In addition to cores, we model the cache hierarchy as proposed in Chapter 3. We model the global cache as multi-banked with 1 access allowed per bank per cycle. The main memory is modeled as having multiple channels, with memory addresses striped across the channels. For DRAM accesses we assume a fixed
initial access latency and then an additional latency for each block of data that is read, per channel, with only one access at a time permitted per channel, such that if there is contention later accesses will incur longer and longer latency. This allows us to model a fixed bandwidth limit at both the gcache and the main memory.

We assume a hardware mechanism to distribute threads across cores. We assume a single task queue for all the cores, and do not optimize the thread distribution to take advantage of any locality. We do not model a performance overhead for task distribution, or for synchronization at barriers. We assume, with justification, that the overheads that would be imposed are small, certainly smaller than the error margin of our area model. We discuss the limitations of our methodology further in Chapter 10.
An important consideration in designing an accelerator architecture is the memory model, and its attendant mechanisms for synchronization and data communication between parallel processing elements. Traditional multicore architectures implement a cache-coherent shared memory scheme, providing extensive support for data communication between parallel processes. Supercomputers, on the other hand, often have disjoint memories for each processing element, with message passing interfaces for synchronization and communication. GPUs implement shared memories, but with minimal if any support for synchronization. Finally, some accelerator architectures such as Cell use scratchpad memories that require all communication to be explicit.

To evaluate the range of choices in developing a memory and synchronization model, we take a look at the data access and communication patterns within the VISBench applications. We propose a memory hierarchy that relies on caching, and we propose synchronization and communication mechanisms that provide scalability and good performance.\(^1\)

5.1 Data Communication Patterns

In order to develop a memory system and synchronization mechanisms appropriate for VISBench, we need to take a look at the data sharing pattern. We evaluate

\(^1\text{This work was done in conjunction with the Rigel group. Portions of this chapter appear in Kelm et. al. [14].}\)
the frequency with which threads access data from another thread, and develop a classification for this data communication.

We first develop the classification. In Figure 5.1, we return to the parallelization model developed in Chapter 2. The figure, which shows one parallel execution region, is annotated with all the possible ways for data to be communicated between threads. Private writes and private reads are stores and loads of data that is never shared outside of a thread. Input reads are loads to data that was generated before the start of the current parallel region (i.e. outside of the current parallel loop), possibly in a prior parallel region or in a serial region of code. Output writes are stores to data that is read after the end of the current parallel region,

Figure 5.1: Block of parallel code showing possible data sharing.
possibly in a later parallel region or in a serial region of code. In our parallelization model, we impose a barrier at the start and end of each parallel region, and so input reads and output writes both imply data that is communicated across a barrier. Shared writes and shared reads may also take place within a barrier. In this case, a store within a parallel loop produces data that is read by a load in a later iteration of the same parallel loop.

Figure 5.2 plots the frequency with which each of the possible means of data communication occurs in the VISBench applications. It plots the number of input reads, output writes, and shared reads and writes within a barrier, per thousand instructions. We can make a number of observations about the data.

First, note the huge number of input reads. These reads, which correspond to the shared reads across barriers in Figure 2.16, occur from 86 to over 500 times out of every 1000 operations. We further break down the input reads in Figure 5.3, showing the proportion of loads to data that has more than one reader and loads to data with only a single reader. The figure shows that out of the input
reads, the vast majority in all of the benchmarks are to shared data. This implies that the VISBench applications rely heavily on read-shared data structures. This phenomenon is not surprising. Examples read-shared data structures include the scene data in Blender and POVRay, the input image and integral image in Facedetect, object information in ODE, and the input images in MRI and H.264.

A second observation is the modest frequency of output writes. In each of the VISBench applications, each thread is essentially writing a single output value. In some applications this is a single pixel or matrix value. In ODE, each thread may be writing motion or position values for multiple objects. In H.264, each thread writes out an array of SAD values.

Finally, we make some observations about the data sharing within barriers. Such sharing does not occur at all in MRI or H.264. Moreover, it is extremely infrequent in Blender and Facedetect (in fact occurring only once in Facedetect). The only applications where sharing within barriers occurs with appreciable fre-
In ODE, shared writes occasionally arise due to interacting objects. More detail on data sharing in PhysicsBench can be found in [5]. In POVRay, the shared writes occur for a reason unrelated to the algorithm; they are incidental dependences due to reuse of data structures between iterations of the main rendering loop. Recall that we are using a serial implementation of POVRay, which can be rewritten to avoid such incidental data sharing in a parallel version, without an algorithm-level rewrite.

5.2 Memory Hierarchy

In Chapter 3, we proposed a memory hierarchy with off-chip main memory and two levels of caching: a shared global cache and private L1 instruction and data caches. Caching has advantages over software-managed scratchpad memories for applications such as VISBench. Hardware managed caches simplify the programming tasks by automating the movement of data on and off chip. Furthermore, VISBench applications utilize linked data structures, which are challenging to implement with scratchpads. While we do not entirely dismiss the advantages of scratchpad memory, such as lower hardware complexity, for this work we assume caching.

We also assume a shared address space rather than message passing. Earlier we saw that the VISBench applications make frequent use of read-shared data structures. In a message passing architecture we deal with read-shared data either by keeping it on a “home node” and sending it to a requester, or by replicating it on multiple nodes. However, the former strategy is a poor choice because of the frequency of access to such data. Moreover, the limited amount of cache and bandwidth available per core on a single chip limits the appeal of the latter.
strategy. On the other hand, shared memory systems can deal easily with read-shared data, and can even take advantage of the sharing to improve utilization of a shared cache.

The high level of read sharing also motivates the global cache. A global cache is only useful if there is sufficient shared data in the program’s working set. If all or nearly all data input into the application is private to a single thread, it may be preferable to provide only local caches to each core. However, VISBench applications make sufficient use of shared data to justify a shared cache.

The gcache in an accelerator serves a somewhat different purpose than the higher level caches in a CPU. In a CPU, the primary reason for higher level caches (L2, L3) is to reduce the access latency for large working sets of data. In a throughput-oriented accelerator architecture, the reduced access latency is less important as a performance factor. Rather, we want to reduce the off-chip bandwidth requirement.

Previous work on parallel applications has found memory bandwidth to be a first-order performance constraint, particularly on unoptimized code. On modern GPUs manufactured using a 65nm process, the number of available pins limits memory bandwidth to a range of roughly 128GB/s. However, the amount of compute cores we can place on a chip can easily saturate this bandwidth if all the data requested by each core must be brought on-chip every time it is needed.

To show the importance of reducing the bandwidth requirement, we plot the throughput versus gcache bandwidth in Figure 5.4. This graph is generated by simulating an array of 475 dual-issue in-order cores (the highest performing architecture from Chapter 7, with a variable amount of global cache bandwidth. From the figure, we see performance for Facedetect and POVRay saturating at 512GB/s and performance for Blender saturating above 768GB/s. If there were no global cache, this would be the off-chip memory bandwidth required to attain optimal
performance, a level that cannot be provided by memory technologies available at the 65nm generation.

We thus assume that we devote one fourth of our 400mm$^2$ area budget to global cache. Using CACTI, we were able to fit within this area a gcach of 8MB, broken into 32 banks, with a line size of 32B and an access time for each bank of just under 1ns. This allows us to operate the cache at 1GHz, with $32 \times 32B$ accesses per cycle, for a total gcach bandwidth of 1024GB/s, sufficient to saturate performance on all our benchmarks.

This gcach configuration is able to service a large fraction (80-90%) of the memory requests coming from the cores. In Figure 5.5, we plot the performance versus main memory bandwidth for the same compute array assuming the presence of the gcach described above. From the figure, we see that performance saturates on all of the benchmarks when the bandwidth reaches 64GB/s. In the rest of this thesis, we assume a main memory with 8 independent DRAM channels, each with a bandwidth of 16GB/s, for a total bandwidth of 128GB/s. This is achievable
Figure 5.5: Performance versus off-chip memory bandwidth (normalized to maximum).

with 512 data pins signalling at 2GHz, roughly comparable to a high-end GPU. Because we are able to achieve a high hit rate in the gcache, this bandwidth is sufficient to allow us to saturate the compute array.

5.3 Synchronization and Communication

The input reads and output writes in VISBench threads necessitate, at a minimum, support for bulk synchronization. The basic idea of the bulk synchronous parallelism model is shown in Figure 5.6. In this model, a parallel region is preceded and followed by barriers. Any data value written before the initial barrier is visible to every parallel task. During the parallel region, all of the parallel tasks are assumed to be independent. Once all the parallel tasks have completed, a communication phase takes place, and then the barrier. After the barrier, all output values produced by tasks before the barrier are visible to any tasks performed after the barrier.
Note the similarity between Figure 5.6 and Figure 2.4. The bulk synchronous parallelism (BSP) model is a natural fit to our parallelization model described in Chapter 2. To support BSP, the hardware must provide mechanisms to flush output data out of the L1 data caches to the gcache or main memory. Furthermore, the hardware must provide a mechanism to ensure that input reads access data from the gcache and avoid reading stale values in private caches. Finally the hardware or programming model must provide a mechanism to implement the barrier itself.

The BSP model covers most of the data communication needs in VISBench, since it handles input reads and output writes. The BSP model does not provide any support for data dependences between parallel tasks running within the same parallel region. However, we saw in Figure 5.2 that some data sharing does occur within barriers, particularly in ODE physics simulation. Hence, we need some sort of mechanism to deal with fine-grained synchronization.
Multicore architectures support fine-grained synchronization and data communication through cache coherence. Cache coherence allows parallel processing elements to communicate data through memory without incurring a performance overhead in the absence of a collision. This is particularly useful if the application has many writes to and reads from write-shared data, but where writes to the same location rarely collide. However, from Figure 5.2 we see that shared writes are infrequent, and while shared loads are common, the vast majority are separated from their corresponding writer by a synchronization barrier. This suggests that fast, automatic cache coherence in hardware exceeds what is necessary for an accelerator. Moreover, cache coherence is difficult to scale to large numbers of cores without incurring a large cost in hardware and verification complexity. Instead, we propose a non-coherent shared memory, and provide hardware support for enforcing coherence in software.

The basic mechanism for keeping shared data consistent is to exclude such values from local caches. Shared reads and writes are performed with special global load and store operations, which always go out to the global cache or memory. In addition, input data must be read in globally at the beginning of each thread while output data must be flushed out from local caches at the conclusion of each thread (though input and output values can be locally cached during the running of the thread). This model requires the programmer to identify the input, output, and shared data and use global operations and flushes when appropriate.

To support fine-grained global locking, we can provide atomic operations at the gcache. To perform an atomic operation, for instance an atomic increment, a core sends a message to the gcache bank containing the value. The increment is performed by the cache controller responsible for the bank. After performing the operation, the controller sends a message back to the core. These atomic operations can be combined with global loads and stores to build a variety of
Figure 5.7: Number of sharers for benchmarks that have within-barrier data sharing.

Note that using global operations to perform data communication affords lower performance for synchronization and communication than is provided by hardware cache coherence. This is due to the inability to cache locks or to cache data that is write-shared within a single parallel region. However, the performance cost of this in VISBench applications is minimal, because such communication is infrequent. Communication at barriers, on the other hand, is just as efficient as in a cache coherent machine, because the the input values need only be read globally once at the start of the parallel task, and the output values only flushed to the gcache once at the barrier, which produces the same amount of communication that would be required in a coherent caching system.

The performance disadvantage of incoherent shared memory on fine-grained synchronization can be further mitigated by making an observation about shared
writes in VISBench. In Figure 5.7, we see that in the applications where fine-grained synchronization may be necessary, the vast majority of intra-barrier communication occurs between small numbers of threads. In such a case, providing small cache coherent domains in the compute array could provide almost all of the performance benefit that would be attained by global cache coherence, with a much lower complexity. In this case, threads running on a single, cache coherent cluster would be able to cooperate with other threads on the same cluster, with high performance, but would have to use slow global operations to cooperate with threads on other clusters.
SIMD EXECUTION

Single-instruction, multiple-data (SIMD) execution is a common route to achieving high performance in architectures ranging from traditional vector-processing supercomputers to modern GPUs. SIMD execution can be very effective at obtaining high performance on numerical applications. On the other hand, it may provide poor performance on control-intensive code.

The compute fabric of the xPU can either be designed using discrete cores which execute in a multiple-instruction, multiple-data (MIMD) fashion, or it can be designed with clustered scalar pipelines that execute in SIMD lockstep in a manner similar to GPUs. Dense numerical applications are able to take advantage of SIMD because of the very regular pattern of control flow. In the case of visual applications, we could replace the thread execution model (which is notably MIMD) with large scale SIMD if the control flow were largely the same across different threads. On the other hand, if control flow varies from thread to thread, SIMD hardware will suffer a performance loss as the parallel threads would be forced to serialize.

In this Chapter, we examine a SIMD architecture for the accelerator compute array and compare it to a MIMD architecture with discrete cores. We evaluate the factors which affect area and performance for the SIMD and MIMD design points. Finally, we examine the tradeoffs of a SIMD architecture from the application programming perspective.
6.1 SIMD Architectures

In single-instruction, multiple-data (SIMD) architectures, a single instruction can operate on multiple data elements. The benefit of SIMD is a savings in terms of chip area or power consumption, as only a single instruction stream is being executed in a parallel computation.

A variety of architectures have been proposed that take advantage of the SIMD execution model. Traditional supercomputer architectures have relied on vector processing. Modern CPU architectures typically include a subset of SIMD instructions that operate on short vectors. On the other hand, modern GPU architectures include clusters of processing elements that execute in lockstep, providing a quasi-MIMD programming model atop SIMD hardware.

Traditional supercomputers often used pipelined vector processing. In such vector processors, each instruction would set up a computation to be performed on an entire vector of data elements. While the elements were processed sequentially rather than in parallel, the use of SIMD allowed the machine to be deeply pipelined and run at very high (for the time) frequencies.

Another style of SIMD, common in modern CPU architectures, is a subset of short-vector SIMD instructions. In this case, a processor generally executes scalar instructions, but for dense numerical calculations a number of instructions are provided that operate on vectors of 2, 4, or up to 16 data values stored in vector registers. Examples of this type of SIMD include SSE for x86 processors, AltiVec for PowerPC, and VIS extensions on SPARC. We examine this style of SIMD architecture in Chapter 7.

In this Chapter, we focus on a style of SIMD used in accelerator architectures such as GPUs, sometimes referred to as streaming SIMD [27] or SIMT (single-instruction, multiple-thread) [11]. In this architecture, shown in Figure 6.2, there
Figure 6.1: A MIMD 4-core cluster.

Figure 6.2: A SIMD 4-core cluster.
is a single control unit for each SIMD cluster of processors. The single control unit includes the instruction fetch and decode, the frontend of the processor execution pipeline. This unit sends decoded instructions to the parallel execution units, each of which is operating on a different set of data.

Compare the SIMD architecture shown in Figure 6.2 to the MIMD architecture shown in Figure 6.1. In the MIMD architecture, each core executes a separate instruction stream, and contains all of the elements of a discrete in-order core as modeled in Chapter 4. In the SIMD architecture, on the other hand, all of the control logic as well as the front-end of the execution pipeline occurs only once. However, the datapath of the processor is replicated, primarily including the register file, execution units, and data bits of the pipeline registers. In addition, for a fair comparison we assume both architectures include the same amount of data cache and same number of data cache ports per processing element.

With a 4KB instruction cache and an 8KB data cache per core, we find that the replicated portion of the pipeline constitutes 60% of the total area of the core (comparing to the single-issue in-order core). The portion of the pipeline that does not need to be replicated (control bits of the pipeline latches, pipeline front end, instruction cache) consists of 40% of the total. Hence, a 2-way SIMD cluster has 1.6 times the area of a scalar pipeline, while a 4-way has 2.8 times the area.

This style of SIMD architecture maps well to the parallelization model developed in Chapter 2. Each thread (i.e. each iteration of the parallel loop) can be mapped to one of the processing units. In the VISBench applications, the iterations of the parallel loop are handling different data elements but running the same code. Hence, we are essentially using a single-program, multiple-data (SPMD) programming model to implement code to run on SIMD hardware, similar to the CUDA model.
6.2 Performance Limiters for SIMD

A major factor that can limit the performance of SIMD architectures such as the one in Figure 6.2 is control flow divergence. In a MIMD architecture, each compute core can follow a different execution path with no loss of efficiency. In the SIMD architecture, on the other hand, the entire cluster is capable of executing only one control flow path at a time. If different threads need to execute different code paths, the cluster will need to be serialized.

We refer to a group of scalar threads executing on the parallel processing elements of a single SIMD cluster as a warp. From the programmer’s perspective, each thread in the warp is capable of executing an independent control flow path. However, since the hardware is only capable of executing one path at a time, we must serialize threads whenever the control flow diverges, executing the subset of threads following one control path followed by the threads executing a different path.

A simple mechanism for handling control flow divergence is predication with guarded execution. Predication, which was implemented in traditional vector processors, is a simple means of handling simple control divergence situations such as code hammocks. However, predication requires every thread to execute every control flow path (committing only results from the taken paths). With any complex control flow, predication is insufficient.

A more general mechanism for handling control divergence in SIMD architectures is a stack-based approach [11]. In this approach, a PC is maintained for each thread. As long as all the threads in a warp are executing the same PC, all the threads are executed. When there is a branch where threads diverge, a sub-warp of threads going in one direction is executed while the stalled threads are pushed onto a stack. When the sub-warp reaches the reconvergence point, the threads in
PC code
1 if (x==0)
2 if (y==0)
3 a = 1
4 else
5 a = 2
6 b = 3
7 else
8 a = 3
9 c = 4

Figure 6.3: Example code for stack-based mechanism for handling control flow divergence.

it are stalled while the previously stalled threads are popped from the stack and executed.

An example of the operation of the stack-based mechanism is shown in Table 6.2. (The code is shown in Figure 6.3.) In this example, there are two branches (instructions 1 and 2) and two reconvergence points (instructions 6 and 9). When threads diverge (cycles 2 and 3), threads following one of the paths are executed and the remaining threads pushed onto the stack. When these threads reach the reconvergence point, these threads are swapped with those on the top of the stack. When the remaining threads reach the reconvergence point, the top of the stack is popped and the reconverged threads can continue executing.

The reconvergence points can be determined statically, by analyzing the control flow graph of the program. The naive approach is to use the end of the thread as the reconvergence point. This is far from optimal, as it performs far from optimally on code that reconverges long before the end of the thread. A much better choice for the reconvergence point is the immediate postdominator of each branch. This approach is near optimal on most code, including VISBench, though there exist pathological cases for which it works poorly [27].

To determine the relative performance of the SIMD architecture from Fig-
Table 6.1: Example of stack-based mechanism for handling control flow divergence. Threads marked with an x are executing in the given cycle.

<table>
<thead>
<tr>
<th>cycle</th>
<th>HW PC</th>
<th>th 0</th>
<th>th 1</th>
<th>th 2</th>
<th>th 3</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>x1</td>
<td>x1</td>
<td>x1</td>
<td>x1</td>
<td>empty</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>x2</td>
<td>x2</td>
<td>x2</td>
<td>7</td>
<td>{t3}</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>x4</td>
<td>x4</td>
<td>7</td>
<td>{t0}{t3}</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>x5</td>
<td>x5</td>
<td>7</td>
<td>{t0}{t3}</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>x3</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>{t1,t2}{t3}</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>x6</td>
<td>x6</td>
<td>x6</td>
<td>7</td>
<td>{t3}</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>x7</td>
<td>{t0,t1,t2}</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>x8</td>
<td>{t0,t1,t2}</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>x9</td>
<td>x9</td>
<td>x9</td>
<td>x9</td>
<td>empty</td>
</tr>
</tbody>
</table>

Figure 6.2 versus the MIMD architecture from Figure 6.1, we simulate the execution of the VISBench applications on a SIMD architecture assuming the stack based reconvergence mechanism and the immediate postdominator as the reconvergence point. We idealize the memory system and assume unit latencies, and compare the instruction throughput per pipeline of the SIMD and MIMD architectures. To minimize the divergence within thread warps, we group together threads which operate on adjacent data elements.

Figure 6.4 plots the relative IPC of the SIMD architecture (per pipeline, relative to MIMD) for varying warp sizes. This number is the fraction of instruction issue slots that are successfully filled, excluding those which cannot be filled due to control flow divergence.

All of the application have either a level or declining SIMD efficiency as the warp size grows. This trend is unsurprising, as a larger warp will naturally tend to
have a greater incidence of control flow divergence than a smaller warp. However, the degree of this trend varies by application. For some applications (H.264 and MRI), the SIMD efficiency is very high, even for large warp sizes. In MRI, which is basically a convolution, all of the threads execute exactly the same code path, resulting in SIMD efficiency remaining perfect for all warp sizes within the range of the study. In the H.264 SAD kernel, most threads execute the same path, except those at the edges where the image needs to be padded. As padding tends to occur in adjacent threads, the SIMD efficiency remains high. The remaining applications, on the other hand, show a pronounced decrease in SIMD efficiency as the warp size grows. These applications all have data-dependent branching. At small warp sizes, these applications are still able to fill most of the issue slots, but with larger warp sizes the efficiency trends towards zero.

While SIMD has a cost in IPC, MIMD has a cost in chip area. MIMD requires all the control flow logic of the core to be replicated for each pipeline, whereas SIMD only requires one set of control logic per cluster. However, SIMD still requires the register files, functional units, caches and cache ports, and data bits of the pipeline latches to be replicated. We find that roughly 40\% of the pipeline area does not need to be replicated. Hence, a 2-way SIMD cluster has 1.6 times the area of a scalar pipeline, while a 4-way has 2.8 times the area.

Figure 6.5 shows the ratio between area and IPC per cluster for varying warp sizes. The benchmarks in VISBench separate clearly into three groups. The first group is the one that has nearly perfect SIMD efficiency (i.e. H.264 and MRI). The second group (Blender and ODE), shows a modest performance benefit for small warp sizes, but then exponentially decreasing performance as the warp size grows large. The third group (POVRay and Facedetect) shows performance loss with any level of SIMD, and exponential performance loss with large warp sizes. Note that we are assuming perfect memory, so this result is an upper bound on
Figure 6.4: SIMD efficiency versus warp size.

the SIMD efficiency.

One thing we have not considered in this study is algorithmic changes to improve SIMD efficiency. Blender and H.264 required a moderate amount of hand tuning in order to achieve their level of SIMD performance, but were not altered significantly at the algorithm level. With additional programmer effort, one may be able to reclaim much more of the performance loss from SIMD, but this optimization comes at a substantial cost in development time. For instance, work has shown that the SIMD efficiency of ray tracing can be improved, though even with major algorithmic changes it remains well below 100% on complex scenes [28].

This result indicates that SIMD, while a substantial constant factor win for some applications, is a much larger performance loss for others. It also illustrates one of the limitations of GPUs as they expand into more general purpose application domains. For numerical applications and the traditional applications for GPUs, SIMD is the right design choice. However, the performance potential of SIMD architectures is limited as the space of applications expands.
6.3 Application Tuning for SIMD

In this section we examine some of the application tuning techniques that can be applied to improve the performance of SIMD architectures. We explore a handful of simple techniques, and as a case study examine how some of these techniques can be applied to improve the SIMD performance of one of our least SIMD-friendly applications, Facedetect.

6.3.1 Tuning techniques

There are a variety of techniques that can be applied to programs to improve their performance on a SIMD architecture. These techniques include low-level code tuning, such as removing specific branches that cause divergence. They also include algorithmic changes, such as sorting tasks, reordering loops, doing additional work in order to avoid control flow, and parallelizing at a finer granularity.

One low-level technique to improve SIMD efficiency is to replace divergent
branches with logical operations. Such an optimization is useful for code sequences such as an if-then-else block. However, such an optimization is difficult or ineffective for removing large-scale control flow, such as where the different control flow paths perform widely different operations.

In many cases, the control flow is dependent on the data elements each thread is working on. In these cases, the divergence can be reduced by sorting tasks or reordering parallel loops so that tasks that will perform similar operations appear in the same warp.

In some cases, control flow divergence occurs as a result of code optimizations. A thread may use control flow to pare down the amount of computation based on the data elements it is processing. An example of such an optimization is going from a full search of some space to a filtered search. On a SIMD architecture, removing such an optimization may actually improve performance, despite an increase in the overall work.

One powerful technique for improving SIMD performance is to parallelize at a finer granularity. On a MIMD architecture, we prefer to parallelize applications at the coarsest granularity at which load imbalance is not an issue. A higher granularity of thread size reduces the overhead of starting, ending, and scheduling tasks. However, on a SIMD architecture, coarse task granularity can increase control flow divergence. The number of possible control flow paths that a thread can follow may grow rapidly, even exponentially, with the length of the thread. Hence, a finer-grained decomposition of the program can reduce the number of potential control flow paths. Moreover, control flow divergence tends to be low among tasks operating on adjacent data elements (hence the high SIMD efficiency for a warp size of 2), and increasing with more distant data elements, since data elements that are closer together tend to require more similar operations. A finer-grained thread will operate on fewer data elements than a coarser-grained thread,
reducing the likelihood of divergence due to varying data values.

Decreasing the granularity of parallelization was applied to some of the VISBench applications where it was relatively simple to do so. In particular, the granularity in Blender was reduced by shrinking each thread from rendering 16 pixels to rendering 4 pixels. Also, the granularity in H.264 was reduced by parallelizing an inner loop. In the case of Blender, the optimization pushed the dropoff in SIMD efficiency to a warp size 4 times larger than before. In the case of H.264, the optimization virtually eliminated control divergence.

6.3.2 Tuning Facedetect for SIMD

For a more in-depth application of the above tuning techniques, we examined Facedetect. In the algorithm used for Facedetect, decribed by Viola and Jones in [8], the image is divided into rectangular subimages. Each subimage is then tested to see if it contains features. A cascade of features is used to determine whether the subimage is a face. In other words, the subimage is tested to see if it contains the first feature in the cascade, and if so is tested for the second feature, and so on. If the subimage contains all of the features in the cascade, it is marked as containing a face.

The main parallel loop of the Haar-cascade classifier in Facedetect is shown in Figure 6.6. The divergent branches in the code are marked. Table 6.2 gives a count of the number of times each of the marked branches diverges during the simulated 2B instructions of execution.

From the code, we can see that all 4 of the diverging branches in this loop result from the result of running a subimage through the classifier cascade. A positive result means that there is a possible face in the subimage, while a negative result means there is likely not. Branches A, B, and D depend directly on the result,
start_outer_pl();
for(_iy = 0; _iy < stop_height; _iy += ystep)
{
    start_outer_parallel_iter();
    int iy = cvRound(_iy*ystep);
    int _xstep = 1;
    for(_ix = 0; _ix < stop_width; _ix += _xstep)
    {
        int ix = cvRound(x*ystep);
        if( pass == 0 )
        {
            int result;
            _xstep = 2;
            result = cvRunHaarClassifierCascade(ix, iy);
            A if( result > 0 )
                {
                    mask_row[ix] = 1;
                }
            B if( result < 0 )
                _xstep = 1;
            }
        C else if( mask_row[ix] )
        {
            int result = cvRunHaarClassifierCascade(ix, iy);
            D if( result > 0 )
                {
                    if( final pass )
                        {
                        save rect(ix, iy);
                    }
                }
            else
                mask_row[ix] = 0;
        }
    end_outer_parallel_iter();
}
end_outer_pl();

Figure 6.6: Facedetect parallel loop code. Some details have been removed.
Table 6.2: Count of divergence at branches in Facedetect loop.

<table>
<thead>
<tr>
<th>branch</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>80354</td>
</tr>
<tr>
<td>B</td>
<td>86662</td>
</tr>
<tr>
<td>C</td>
<td>83624</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
</tbody>
</table>

while C depends on a result from a previous loop iteration.

We can reduce the impact of control flow divergence in this code by parallelizing it at a finer granularity. This can be achieved by parallelizing the inner loop, across different values of _ix. However, the inner loop has a loop carried dependence during the first pass, since we vary _xstep. Essentially, every time there is a positive result from running the classifier, we skip the next iteration.

To get around this, we split the loop into two, one for the first pass and another for the later passes, as shown in Figure 6.7 and 6.8. We can then parallelize the inner loop in Figure 6.8. The SIMD efficiency of the inner loop, in this version 2 of the code, is shown in Figure 6.10. The SIMD efficiency of the inner loop is little improved for small warp sizes but is improved by a factor of 3 for large warp sizes. However, because the first pass, which accounts for roughly 70% of the compute time still has very low efficiency, the overall SIMD efficiency of version 2 is only slightly better than for the baseline.

The loop carried dependence in the first pass (Figure 6.7) is the result of an optimization which is pruning the search space. In fact, if we eliminate the manipulation of _xstep, we can parallelize the first pass inner loop as well. However, because we are no longer skipping iterations, we are doing extra work and also slightly increasing both the detection accuracy and the false positive rate. However, the improvement in SIMD efficiency in version 3 of the code, with this change, is considerable, as seen in Figure 6.10.
start_outer_pl();
for(_iy = 0; _iy < stop_height; _iy += ystep)
{
    start_outer_parallel_iter();
    int iy = cvRound(_iy*ystep);
    int _xstep = 1;
    for(_ix = 0; _ix < stop_width; _ix += _xstep)
    {
        int ix = cvRound(x*ystep);
        int result;
        _xstep = 2;
        result = cvRunHaarClassifierCascade(ix, iy);
        if( result > 0 )
        {
            mask_row[ix] = 1;
        }
        if( result < 0 )
        _xstep = 1;
    }
    end_outer_parallel_iter();
}
end_outer_pl();

Figure 6.7: Facedetect first pass loop, version 2.
start_outer_pl();
for(_iy = 0; _iy < stop_height; _iy += ystep)
{
    start_outer_parallel_iter();
    int iy = cvRound(_iy*ystep);
    int _xstep = 1;
    start_inner_parallel_loop();
    for(_ix = 0; _ix < stop_width; _ix += _xstep)
    {
        start_inner_parallel_iter();
        int ix = cvRound(x*ystep);
        if( mask_row[ix] )
        {
            int result = cvRunHaarClassifierCascade(ix, iy);
            if( result > 0 )
            {
                if( final pass )
                {
                    save_rect(ix, iy);
                }
            }
            else
            mask_row[ix] = 0;
        }
        end_inner_parallel_iter();
    }
    end_inner_parallel_loop();
end_outer_parallel_iter();
}
end_outer_pl();

Figure 6.8: Facedetect later passes loop, version 2.
start_outer_pl();
for(_iy = 0; _iy < stop_height; _iy += ystep)
{
    start_outer_parallel_iter();
    int iy = cvRound(_iy*ystep);
    int _xstep = 1;
    start_inner_parallel_loop();
    for(_ix = 0; _ix < stop_width; _ix += _xstep)
    {
        start_inner_parallel_iter();
        int ix = cvRound(x*ystep);
        int result;
        _xstep = 2;
        result = cvRunHaarClassifierCascade(ix, iy);
        if( result > 0 )
        {
            mask_row[ix] = 1;
        }
        end_inner_parallel_iter();
    }
    end_inner_parallel_loop();
    end_outer_parallel_loop();
}
end_outer_pl();

Figure 6.9: Facedetect first pass loop, version 3.
However, the change in version 3 increases the overall computation time, by 65%. As a result, the effective SIMD efficiency, in comparison to the baseline version, is improved by much less. Nonetheless, for large warp sizes version 3 performs about 5X better than the baseline, as can be seen in Figure 6.11.

One could also make further changes to improve the SIMD efficiency of the algorithm. For instance, there is a substantial amount of divergence within the cvRunHaarClassifierCascade function, as each execution of this function traverses a cascade of classifiers. At each step of the cascade, we check to see if a certain
feature is present in the rectangle. If it is present, we continue, and otherwise we return a negative result. In most rectangles, the computation quits after the first one or two features, but in rectangles with a face, the computation continues to the end of the cascade. We can reorder this computation by decomposing it into two phases. In the first phase, all of the rectangles are tested against the first two features of the cascade. We then gather the set of rectangles that contained features and run them through the second phase, which traverses the rest of the cascade. Such an algorithm change would work poorly on the MIMD architecture, since it would require an extra barrier, but on a SIMD architecture the lockstep execution makes a warp-level barrier very fast. However, even this change would not achieve full SIMD utilization, especially in the second phase.

The overall performance per area of Facedetect remains better on the MIMD architecture, even after tuning steps are applied. However, the example illustrates that SIMD efficiency can be improved through programming effort. Heavy application tuning can improve SIMD performance even on application with lots of divergence, and if these applications are of secondary importance, the SIMD architecture is a defensible design choice. However, if we are designing an accelerator architecture for a large class of applications with control flow divergence, as is the case in this thesis, the MIMD architecture remains the better choice.
CHAPTER 7

ARCHITECTURAL TRADEOFFS IN THE ACCELERATOR CORE

In this Chapter, we examine the architecture of the cores in the compute array. We focus on macro-architectural design choices, such as pipeline organization and execution units. We also examine the sizing of the private L1 caches.

7.1 Experimental Method

In this study we experimentally approach the question of how to architect the compute array. In particular, we focus on the architecture of the cores in the array, including the L1 caches. We evaluate different pipeline configurations that we modeled in Chapter 4, as well as a number of architectural features that have been proposed to improve throughput.

The objective function we seek to maximize is performance given a fixed area budget. The chip we are modeling has a 400mm² area budget as described in Chapter 4. We devote 100mm² of this area to the global cache, as described in Chapter 5. Furthermore, we assume that 100mm² will be consumed by the global interconnect, I/O, memory controllers, and control logic. This leaves half of the chip area, 200mm² to devote to the compute cores and the L1 caches.

We use our hybrid area model to compute the area per core, including L1 cache, for each of the architectural configurations in our design space. We then fill the 200mm² of area devoted to the compute cores with as many cores as will fit. In other words, we set the core count to be $\frac{200\text{mm}^2}{\text{areapercore}}$, rounded off to the
nearest integer.

We use the performance methodology developed in Chapter 4 to simulate the chip with each architectural configuration. We architected each of the core configurations with 30-40 FO4 delays, which results in a cycle time of slightly under 1ns on a 65nm process. We therefore assume that all of the core configurations run at a 1GHz clock rate, and compare the IPC of the various architectures.

We assume an architecture with two levels of caching, with an L1 dcache and icache that is private to each core and a global cache (gcache) that is shared by all cores. As described in Chapter 5, we used CACTI to find the largest global cache size that would fit in 100mm² with 32 banks such that each bank has an access delay under 1ns (8MB). With a 1GHz clock rate this configuration provides a global cache bandwidth of 1024GB/s. The size of the gcache is consistent with the cache sizes seen on current generation microprocessors. We also assume a fixed memory bandwidth of 128GB/s; memory bandwidth is a function of the number of pins available on a package, and tends to have an upper bound in any given process generation. The gcache bandwidth we need to fully utilize the memory bandwidth is the memory bandwidth divided by the expected miss rate. With a miss rate of 12%, 32 gcache accesses per cycle (1024GB/s) is a reasonable value.

The baseline configuration of the xPU architecture is given in Table 7.1. Table 7.2 gives the range of parameter values in our design space. In this study we focus on what we call the macro-architecture of the core, the architecture that affects the algorithm of the core, with the exception of L1 cache size, which is an element of the micro-architecture.
Table 7.1: Baseline XPU architecture.

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Latency</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int ALU</td>
<td>32-bit</td>
<td>1 cycle</td>
<td>1 or 2 ALUs</td>
</tr>
<tr>
<td>FPU</td>
<td>single-precision</td>
<td>5 cycle</td>
<td>1 FPU</td>
</tr>
<tr>
<td>L1 ICache</td>
<td>4KB</td>
<td>1 cycle</td>
<td>2-way</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>8KB</td>
<td>1-2 cycles</td>
<td>4-way</td>
</tr>
<tr>
<td>GCache</td>
<td>8MB</td>
<td>20+ cycles</td>
<td>32 banks 8-way</td>
</tr>
<tr>
<td>DRAM</td>
<td>128GB/s</td>
<td>50ns</td>
<td>8x64bits</td>
</tr>
</tbody>
</table>

Table 7.2: Study parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issue width</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>scheduling</td>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>ALUs</td>
<td>32-bit scalar</td>
<td>128-bit vector</td>
</tr>
<tr>
<td>multithreading</td>
<td>single-thread</td>
<td>4-way</td>
</tr>
<tr>
<td>transcendental</td>
<td>50 cycles</td>
<td>6 cycles</td>
</tr>
<tr>
<td>L1 ICache</td>
<td>2KB</td>
<td>16KB</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>4KB</td>
<td>32KB</td>
</tr>
</tbody>
</table>

Figure 7.1: Performance for different pipeline configurations.
7.2 Results

In Figure 7.1, we plot the performance of all of the architectural configurations evaluated in our design space. Below, we examine the results focusing on the parameters varied in Table 7.2.

7.2.1 Superscalar issue and dynamic scheduling

Initially we ask the question of how one should design the core pipeline architecture of each processing unit within the xPU given the tradeoff in area/performance for each style of core.

Figure 7.1 plots the performance per area (total throughput divided by the $200 \text{mm}^2$ of area used by the compute array) versus the area per core. Each data point represents the harmonic mean of the throughputs of the 6 benchmarks for a particular core architecture, cache size, and core count corresponding to that configuration. The smallest single-issue in-order configuration, with a core area of $0.205 \text{mm}^2$ including L1 caches, allows us to fit 975 cores in the array. The largest 2-issue out-of-order configuration, with a core area of $1.10 \text{mm}^2$, allows us to fit 182 cores in the array.

First, we note that the smallest configuration in1 (with the 2KI/4KD caches) is not the highest performing despite its high core count. With a small cache and a large number of cores, this configuration is bound by the gcache bandwidth on most of the benchmarks. On the one benchmark requiring the least gcache bandwidth (H.264 ME), this minimal configuration is the highest performer.

Second, we note that the highest performing configurations are in2. The in2 configurations provide the highest theoretical throughput, and even with code that is not optimally scheduled, we are able to take enough advantage of ILP in order to overcome its area penalty versus in1.
Third, we note that the best performing in1, in2, and out2 configurations are fairly close in performance, within the uncertainty margin of our area model. Moving from in1 to in2 reduces the pipeline utilization (achieved throughput vs. theoretical throughput), but benefits from increased execution resources. Moving from in2 to out2 restores the utilization by scheduling around instructions such as FP operations with moderate latency, but the increased area overhead from the scheduling logic matches the performance gain. Figure 7.2 breaks down by benchmark the performance of the baseline in1, in2, and out2 architectures (with the intermediate 4KI/8KD caches).

7.2.2 Vector ALUs

In Section 3, we examined the performance potential of scalar threads executing in SIMD lockstep. Here we examine word-level SIMD (SIMD operations on small vectors, e.g. SSE), featured in most high performance architectures. We model the area cost of the vector ALUs required to support the small-vector SIMD
operations by replicating each arithmetic integer and floating point unit 4 times. We also model the cost of 128-bit wide read and write ports in the register file.

Figure 7.3 shows the speedup from adding 128-bit SIMD execution units to our baseline architecture. For in1, the area overhead is 59% (39-89% with our area model confidence interval), for in2 it is 58% (39-87%), and for out2 it is 43% (28-64%). The minimum overhead is computed by comparing the upper bound of the baseline area with the lower bound of the extra area consumed by SIMD. Likewise the maximum overhead is computed by lower bound of the baseline with the upper bound of the SIMD area. The error bars on the figure account for the variable area estimates, as described in Chapter 4.

From the figure we see that SIMD generally results in a loss of performance on our versions of VISBench, especially with in-order pipelines. This is because, for small core sizes, the area of the FP unit is a large fraction of the total and the penalty for replicating it 4 times is very large relative to the utilization rate.

Furthermore, the benefit is limited to those applications for which the gcc compiler is able to generate substantial amounts of vector code. The MRI kernel is easily vectorized. The OpenCV library was heavily optimized for SSE instructions, with 14% of all operations being SIMD. These two applications were able to achieve modest speedups with the out-of-order configuration, but losses with in-order. Portions of ODE, POVRay, and Blender were also vectorized. However, in the other applications the amount of vectorized code is too small to overcome the area cost of the additional FP units.

Traditional GPU architectures often used 4-wide vector ALUs in pixel shader pipelines, in addition to the sort of lockstep execution examined in Chapter 6. However, the newest GPUs have moved to scalar shader pipelines, reinforcing our result. One should note that vector ALUs make more sense on a processor optimized for single-thread performance (such as a multi-core desktop CPU) where
area per core is large and the cost of SIMD support is relatively minor.

### 7.2.3 Multithreading

Multithreading is an important technique to get around stalls due to long latency memory operations, and for applications that are throughput-oriented, hardware multithreading seems like a natural fit.

Recall from Chapter 4 that the area overhead of multithreading involves the replication of register files, pipeline registers, rename tables, and the PC. Table 7.3 shows the change in core count when multithreading is added to the architecture of the superscalar pipelines. The area overhead of 2-way multithreading is 9-10%, while that of 4-way is 27-31%.

In Figure 7.4, we show the relative performance of configurations with fine-grained multithreading added. The results show that, for most benchmarks, 2-way multithreading is able to provide a performance benefit. The exceptions are the H.264 motion estimation kernel, which achieves high utilization even without MT, and Facedetect, which suffers from load imbalance as the number of contexts
Table 7.3: Core counts with multithreading.

<table>
<thead>
<tr>
<th>pipeline</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>in2</td>
<td>473</td>
<td>428</td>
<td>361</td>
</tr>
<tr>
<td>out2</td>
<td>341</td>
<td>313</td>
<td>269</td>
</tr>
</tbody>
</table>

exceeds the number of available threads.

Four-way multithreading, on the other hand, does not generally have a performance benefit. Our VISBench applications spend far less than half of their time stalled waiting for loads, and 2 threads is sufficient to cover most of the stall cycles. Furthermore, the area cost of multithreading grows linearly as thread count, as you need to continue replicating structures such as the register file.

Facedetect is the only benchmark that fails to achieve any performance benefit from multithreading. The parallel phases of Facedetect each have roughly 450 parallel threads, which is not enough to utilize all the hardware contexts in the multithreaded configuration (in our case, the base in2 has 341 contexts, in2.mt2 has 606 contexts, and in2.mt4 has 988 contexts) and still maintain load balance. As a result, Facedetect becomes load imbalanced as the shortest threads complete.

7.2.4 Hardware implementation of trigonometric and transcendental functions

We pointed out in Chapter 2 that some VISBench applications rely heavily on a small set of complex functions, such as sin, cos, atan, exp, sqrt, and rsqrt for calculating transforms, angles of incidence, and various other geometric and visual actions. Table 7.4 shows the frequency of such operations in VISBench.

Figure 7.6 shows the relative performance of the configurations with an added hardware support for trig and transcendentals. The figure shows improved inorder performance with the trig units. The performance benefit is particularly
Figure 7.4: Performance of 2-wide in-order configuration with multithreading.

Figure 7.5: Performance of 2-wide out-of-order configuration with multithreading.

Table 7.4: Frequency (per 1000 instructions) of trig and transcendental operations.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>sin</th>
<th>cos</th>
<th>atan</th>
<th>sqrt</th>
<th>exp</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>blender</td>
<td>.013</td>
<td>.006</td>
<td>.006</td>
<td>.06</td>
<td>0</td>
<td>.089</td>
</tr>
<tr>
<td>mri.fh</td>
<td>15.0</td>
<td>15.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30.0</td>
</tr>
<tr>
<td>facedetect</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>.59</td>
<td>0</td>
<td>.59</td>
</tr>
<tr>
<td>povray</td>
<td>.001</td>
<td>.08</td>
<td>.03</td>
<td>.03</td>
<td>.52</td>
<td>1.7</td>
</tr>
<tr>
<td>ode</td>
<td>.02</td>
<td>0</td>
<td>.002</td>
<td>0</td>
<td>.01</td>
<td>.03</td>
</tr>
</tbody>
</table>
strong for MRI, which has the heaviest concentration of such operations, with one of every 32 operations being either sin or cos. SAD, on the other hand, shows very little performance effect from such units. Facedetect, ODE and POVRay, meanwhile show a small performance benefit, roughly within the margin of error. The actual trig calculation is performed with a series of table lookups, FP adds, and FP multiplies, each of which is sequentially dependent. Hence, a slow trig calculation occupies the FP unit and blocks progress in the in-order machine but not the out-of-order machine.

7.2.5 Cache sizing

Cache sizing is an important parameter in architecture design. Previous CMP optimization studies have shown that optimal cache size is a function of the application being run. We examine performance per area for the core configurations with varying cache sizes. Figure 7.7 plots the data points showing the area per core and performance of in1, in2, and out2 configurations with varying cache sizes.

From the plot, we can see that most of the highest performing configurations
Figure 7.7: Area per core versus performance per area while sweeping the cache sizes from the smallest to the largest.

have the 4K/8K L1 cache sizes, while many of the 8K/16K configurations also perform well. On the other hand, the configurations with the smallest and largest cache sizes perform worse.

To see some of the application specific behavior, we examine the 2-wide inorder configurations with the full range of cache sizes. The points are broken down by benchmark in Figure 7.8. Here, we see two behaviors: declining performance for all cache sizes, and performance that rises with the cache size and then falls off. The former case includes applications where the working set in each thread is small (H.264 and ODE) and where cache locality is poor for all cache sizes (Blender). In the latter case, performance rises sharply from the 4K dcache to the 8K dcache as the cache miss rate falls, and then drops off as the declining core count reduces the available execution resources.
Figure 7.8: Performance versus L1 cache size for the in2 configuration, by benchmark.
In Chapters 3 through 6, we looked at the high level architecture of the accelerator. In Chapter 7, we examined the macro-architecture of the cores in the compute array.

We continue the exploration of architecture tradeoffs in designing accelerator architectures by examining the low-level micro-architecture of the cores. The low-level micro-architecture of a processor is closely tied to the circuit-level implementation of that micro-architecture. In this Chapter, we introduce a methodology to co-optimize the micro-architecture and physical design of the chip. We use this methodology to gain a further understanding of the tradeoffs between area and performance.\textsuperscript{1}

\section{Optimization Problem}

Given the meta-architecture of the accelerator, there is a large design space for architecting each part. This design space includes the design of the compute cores, the interconnect, the caches, and the external interface. In Chapter 7, we focused on the macro-architectural design of the compute cores and the private L1 caches. In this Chapter, we revisit the core and cache design with a focus on the low-level micro-architecture and the physical design.

We continue with performance per area as our primary metric. However, \textsuperscript{1}This work was performed jointly with Omid Azizi and Mark Horowitz at Stanford University.
by extending the design space we can examine architectural changes that affect
clock frequency as well as IPC. In general, micro-architectural changes that in-
crease clock frequency will also increase area. For instance deeper pipelines re-
quire a greater number of pipeline latches. Moreover, implementation-level design
changes that increase clock frequency will also increase area, for instance through
more aggressive logical design or circuit sizing. In this chapter, we co-optimize
across macro-architecture (which affects IPC), micro-architecture (which affects
IPC and clock rate), and logic/circuit level implementation (which affects clock
rate) to find the configurations that maximize overall performance within given
area constraints.

8.1.1 Core design space

The design space for the cores in the xPU compute array includes the both the core
architecture and the physical design (circuits) implementing that architecture. To
find the most power-efficient core for our compute array, we need to examine both
design spaces.

Architectural design space

The architectural design space has many different parameters, which we parti-
tion into two groups. Macro-architectural design choices essentially change the
processing ‘algorithm’ of the architecture, and include superscalar processing,
multi-threading, out-of-order execution, the implementation of dedicated units
for processing complex or SIMD instructions (ISA extensions), and various forms
of speculative execution.

In this chapter, we are interested in core architecture for an accelerator. Hence,
we examine relatively narrow and simple cores. We examine the same single-issue
in-order, dual-issue in-order, and dual-issue out-of-order cores as in Chapter 7. Another macro-architectural feature we examine is multithreading. Fine-grained multithreading is an important technique for avoiding stalls due to memory latency, and particularly useful on throughput-oriented accelerator workloads. We include hardware implementation of trigonometric functions and exclude vector ALUs in all configurations, to narrow our search space.

In contrast, micro-architectural design choices affect performance, energy and area, but do not change the basic order of instructions. Micro-architectural parameters include sizing of caches, queues and other structures, the setting of pipeline latencies for different units, and the choice of clock frequency.

In this chapter, we evaluate trading off the relative latencies of different execution units, the fetch and decode stages, as well as the latency and size of the L1 caches. We assume a simple and power-efficient bimodal branch predictor, and examine the tradeoffs due to its sizing, along with the size of the branch target buffer.

We classify these architectural parameters into these two sets to signify how they can be explored and optimized. The micro-architectural parameters, being more numerical in nature, are more amenable to mathematical optimization techniques. The macro-architectural parameters, on the other hand, are more discrete in nature. Since they represent very different architectural topologies, we generate individual models for each of these macro-architectures and optimize them separately.

Circuit and logic-level optimization

Each component of the core micro-architecture is implemented by a circuit. The circuit may consist of any combination of a combinatorial logic circuit, latches, flip-flops, and SRAMs. We noted above that there is a design space for the
Figure 8.1: Area-performance tradeoffs for various circuits.

micro-architecture of a core. But even for a specific component in a given micro-
architecture, there is also a large trade-off space between energy, area and delay, resulting from choices in the logic and circuit-level design. This trade-off space needs to be characterized as it can have a considerable impact on the area and performance characteristics of the processor as a whole.

There are many design choices at the logic-level. Particular pieces of logic can be implemented in different ways, resulting in different delay and area characteristics. For example, a ripple-carry adder is small and consumes little energy, while a carry-lookahead adder is much faster, but requires more area and energy consumption. Logic functions throughout the microprocessor can be implemented in varying ways to trade off energy, delay, and area. Figure 8.1 shows the tradeoff between area and delay for several of the circuits we use in our model pipeline.

The choice of circuit style is another parameter in the implementation design space. Static CMOS circuits save energy, but are slower than dynamic and differential logic styles.

Circuit sizing is yet another low-level optimization. We can save area and power by using minimally-sized gates, at the cost of higher delay. Alternatively, we can size gates to minimize delay using techniques such as logical effort. This results in a much faster circuit, but comes at the cost of added area and energy consumption.
Figure 8.2: Area-performance design space. Each point is a possible design. Those points which lie on the frontier are optimal area-efficient designs.

We use RTL synthesis to generate circuit-level area and power models. Our synthesis methodology allows us to generate circuits with varying optimizations for area, power, and delay. These optimizations include logic optimizations, adder types, and some circuit sizing, but all circuits are implemented using static CMOS. Such optimizations cover the circuit-level design choices available to architects designing chips using a synthesis-based design flow.

8.1.2 Optimal design points

For a uniprocessor design, the optimization process is a search in a two-dimensional design space, with performance on one axis and the cost function (energy or area) on the other axis (Figure 8.2). In this space, there is a Pareto-optimal curve consisting of all the design points that maximize performance for any given cost. Equivalently this can be viewed as the set of points that minimize cost for any
given performance level. In the design process, the design of choice should be the point along this Pareto curve that is closest to our cost budget (in terms of area or power).

In a parallel architecture, the optimization criteria is the same. However, when our workload has sufficient parallelism, we can make a simplifying observation. As long as we are not constrained by overheads or by resources outside the processing core (for instance, memory bandwidth), then performance will scale linearly with the number of cores. We demonstrated in Chapter 2 that linear scaling of the parallel code sections is a reasonable approximation up to several hundred cores for all the benchmarks. In this case, we can optimize the design of the cores by looking at the performance versus area or power tradeoffs of a single core.

More concretely, we assume that we can fill the compute array of the xPU with as many cores as fit within our area budget. For instance, if our area budget for the entire compute array is 200mm², we can divide that area budget amongst 50 cores with a 4mm² area, or a 100 cores with a 2mm² area. In such a case, we maximize performance by choosing the core design that maximizes performance per square millimeter.

Naturally, this simplifying assumption does not always apply. Nevertheless, it is instructive to look at the performance, power, and area tradeoffs of a single core as part of the design process for a parallel architecture. The performance versus power tradeoff is the analysis we perform in the rest of this chapter.

Table 8.1 summarizes the design space.

8.2 Co-Optimization Methodology

To perform a design space exploration that spans both the architecture and circuit design, we first model each of the spaces independently, and we then establish the
Figure 8.3: Overview of the optimization framework. The architectural simulator and model fitting generate the architectural design space. The circuit tradeoffs define the circuit design space. The link is made through a higher level model which then feeds to the optimizer.
Table 8.1: Co-optimization design space

<table>
<thead>
<tr>
<th>parameter</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>issue width</td>
<td>1-2</td>
</tr>
<tr>
<td>scheduling</td>
<td>static or dynamic</td>
</tr>
<tr>
<td>threads</td>
<td>1-4</td>
</tr>
<tr>
<td>bpred size</td>
<td>0-4096 entries</td>
</tr>
<tr>
<td>BTB size</td>
<td>0-4096 entries</td>
</tr>
<tr>
<td>icache</td>
<td>2KB-32KB</td>
</tr>
<tr>
<td>dcache</td>
<td>4KB-64KB</td>
</tr>
<tr>
<td>pipeline</td>
<td>5-14 stages</td>
</tr>
<tr>
<td>cycle time</td>
<td>unrestricted</td>
</tr>
</tbody>
</table>

appropriate relationships between the two design spaces to create a higher-level model.

The architectural design space is a large multi-dimensional space; this design space grows very quickly, making it impractical to explore fully through simulation. To resolve this issue, we use statistical data fitting methods over a relatively small number of samples to create a model of the design space.

To determine the circuit tradeoffs of the underlying units, we rely on a combination of RTL synthesis and CACTI to model circuit area-delay tradeoffs. We finally integrate the architectural models with the circuit tradeoff data in a high-level model. This integrative model makes the connections between circuit delays, architectural cycles and the clock cycle time to characterize the full design space. Figure 8.2 provides an overview of the entire optimization framework.

8.2.1 Performance model

We again utilize the performance simulation infrastructure and benchmark code from Chapter 4. However, we simplify the simulation by modeling only a single core, rather than the full chip. Our simplification has the unfortunate drawback
of preventing us from examining the interaction between parallel threads. This interaction is important for accurately measuring performance, and is valuable in evaluating higher-level architectural tradeoffs. However, the effects of the interaction are less important when we are focusing on the micro-architecture at the core level.

We use this simulation environment to generate architectural performance data. For each benchmark application and each macro-architectural configuration, we determine the IPC resulting from a variety of micro-architectural configurations. We use statistical approaches similar to those proposed in several works [29,30], and generate an analytical form of the design space. The general approach constitutes randomly sampling the micro-architectural design space and using the results to generate an analytical model to characterize the effect of the parameters on the performance. By using a statistical approach, we can limit the number of simulations that are required to map a design space.

We use the IPC values generated by all of our random micro-architectural configurations and fit the data to a mathematical formula. The formula provides a mapping from an \( n \)-tuple of \( n \) parameters to an estimated IPC. While previous work used either cubic splines [30] or artificial neural networks [29], we chose to apply a convex fit instead. A convex fit is an approximation of the data using a posynomial function [31]. The convex fits were able to provide us with good fits—cross-validation errors of lower than 5% on average. More importantly, the posynomial mappings produce monotonic functions, which allows us to leverage convex optimization techniques during the optimization phase described below.

As with any method of statistical sampling, the expected error is smaller as we increase the number of sample points. By simplifying the simulation from the whole chip to a single core, we enable the use of a much larger number of samples.

We capture the lower-level micro-architectural parameters through this data-
fitted model. The model incorporates pipe latencies (in cycles for each pipeline stage), functional unit latencies, L1 cache sizes, branch predictor size and BTB size parameters. We generate separate models for the higher-level architectural design choices of multi-threading, dynamic scheduling, and superscalar processing, and for each benchmark.

8.2.2 Circuit models

To characterize the area-delay tradeoffs of circuits, we rely on a mixed approach of using verilog models for logic portions of the processor, and CACTI for memory units. For logic units, such as the ALUs and the decode stage, we use verilog implementations which we synthesize for different delay targets using Synopsys Design Compiler. We synthesize the circuits using 90nm technology, for which we have access to libraries. Design Compiler tries to meet each target delay while attempting to minimize area, yielding circuit implementations with varying topologies and gate sizings. By sweeping the delay target and creating designs with different delay and area characteristics, we are able to map out the tradeoff space between the area and delay of circuits.

We should note that parameters such as supply voltage, Vt or circuit style are not included in the circuit design tradeoff space we currently generate—only logic synthesis and gate sizing are circuit parameters. This restriction is only due to Design Compilers abilities and the limitation of our libraries. If we had more design data points that included designs of different circuit styles or with different threshold voltages, these could easily be incorporated into the circuit tradeoffs.

For the memory units, we use CACTI 6.0 to explore the SRAM design tradeoff space. CACTI internally performs an exhaustive exploration of several important SRAM design parameters. Because we are interested in the area-delay tradeoffs,
we modify CACTI to dump area and access latency information for all the configurations it explores during its search. We then analyze this data to generate the Pareto-optimal tradeoff curve of access time versus area for the desired SRAM.

Since cache size is one of our micro-architectural optimization parameters, we also need to include cache size in the circuit tradeoff space for caches. In these cases, therefore, we run CACTI for different cache sizes, gathering area and delay information for each size. We then compile this data into a 3-dimensional delay-size-area tradeoff. This way we essentially have a function that defines the area cost of a cache as a joint function of its size and delay. The optimizer can then use this information to jointly optimize cache size and delay.\(^2\)

### 8.2.3 Architecture to circuit link

We have described an architectural performance model and the circuit-level area-delay tradeoffs. As a final step before optimization, we must link these two models together to form the complete area-performance design space.

The architectural latencies indicated by the architectural simulator are linked to the physical delays of the circuit tradeoffs through the clock cycle time parameter. Ignoring any overheads caused by pipeline registers, the number of pipe stages is the delay of the circuit divided by the clock period, simply cutting the logic into stages. If we add delay overheads for pipeline registers we get following relationship

\[
N_i = D_i / (T_{cyc} - T_{reg}).
\]  

(8.1)

Here \(N_i\) is the number of pipeline stages in the architectural block \(i\), \(D_i\) is the sum of the physical delays of all circuits which make up the architectural block \(i\). \(T_{cyc}\) is the clock period and \(T_{reg}\) is delay overheads of registers (caused by register

---

\(^2\)The circuit characterization work was done primarily by Omid Azizi at Stanford University, though portions of the original RTL code were written at the University of Illinois.
setup and clock-to-q times). For datapath units, this equation holds; control logic, such as the computation of the next PC, must occur once a cycle, so we add the constraint that $N_i = 1$ in these cases.

Because we are allowing the number of pipe stages per unit to change in this formulation, we also need to account for the area overheads of all inserted pipeline registers. We can approximate the number of pipeline registers as a simple function of the number stages and the data width of the logic:

$$A_i = (N_i^\alpha \times W_i) \times A_{reg}. \quad (8.2)$$

Here $A_i$ is the area overhead due to pipeline registers in block $i$, $N_i$ is the number of stages in the block, $W_i$ is the width of the block, and $A_{reg}$ is the area cost of a single register. The parameter $\alpha$ allows for modeling super-linear pipeline register growth, and can be unique for each architectural block. Currently, we use $\alpha = 1.0$ for all units.

We finally input all this design space information—the convex architectural models, the circuit tradeoffs, and the links discussed in this section—into a geometric programming optimizer. The architect can specify the design objectives (performance target) and constraints (in this case an area budget). The optimizer then searches through the design space and finds the optimal micro-architectural configuration as well as its corresponding estimated performance. Because of our use of convex fits, the optimizer can use an efficient geometric programming algorithm, running about 20-30 seconds per search. Furthermore, because we input both architecture and circuit models, the optimizer is able to search through both the architectural and circuit design parameters to produce a globally optimal configuration.\footnote{The convex optimizer is the work of Omid Azizi and Mark Horowitz at Stanford University.}
8.3 Results and Analysis

Using the optimizer, we can gain a picture of design tradeoffs between area and performance by sweeping the optimization through a whole set of area targets. The results, showing the overall Pareto optimal area-performance tradeoff curves are in Figure 8.4 (optimizing for an average of all benchmarks) and Figure 8.6 (optimizing for individual benchmarks). These curves are broken down by issue width and number of threads, our two higher level architectural parameters.

Examining Figure 8.4, we can see that as our area budget increases from about 0.1 \textit{mm}^2 to 1 \textit{mm}^2, the most efficient architecture changes. For very area-constrained designs, the simplest single-threaded (ST), single-issue machine is the only architecture that can meet the objective. As we increase the area budget,
Table 8.2: Optimal configuration of five design points on the area-efficient frontier.

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
<th>Design 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>0.0122</td>
<td>0.317</td>
<td>0.411</td>
<td>0.452</td>
<td>0.591</td>
</tr>
<tr>
<td>issue width</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>scheduling</td>
<td>static</td>
<td>static</td>
<td>static</td>
<td>ooo</td>
<td>static</td>
</tr>
<tr>
<td>threads</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>clock freq</td>
<td>39MHz</td>
<td>1.0GHz</td>
<td>960MHz</td>
<td>867MHz</td>
<td>920MHz</td>
</tr>
<tr>
<td>icache</td>
<td>2KB</td>
<td>2KB</td>
<td>2.15KB</td>
<td>8.44KB</td>
<td>10.0KB</td>
</tr>
<tr>
<td>dcache</td>
<td>4KB</td>
<td>4KB</td>
<td>5KB</td>
<td>4.19KB</td>
<td>45.9KB</td>
</tr>
</tbody>
</table>

The optimal architecture gradually changes, first moving to a single-issue 2-way multi-threaded (MT2), then to a dual-issue superscalar MT2 design, then a dual-issue out-of-order MT2 design, and finally to a dual-issue 4-way multi-threaded machine (MT4).

Interestingly, the highest performing point is not dual-issue MT4, despite being the most complex macro-architecture. This is because the in-order configuration is able to scale to a higher clock frequency.

These results indicate that multithreading is initially a more cost-effective design choice in comparison to superscalar issue. This is expected, since the amount of latency and area overhead due to multi-threading is lower than for superscalar design, and yet it is an easy source of additional work for the processor which can be used to hide stalls in individual threads.

Neither the single-issue MT4 architecture, nor the dual-issue ST architecture is ever an area-efficient choice. This result is not completely unexpected either, as we expect an efficient design to be balanced. Both of these architectures allocate a lot of area resources to improving either superscalar issue or multi-threading, but ignore the other. This results in an imbalanced result which leaves more cost-effective sources of performance unexploited.

In Table 8.3, we examine some of the underlying micro-architectural and circuit parameters in more detail. We see the clock frequency rise rapidly as we push for
more performance. There is, of course, an area cost in terms additional pipeline
registers, but the benefits of frequency scaling seem to easily overshadow these
costs, as is indicated by the immediate jump from 39 MHz to 1.0 GHz from Design
1 to Design 2. In Designs 3 and 4, which are superscalar MT2 architectures,
physical limitations of more complex circuits limit frequency to under 1 Ghz, a
number lower than the simpler single-issue Design 2. Similarly, Design 5 is limited
to 920MHz.

Other things to note include the minimal amount of area allocated to the I-
cache in most cases versus increasing amounts of area spent on the D-cache. The
small I-caches are the result of the small instruction footprints of these applications
on average. We also see a gradually decreasing FP circuit delay. In some cases,
we see circuits that have reached their physical delay limit, as is the case with the
1.17 ns integer ALU.

So far, we have examined the area-performance design tradeoffs for an individ-
ual core. In parallel architectures, however, core count is variable. If we assume
linear performance scaling with the number of cores, then the optimal config-
uration that we should choose should be the one which maximizes the ratio of
performance to area. In the figures, a line is drawn from the origin through this
optimal configuration, which shows the area-efficient frontier for the whole system
under these assumptions. Designs on the line are efficient, while those points far
above the line have poorer performance to area ratios, and are hence are poorer
choices for an accelerator architecture.

Put another way, our area metric is performance per area, as it has been in
prior chapters. This can be seen in Figure 8.5, which shows the same axes as the
plots in Chapter 7. Once again the points from Table 8.3 are highlighted. Three
of the highlighted points represent the most efficient in1, in2, and out2 designs.
In this case, we see that the most efficient architecture is in1-MT2. However, as
Figure 8.5: Performance per area versus area per core tradeoff space based on an average of all six benchmarks. Area is in $\mu m^2$. The same five area-efficient designs are annotated as before.

was seen in the previous chapter, the most efficient in2 and out2 designs do not differ by a large margin, in this case 5% and 10% worse, respectively. The reason why in2 was preferable in the previous chapter and in1 is preferable here can be seen in the clock frequencies. Previously, we assumed all the configurations ran at the same clock rate. However, here the in1 configuration is able to achieve a higher clock speed than either of the optimal superscalar configurations.

Interestingly, these results indicate that for our set of benchmarks, both the single-issue MT2 architecture and the dual-issue MT2 architecture are good choices. While the single-issue MT2 core provides less performance, its lower area allows for more cores to be fit on die. On the other hand, the larger area of the dual-issue MT2 core means less cores can be included on the chip, but it mostly makes up for this through better individual core performance.

While the optimization of the architecture needs to be done for the set of all benchmarks (as we have done), looking at individual optimizations of each the benchmarks can be instructive on the internal workings of the optimization as
well. Figure 8.6 shows these results for the in-order architectures. Each of the
optimizations favor different aspects of the architecture—a reflection of the different
behaviors of the benchmark applications. For example, the MRI benchmark has
a lot of dependent FP operations. The FP unit being a longer latency function,
MRI favors a high degree of multi-threading in both the single-issue and dual-
issue cases to hide stalls that would otherwise degrade performance. As another
example, MRI, Facedetect and H.264 put virtually none of their area resources into
the I-cache size, POVray and ODE allocate area to I-cache size when performance
is pushed hard, while Blender depends most heavily on larger I-cache sizes to
achieve higher performances. This reflects how well the instruction footprint of
each of the applications fits in the I-cache, and thus how critical the I-cache size
is to performance.
Figure 8.6: Performance-area tradeoff space for each individual benchmark. Area is in $\mu m^2$. 
CHAPTER 9

POWER-PERFORMANCE OPTIMIZATION

In Chapter 8, we developed a methodology that enabled us to evaluate the performance and area tradeoffs of not only macro-architectural parameters but also low-level micro-architectural and circuit designs. This co-optimization process enabled us to revisit the results of Chapter 7 with the added consideration of clock speed.

In this Chapter, we extend the circuit and architecture model developed in the previous section to evaluate power. We describe the optimization problem for power-efficient architectures. We evaluate the power-performance tradeoffs across the design space for the accelerator cores and caches. Finally, we combine the area and power models to evaluate the tradeoffs between area and performance under varying power constraints.\footnote{This work was performed jointly with Omid Azizi and Mark Horowitz at Stanford University.}

9.1 Power Optimization Problem

In modern architectures, power is often the limiting constraint, rather than area. A power constraint can be the result of a cost calculation, as with area, where the constraint is a result of the cost of cooling or a power supply. In high performance architectures, it may be the result of a technical limit due to the thermal limits of the die. On the other hand, in a low power mobile architecture, the power constraint may be the result of a limited energy supply.
We can roughly divide the universe of power-constrained architectures into two classes. In one class, the chip has a set power budget, and the design objective is to maximize performance within this power budget. In the other class, the chip has a set performance target, and the design objective is to minimize the power or energy consumption.

Of course, real world designs may not fit perfectly into either class, and may be subject to a variety of additional design constraints. Nevertheless, we can roughly envision high-performance architectures as belonging to the first class, while viewing low-power architectures as belonging to the second class.

In the high performance architecture, the objective is to maximize performance for a given power budget. In other words, this means maximizing:

\[
\frac{\text{insts/sec}}{\text{watt}} = \frac{\text{insts}}{\text{joule}}
\]

In the low power architecture, we are given a performance target and the design objective is to minimize either power consumption (in watts) or energy consumption (total joules over the run time of some application). The optimization function is to minimize:

\[
\frac{\text{watts}}{\text{insts/sec}} = \frac{\text{joules}}{\text{inst}}
\]

In either case, the key metric is energy per instruction.

In a real design, we care about both power and area. Presuming the same high-performance architecture where we are given an area constraint, as in Chapter 7 and 8, we add an additional constraint in terms of total power consumption. With this dual optimization problem, the pareto optimal set is no longer a curve in a 2D tradeoff space, but rather a surface in a 3D tradeoff space.

Below we describe an extension of the methodology developed in Chapter 8.
that lets us optimize for both area and power.

9.2 Power Modeling

The addition of power to the methodology of Chapter 8 adds some complexity to the model. An overview of the extended methodology is shown in Figure 9.2.

The power consumption of the chip is divided into the static power consumption, the dynamic power consumption, and the power consumed by the clock tree. The static power consumption is the leakage power that the chip consumes whenever it is running. The dynamic power consumption results from the switching of the transistors in the logic circuits. The power consumption of the clock tree results from switching all of the clock driven parts of the pipeline, particularly the registers and pipeline latches.

9.2.1 Static power consumption

The model for static power consumption is a straightforward extension of the area model. Static power consumption results from leakage current from the gate to the drain in a transistor that is switched on, and from the source to the drain in a transistor that is switched off. Assuming a constant supply voltage $V_{dd}$, each transistor has a characteristic leakage level in the off state and a different characteristic leakage level in the on state, with the source to drain leakage in the off state being the larger amount.

In a CMOS circuit, each logic gate consists of a pull-up network of PMOS devices, and a pull-down network of NMOS devices. Typically, the NMOS and PMOS devices are sized in inverse proportion to the charge carrier mobility, in other words, so they both present the same I-V characteristic. Furthermore, regardless of the inputs to the logic gate, either the pull-up network or the pull-
Figure 9.1: Overview of the optimization framework with additional infrastructure for power. In order to determine dynamic power consumption, we need to count the number of operations going through each logic unit and SRAM.
down network will be in an off state. As a result, the leakage power consumption of the logic gate is roughly constant. Furthermore, if we look at the whole chip in aggregate, we can be fairly certain that at any given time approximately half of the PMOS and half of the NMOS transistors will be switched on and half switched off. As a consequence, the leakage power can be considered to be constant, dependent only on the layout of the circuit itself rather than the operation.

To obtain a value for the leakage power, we again use the Synopsys Design Compiler. As noted in Chapter 8, the RTL synthesis tool provides us the latency of each synthesized circuit. It also provides us with a value for the leakage power and the dynamic energy per operation. For the cache blocks, we obtain similar numbers for leakage power and dynamic energy per access from CACTI. Because we can consider the leakage power to be roughly constant, we use the numbers from the RTL synthesis and CACTI in the same manner as the area numbers.

9.2.2 Dynamic power model

The model for dynamic power consumption is much more complex. The dynamic power consumption is a result of the switching activity of the logic and the SRAMs within the chip. As a result, in order to calculate dynamic power we need to know the activity factor for each circuit.

Our RTL synthesis flow provides us with an estimate of the dynamic energy per operation or each version of each circuit that we synthesize. Unlike the leakage power value, however, this value cannot be taken as a constant. Different sequences of inputs to a logic block will result in different sets of transistors being switched, with a widely varying energy consumption per operation. The RTL synthesis provides us with an estimated energy consumption per operation assuming that half of the transistors switch between different operations.
We could calculate from the generated circuit the exact energy consumed for
every operation with every combination of inputs. However, for any non-trivial
circuit this is intractable, especially with a wide number of synthesized implementa-
tions of each logic block. Hence, as an approximation, we assume the dynamic
energy value produced by the synthesis tool is the energy consumption for every
operation performed by that unit.

The number of operations performed by each logic block in the pipeline is a
function of the operation of the pipeline, and is in turn affected by the instruc-
tions executed by each benchmark. For the logic blocks in the fetch, decode,
scheduler, and commit stages of the pipeline, the number of operations is given
by the number of instructions that pass through the stage. This is the number
of correct instructions that pass through plus the number of wrong-path instruc-
tions. Counting the number of correct instructions is simple: it is the number of
instructions we execute from the original benchmark. The number of wrong-path
instructions, on the other hand, is dependent on the micro-architecture, in par-
ticular on the misprediction rate and the latency of correcting a misprediction.
In our case, we assume that a BTB misprediction can be corrected after the in-
stuction passes through the decode stage (since the offset is part of the bits of
the instruction), while a branch misprediction can be corrected after the execute
stage (where the branch direction is determined). Indirect branches are always
resolved at the execute stage. Hence, for each micro-architecture we include BTB
and branch predictor mispredict rates. We estimate the number of wrong-path
operations as the number of mispredicts times the latency to the resolution stage.

For the logic blocks in the execution and memory access stages, the number
of operations is dependent on the number of instructions in the benchmark that
utilize that logic block. To measure this, we run each benchmark through a
functional simulator and determine the number of operations that pass through
each type of functional unit, in other words the number of adds, FP multiplies, shifts, loads, etc.

The dynamic power consumption of the caches is dependent on the number of accesses to each cache. The L1 instruction cache can be considered part of the fetch unit, and its activity can be calculated the same way. Likewise, the L1 data cache can be considered in the same manner as a functional unit, and its activity is dependent on the number of load and store instructions. We again use CACTI to determine the dynamic energy consumption for the L1 caches.

In Chapter 8, we neglected consideration of the data cache because, as in prior chapters, we fixed its size at 8MB and hence fixed its area. However, the power consumption of the gcache is dependent on the micro-architecture of the compute cores. In particular, it is dependent on the number of gcache accesses, which is the same as the number of misses in the L1 cache. To capture this, we determine the L1 cache miss count for each benchmark for each micro-architecture, and then use the results to model the cache miss rate as a function of cache size. As with the L1 caches, we use CACTI to determine the dynamic energy consumption per gcache access.

The I/O pins are an important factor in the dynamic power consumption. Unfortunately, our use of single-core simulation as well as the lack of a memory controller and PHY interface model prevents us from accurately characterize the behavior of the chip I/O. However, because we are assuming a fixed gcache size, we can presume that the number of accesses to main memory will be dependent more on the benchmark than on the micro-architecture. Hence, we believe that neglecting I/O power, while unfortunate, does not diminish the validity of the power model we use to evaluate the core architecture.
9.2.3 Clock power

The power of the clock tree is straightforward to model. This component of power consumption is dependent on the total input capacitance of all the transistors in the design which are connected to the clock tree. In other words, this is the total input capacitance of all the clocked elements such as level-sensitive latches and flip-flops. The major components of the pipeline that include such structures are the register file, PC, scheduler, and the pipeline latches. In the case of all the structures besides the pipeline latches, we can determine statically the amount of capacitance each will impose on the clock tree. Hence, these components can be handled in the same manner as the static power consumption.

The pipeline latches must be handled differently. The reason for this is because, as noted in Chapter 8, the insertion of area for pipeline latches occurs during the optimization and is not part of the circuit design space model. However, the contribution of the pipeline latches to the power consumption can be computed as a straightforward extension of the area calculation.

9.2.4 Optimization phase

The components of dynamic energy consumption (circuit blocks, functional units, caches) are modeled as functions of the benchmark and the micro-architecture. We take this dynamic power model and use it in the optimizer. In the optimizer, we characterize the total energy consumption of any architecture as:

\[ E_{total} = \Sigma_i (\alpha_i \times E_{i}^{dyn}) + T_{total} \times \Sigma_i P_i^{leakage} + T_{total} \times \Sigma_k P_k^{pipelatch} \]

In this formula, \( E_i^{dyn} \) is the dynamic energy consumption and \( \alpha_i \) is the activity factor for circuit \( i \). \( T_{total} \) is the execution time for a given benchmark on the given configuration in seconds. \( P_i^{leakage} \) is the leakage power consumed by circuit \( i \).
Finally, $P_{k}^{\text{pipelatch}}$ is the power consumed by pipeline latch $k$.

We use the same convex optimization technique as in Chapter 8 to find micro-architectural configurations on the pareto optimal frontier of the performance-power tradeoff.

### 9.3 Power Efficient Design

There is strong correlation between the area and power consumption of a core. Circuits with larger area will also have higher leakage power consumption. In addition, architectures of higher complexity will require each instruction to pass through more gates, hence consuming greater dynamic power.

Figure 9.2 plots the Pareto-optimal power consumption versus performance for an average of all the benchmarks. Figure 9.3 shows the results broken down by benchmark. Again, these figures are broken down by issue width and multi-threading.

Once again, we see that as our power budget per core increases, from about 0.77mW to 625mW, the optimal architecture changes. At the lowest power budget, the optimal architecture is in-order, single-thread. At 11mW, the most efficient (highest performing) design is in1 (11mW is the lowest power level at which all the architectures are represented). From 12mW to 23mW, single-threaded in2 is the most efficient. From that point to 65mW, in2-MT2 is the most efficient. Above 65mW, the optimal changes to out-of-order MT2.

In the case of a single core, the area and power tradeoffs look very similar. However, in an accelerator architecture the number of cores is a design choice. In this case, and if we assume linear performance scaling, then the optimum configuration should be the one which maximizes performance per watt (equivalent to instructions per joule). Analogous to the perf per area versus area per core
graphs, we plot perf per watt versus power per core in Figure 9.2.

The optimal point in the power case is very different from the optimal point in the area case. Several of the higher complexity core configurations were at or close to the area-optimal line. This indicates that performance per area scales up well even for complex cores. However, in the power optimized case, the optimal point is a low complexity design point. None of the multithreaded or superscalar configurations is able to match the performance per watt of the in-order single-threaded cores with minimal cache sizing.

This result is not very surprising, and is seen consistently across the benchmarks. The simplest architectural configuration minimizes the number of circuits that each instruction must pass through during execution. In addition, the simplest, minimally sized circuits minimize the size and number of the gates that implement each stage of the pipeline. Hence, the single-issue single-thread configuration is able to minimize the dynamic energy consumed per instruction. Moreover, the small cache sizes and small logic circuits minimize the amount of leakage power consumed.

In Figure 9.3, we plot the power versus performance for the individual benchmarks for the in-order configurations. We see the same trend of moving from simpler to more complex architectures with increasing power consumption.

9.4 Combined Area and Power Efficiency

Above, we see that the highest performance per watt is achieved by the simplest macro- and micro-architecture. However, the configuration with the best performance per watt is not the same as the one with the best performance per area. In a real design, there is both an area and an energy constraint. In the case where we want to examine both area and power constraints, the pareto-optimal
Figure 9.2: Performance-power tradeoff space for based on an average of all six benchmarks. Power is in mW.

Figure 9.3: Performance-power versus power consumption. Power is in mW.
tradeoff can no longer be seen as a curve in 2-dimensions, but rather as a surface in 3-dimensions.

The optimization framework allows us to set a combined constraint, with any two of area, energy per instruction, and performance. Performing this optimization over a range of values for all of the macro-architectures produces a collection of surfaces in a 3D space, with performance on one axis, power on one axis, and area on one axis (all numbers are per core). This data is plotted in Figure 9.4.

It is instructive to examine tradeoffs between two of the parameters at a time, holding the third parameter fixed. In Figure 9.4, we fix the power budget, and examine the area-performance tradeoff. Note that when the power budget is small, the optimal configuration is usually single-threaded in1. However, as the power budget per core rises to 120mW, in2-MT2 and out2 become more area-efficient.

In Figure 9.4, we fix the area budget per core, and examine the tradeoff between power and performance. When the area per core is small, the simplest macro-architectures are more power efficient (and in some cases only the simplest
Figure 9.5: Performance and power per core for pareto optimal configurations. Power is in mW.
ones can fit). As the area budget grows larger, the tradeoff becomes more interesting. At a lower performance level, the simple architectures are again more power efficient, but at higher performance levels the more complex architectures actually consume less power. This is because the more complex architectures can be run at a lower clock speed to achieve the same performance.

Finally, Figure 9.4 shows the tradeoff between area and power consumption for any given performance level. As in the fixed area and fixed power case, a larger performance level favors more complex architectures. The tradeoff between area and power consumption for each macro-architecture is itself interesting. Each tradeoff curve has a convex shape.

To further evaluate the reason for this convex tradeoff, we examined the data points on the in2-mt2 curve for the 0.5 insts/ns constraint, shown in Figure 9.4f. The data point at 50mW has a cycle time of 1.0ns and a dcache of roughly 8KB. The data point at 75mW has a cycle time of 0.91ns and a dcache of roughly 5KB. Finally, the data point at 100mW has a cycle time of 0.89ns and a dcache of roughly 4.3KB. In each case, the area savings is coming from the reduced size of
Figure 9.7: Performance versus area under varying power constraints.
Figure 9.8: Performance versus power under varying area constraints.
the dcache, while the increased power results in part from the increased clock rate and in part from a larger cost from global cache accesses. In this case, one can save power at the expense of area by increasing the size of the local cache, or save area at the expense of power by decreasing the size of the local cache.
Figure 9.9: Power versus area under varying performance constraints.
CHAPTER 10
DISCUSSION

In this Chapter, we discuss additional topics. We first review the insights of the previous chapters, describing a sample xPU architecture and the performance gain that can be expected from such an architecture. We then explore some of the limits of the studies in the preceding chapters. We examine the ways that the design space can be extended beyond what we have explored. We also discuss some of the limitations of our methodology.

10.1 Putting It All Together: An xPU Prototype

Pulling the optimal result from Figure 7.1, we obtain a configuration with 2-wide in-order issue and a 4K icache/8K dcache, a configuration with 573 cores and an average throughput of 165GOPs. Figure 10.3 shows the speedup of this configuration over a single-core 2.2GHz Opteron. On the parallel sections of VISBench (86% to over 99% of the execution time), the xPU attains an average speedup of 103X. Even when sequential code sections are factored in (up to 14% of execution time in Facedetect and ODE), the total speedup obtained remains over 6X.

Figure 10.1 shows the performance of a 2.2GHz Opteron, while Figure 10.2 shows the performance of our optimal xPU configuration. Both graphs plot the throughput in x86 instructions per second for the parallel sections of each VISBench application. The figure shows the xPU obtaining speedups averaging 100X
against the single-core processor. Even adding back the sequential execution portions, which range from <1% of execution time in H.264 ME and Blender, to 14% of execution time in Facedetect and ODE, we obtain the speedups shown in Figure 10.4.

The exploration of performance and area taking into account micro-architectural and circuit design spaces largely confirmed our results, though the in1 configuration slightly outperformed the in2 configuration due to a higher clock speed. Once again, multithreading and compromise cache sizes were ideal. And again, in1, in2, and out2 performed fairly close to each other.

Considering power consumption changed the optimal points. For a purely power efficient design, the simplest architecture is optimal. However, in the more
Figure 10.3: xPU performance versus 2.2GHz Opteron

Figure 10.4: Total CPU+xPU speedup over CPU only
realistic case with a combination of power and area constraints, the tradeoff is more interesting. With a power constraint typical of high performance devices, the tradeoff between area and performance looks similar to one without a power constraint, albeit excluding some extreme points. However, with a strict power constraint of the type one would find on a low-power architecture, the area efficient designs tend to be the ones with the simplest architecture.

The “optimal” configuration described above, however, is a compromise. Although all of the applications in VISBench are from the same “class” of applications, they often exhibit widely varying characteristics. As we saw in Chapters 5, 6, and 7, the different applications pull the architecture in different directions. Table 10.1 breaks down the features of our optimal architecture by the benchmarks that require them.

Table 10.1: Important architectural features for each benchmark.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blender</td>
<td>memory bandwidth</td>
</tr>
<tr>
<td></td>
<td>multithreading</td>
</tr>
<tr>
<td>MRI</td>
<td>FP throughput</td>
</tr>
<tr>
<td></td>
<td>multithreading</td>
</tr>
<tr>
<td></td>
<td>trig instructions</td>
</tr>
<tr>
<td>Facedetect</td>
<td>FP throughput</td>
</tr>
<tr>
<td></td>
<td>memory bandwidth</td>
</tr>
<tr>
<td></td>
<td>MIMD</td>
</tr>
<tr>
<td>ODE</td>
<td>FP throughput</td>
</tr>
<tr>
<td></td>
<td>synchronization support</td>
</tr>
<tr>
<td></td>
<td>MIMD</td>
</tr>
<tr>
<td>POVRay</td>
<td>FP throughput</td>
</tr>
<tr>
<td></td>
<td>MIMD</td>
</tr>
<tr>
<td></td>
<td>larger cache size</td>
</tr>
<tr>
<td></td>
<td>multithreading</td>
</tr>
<tr>
<td>H.264 SAD</td>
<td>maximum integer throughput</td>
</tr>
<tr>
<td></td>
<td>small cores</td>
</tr>
</tbody>
</table>
10.2 Extending the Design Space

In Chapters 5 through 9, we examined much of the design space for an accelerator architecture. However, we cannot cover the entire universe of possible design choices. In this Section, we examine some of the ways to extend the design space exploration.

The first part of the design space we can extend is at the application level. In Chapter 2, we perform small optimizations on the benchmarks in order to improve their performance in our simulated architecture. In particular, we changed the order of the parallel loops in MRI. We made changes to remove loop carried data dependences in H.264 in order to parallelize one of the inner loops. We made similar modest changes to the other benchmarks, but did not examine tradeoffs at the algorithmic level. An interesting extension to the work in this thesis would be a co-design or co-optimization of the algorithm and the architecture.

Another portion of the design space we can extend is the acceleration model itself. In Chapter 3, we assumed a uniprocessor CPU connected to a homogeneous, parallel xPU via a standard bus, with disjoint memories. In fact, there are a variety of ways to organize the system. First, CPU architectures have moved from single-core to multi-core. This allows us to examine a more complex acceleration model, where we are not merely decomposing the application into parallel and sequential parts but also decomposing the parallel sections into portions that can run on a multicore CPU and on the xPU. Another possible alternative to our model is one where the CPU and xPU have a shared memory, or going even further where the CPU and xPU are on the same die, as is proposed for the AMD Fusion. We don’t evaluate any overhead from disjoint memories, so our analysis still applies for a system architecture with a shared memory. However, with joint memory we could also extend our acceleration model to allow cooperating threads...
to run in parallel on the xPU and CPU.

We can extend the design space for the architecture of the compute array. Currently, we only consider a homogeneous architecture. However, non-homogeneous architectures are common, particularly in mobile devices. Mobile devices often contain a system-on-a-chip with a variety of specialized units for different applications.

In this study we did not examine the design space for on-chip interconnects. In Chapter 5 and 7, we assumed that an interconnect could be built within the 100mm² area budget that provides sufficient aggregate bandwidth so as not to be a performance bottleneck. Consequently, we assumed that modeling bank contention and bandwidth limitations at the global cache was sufficient to obtain accurate performance estimates. Our architecture does not support point-to-point communication, and our experience suggests that the basic assumption that the interconnect is feasible is reasonable. Nonetheless, the design of an area and power efficient on-chip interconnect is an interesting problem in its own right.

In our examination of architecture/circuit co-optimization in Chapters 8 and 9, we limited ourselves to the use of a synthesis-based design flow. The state-of-the-art in synthesis tools has advanced to the point where synthesized circuits are able to obtain multi-gigahertz clock frequencies. At the same time, the design cost for custom circuits has grown dramatically. Given the design costs, it is very reasonable to restrict our study to an exploration of synthesized circuits. Furthermore, the design optimization process within the synthesis tools facilitates the quantitative optimization process we present and enables us to clearly see the tradeoff curve without doing a hand search of possible circuits. Nonetheless, custom circuit design retains significant advantages over synthesis for very high speed designs. Custom designed circuits are able to take advantage of techniques to reduce delay, power, and area that synthesis tools cannot. A possible extension
of this work could include an examination of custom circuit design.

10.3 Limitations

Above, we discussed some of the limitations of our design space, as well as possible extensions. In this Section, we examine some of our methodological limitations and discuss how they impact our results.

One limitation of our studies is that our code was not optimal. For the VIS-Bench benchmarks, we chose open-source applications that have a substantial commercial deployment. However, these open source applications do not represent the state-of-the-art. Proprietary applications that are written specifically for accelerator architectures often employ optimizations that are not available to the open source community.

One example of this can be seen in comparing the physics simulation engine in ODE to that used by AGEIA PhysX. ODE uses a less sophisticated collision detection kernel than PhysX. Furthermore, less effort has been devoted to parallelizing the phases of ODE that are difficult to parallelize than is the case for PhysX.

Similar limitations can be seen in comparing Blender and POVRay against GPU-optimized rendering tools. While GPUs are designed for real-time rendering, they have also been used for high-quality, non-real-time rendering, for instance NVIDIA Gelato [32]. In this case, the GPU-accelerated tool uses a combination of techniques from realtime and non-realtime graphics, for instance calculating visibility using the GPU z-buffer while computing most special effects in software on the CPU.

In each of these cases, we see algorithmic differences that are a result of optimization for a specific architecture. These optimizations can make a significant
performance difference, but they also make the application less suitable for use in a wide-ranging design space exploration. For the wide-ranging exploration presented in this thesis, the more generic code of our open-source applications is more suitable than the heavily architecture-specific code seen in state-of-the-art proprietary equivalents.

On the other hand, different limitations can be seen in comparing Blender and POVRay against commercial software-only rendering tools. In this case, the commercial tools offer not improved performance but rather a greater variety of effects for improving image quality. Open source applications often contain a lesser feature set than commercial applications, but in our case nonetheless include those features that are the most frequently used. Despite the feature differences, the open source tools serve as a viable experimental proxy.

Another limitation on our software side is the code generation. We rely on a standard compiler, gcc, to generate our machine code. Moreover, we compile to a standard ISA, x86, that is not necessarily the ISA that would be ideal for an accelerator. The reason for this platform is related to our applications. We are using full-featured applications which require a modern OS platform including recent versions of Linux and its system libraries.

Our evaluation of word-level SIMD in Chapter 7 is constrained by compiler technology and by the vectorizability of our code. We compile our benchmarks using a recent version of gcc. While gcc supports vectorization and in fact generates substantial vector code on our benchmarks, it is not as aggressive as the best available compiler. Hence, our results on subword SIMD, while based on standard compiler technology, are not an upper bound. From examining the generated code, we know that gcc vectorizes loops in 5 of the 6 benchmarks (MRI, facetdetect, povray, ode, and blender). The Intel compiler does a better job vectorizing povray, ode, and blender, but still vectorizes only a small portion of the code. We
see 10-15% performance improvement on native runs using Intel binaries versus gcc. Even if this were entirely due to superior vectorization it would not alter the conclusions of Chapter 7.

Another limitation of our code generation is the code scheduling. In our comparison of in-order and out-of-order pipelines, the quality of code scheduling can have an effect. Poor code scheduling can inhibit the performance particularly on superscalar in-order pipelines, such as 2-wide in-order. Our code generation schedules for the Pentium 4, an architecture which also requires good code scheduling. However, the Pentium 4 schedule is not an exact match for our in-order pipelines. Furthermore, the x86 ISA itself is better suited to out-of-order pipelines than other ISAs. In light of this limitation, the competitive performance we show for in-order architectures in Chapters 7 to 9 is a strong result.

Our methodologies for performance, area, and power consumption also have limitations.

In our performance simulation, we do not model overheads for task creation and destruction. The only thread startup cost we model is flushing dirty data from caches. VISBench tasks are fairly course-grained (typically hundreds of thousands of instructions as described in Chapter 2). As a result, even at the chip level we are only initiating a new task every few to every few hundred cycles. Hence, we believe our assumption is valid; essentially we can build a task management system that alleviates task creation overheads as a major performance factor. In fact, our assumption is justified by prior work. [13] demonstrates a low-overhead task distribution system in hardware. Going even further, [14] demonstrates a task distribution system built entirely in software, on the same incoherent shared memory model as ours, that shows a performance overhead of under 300 cycles, less than 1% of the typical task length in VISBench.

We do not fully model the on-chip interconnect. We assume that a network can
be designed that allows essentially full utilization of the global cache, achievable by somewhat overprovisioning the network. Instead, we model the interconnect as having a fixed latency and model bank contention at the global cache. As a result, we are still modeling the bandwidth limitation of the cache hierarchy, which is the important factor in performance.

The area model presented in Chapter 4 has some limitations. As mentioned in that chapter, we do not take into account circuit-level tradeoffs, and we account for metal interconnects as just a constant factor. In Chapter 7, we accounted for these limitations by computing an error margin. Because of the low-level circuit issues, it is impossible to create a high-level area model that provides perfect precision.

The area and power models presented in Chapters 8 and Chapter 9 do take into account the circuit-level design space. However, the co-optimization methodology presents new limitations, some of which were discussed in the respective chapters. For one, we are naturally limited to the design space that is covered by the synthesis flow and CACTI. Furthermore, we synthesize pipeline components rather than the whole processor, and this can have an effect on such factors as interconnect delay between pipeline stages, which we cannot account for. Finally, the manner in which we insert pipeline registers is essentially an estimate.

The co-optimization process itself is limited by the fact that it models the design spaces as continuous rather than discrete. As a result, it produces results such as an optimal cache size of 27KB, which may not be area or performance optimal in themselves but rather are close to an optimal in the true, discrete design space. Nonetheless, the trends we determine hold even for the discrete optimization space.
Related work to our study falls into three categories: benchmarking of parallel applications, accelerator architectures, and design space exploration for parallel architectures.

11.1 Benchmarking of Visual Computing

A number of benchmark suites has been published in the area of parallel computing. An early effort to provide a parallel benchmark suite was SPLASH [33], which consists of numerical kernels and some parallel applications ported to a variety of parallel architectures. Another parallel benchmark suite is SPEComp [34], which consists mainly of OpenMP versions of SPECfp. The 13 Dwarves [35] are a set of kernels important to parallel applications. Our benchmark suite differs from these general purpose parallel benchmarks in that it is targeted at a more specific application area. We also seek to study a new and emerging applications rather than traditional HPC workloads.

Other early examples of benchmark suites include MediaBench [36] and EEMBC [37]. MediaBench targets multimedia and network applications that were emerging applications at the time. EEMBC targets applications, including visual applications, that are important for embedded processors rather than high-performance processors.

More recently, Intel has identified Recognition, Mining, and Synthesis (RMS)
[38] as important application areas for upcoming parallel architectures. RMS represents application areas meant to capture emerging uses of computing power, like VISBench, but consists of a broader set of areas. The PARSEC benchmark suite [39] was developed to target these application areas. PARSEC targets a somewhat different application area than VISBench.

Ad-hoc benchmarks have been published for specific application areas included in our study. Benchmarks for graphics rendering include 3DMark by Futuremark, used to measure real-time rendering performance. The graphics benchmarks we use in this paper differ in that they are aimed at high-fidelity, non-real-time rendering. PhysicsBench [5] is a benchmark suite for physics simulation which we use in this paper.

11.2 Accelerator Architectures

Accelerator architectures are becoming more and more important as a number of vendors have proposed or are providing accelerator chips. These make up the general notion of an xPU [1], a co-processor accelerator than a traditional graphics chip. Intel has produced an 80-core VLIW research chip [40]. IBM produces the Cell processor, used in the PlayStation 3 [41]. A number of smaller companies have also introduced parallel accelerators, including Tilera [42], ClearSpeed [43], and Ambric [44].

GPUs are moving in the direction of general purpose accelerators. The use of GPUs as general purpose accelerators is the subject of GPGPU research. Luebke et. al. [45] provide a survey of this work. NVIDIA’s CUDA [11] and AMD’s CTM [46] provide programming interfaces for GPGPU programming.

Application specific architectures have been proposed for some of the applications we examine. Ageia produces the PhysX chip [47], intended for physics
simulation, which consists of an array of vector cores. Another proposed physics architectures is ParallAX [5], a heterogeneous architecture containing both coarse and fine grained cores. NETRA [48] was a parallel architecture for computer vision, consisting of a programmable crossbar and clusters of processors that could operate in SIMD, MIMD, or systolic modes. The Ray Processing Unit [49] is an accelerator for ray tracing.

11.3 Area and Power-Efficient Architecture

As such, our study extends a large body of work on area-efficient architecture. An early example of area-efficient architecture was the RISC project. The motivation for RISC was to put a whole processor onto a single CMOS chip [50]. Likewise, the motivation for early CMP work was to put a multiprocessor onto a single chip, thus achieving high-throughput on multiple applications [51].

A number of CMP studies have focused on area-tradeoffs for maximizing throughput/area or throughput/watt. In one early study, [15], Huh et. al. compare fixed-area CMPs made up of either in-order or out-of-order cores, and find that on SPEC workloads the out-of-order configurations with fewer cores still provide higher throughput. Kumar et. al. examine the microarchitectural optimization of cores [52] and on-chip interconnects [53], again using SPEC workloads. In [54], Balfour and Dally also examine on-chip communication networks. Monchiero et. al. [16] studied the impact of core count, cache hierarchy, and interconnects on CMP power consumption. Li et. al. [55] perform a design space exploration with core count and core complexity under various power and area constraints. Hsu et. al. [56] study the cache design space for many-core CMPs.

These previous studies have influenced our methodology for modeling processor area. However, they have generally examined a very different set of archi-
tectural parameters and using a very different set of applications. Whereas we examine architectural choices like multithreading and SIMD execution, the prior work tends to emphasize interconnection networks. Both our work and the prior work examine cache hierarchy, dynamic scheduling, and core count. The previous work also focuses on purely general purpose architecture, and consequently uses general purpose workloads, whereas we examine an accelerator architecture that lies between general purpose and application-specific.

There has been a large body of work on microprocessor optimization, both in the architecture and circuit design domains. Some recent works in large-scale architecture optimization focus on modeling the high-dimensional architecture design space through statistical sampling and model fitting \cite{29, 30}. These models facilitate design space exploration/optimization, and have been used to generate insightful design space studies. While these approaches are very powerful (our convex fitting approach even being inspired from them), these works do not incorporate the circuit design trade-offs into their analysis. Thus, our work stands apart from these and earlier works by its integrated analysis of the processor design space. In other words, our architecture optimization is aware of the design tradeoffs and constraints of the underlying circuits.

There have been a select number of works that have performed integrated architecture-circuit analyses in the past. Zyuban et al. present a ‘unified methodology’ for examining tradeoffs at the microarchitectural and circuit levels \cite{57}. They introduce a notion of hardware intensity which they use to connect issues that affect both circuits and architecture. Their work is lower-level than our work and is more analytical in nature. We rely more on architectural simulations and actual data from RTL synthesis. Furthermore, their work does not span to include application/workload effects.

A number of other studies look solely at performance versus pipeline depth
[58–61]. We also incorporate pipeline depth in our analysis, but our approach goes further by optimizing across other micro-architectural design parameters such as cache sizes.
CHAPTER 12

CONCLUSION

The continuous scaling of semiconductor manufacturing technology under Moore’s Law is enabling the rise of architectures with increasing performance density. The rise of such architectures is providing researchers in a variety of computing fields with opportunities to enable new applications.

Many of the new, emerging applications fall into the class of visualization, interaction, and simulation (VIS). This class includes applications such as graphics rendering, video encoding, simulation, and computer vision. These applications are ideally suited for accelerators because of their parallelizability and demand for high throughput.

Research into architectures for applications in this class is hampered by the lack of standardized, readily available benchmarks. In this thesis, we presented VISBench, a benchmark suite to serve as an experimental proxy for VIS applications. VISBench contains a sampling of applications and application kernels from traditional visual computing areas such as graphics rendering and video encoding. It also contains a sampling of emerging application areas, such as computer vision and physics simulation, which are expected to drive the development of future accelerator architectures.

We presented a characterization of the VISBench application suite. We examined the instruction mix presented by the applications. We examined the parallel scalability of the applications, including characteristics such as thread length and data sharing patterns. Finally, we examined how these characteristics affect the
tradeoffs in designing an accelerator architecture.

Using insights from VISBench, we developed a high-level architecture for a visual computing accelerator. We refer to this accelerator, more general purpose than a heavily graphics-oriented GPU, as an “xPU”. We developed a non-coherent shared memory model for the accelerator which eliminates the complexity of cache coherence while still providing sufficient support for data communication and synchronization. We evaluated the execution model for the computational portion of the xPU and made a case for an architecture consisting of discrete cores executing in a MIMD fashion.

Given these high level choices, we used VISBench to explore the design space at the core level. We found that a design made of small, simple cores achieves much higher throughput than a general purpose uniprocessor. Further, we found that a limited amount of support for ILP within each core aids overall performance. We found that fine-grained multithreading improves performance, but only up to a point. We found that vector ALUs provide a poor performance to area ratio. Finally, we found that cache sizing is an important but application-specific design parameter, and that for overall performance the best cache size is an intermediate one.

We proposed a methodology for performing an integrated optimization of both the micro-architecture and the physical circuit design of the cores and caches. We used statistical sampling of the design space for evaluating the performance of the microarchitecture and RTL synthesis to characterize the area-power-delay of the underlying circuits. This integrated methodology enabled a much more powerful analysis of the performance-area and performance-power tradeoffs in the accelerator architecture. We used this methodology to find the optimal design points for an accelerator architecture under area constraints and power constraints. Our results indicated that more complex architectures scale well in terms of perfor-
mance per area, but that performance per watt is minimized by the simplest architectures.

As an enduring contribution, this thesis offers three elements. First, it presents a benchmark suite of commercially relevant applications that can serve as targets for future accelerator architectures. Second, it introduces a process for gaining architectural insights from the benchmarks, at multiple levels of the abstraction hierarchy of hardware design. Finally, it determines a set of architectural features, in Table 10.1, that aids performance for each of the benchmarks, and presumably for many other related applications. In sum, this thesis provides a guide for the understanding and development of a new class of general-purpose accelerator architectures for visual computing.
REFERENCES


Aqeel Mahesri received a Bachelor of Science degree in Electrical Engineering and Computer Science from the University of California at Berkeley in May 2002. He continued his studies at the University of Illinois at Urbana-Champaign, receiving a Master of Science in Computer Science in May 2004, and continuing to a Doctor of Philosophy in Computer Science in May of 2009.

Aqeel’s primary research interest lies in computer architecture, with a focus on massively parallel architectures for visual computing. During his studies, he has researched and published work on a variety of topics within computer architecture, including decoupled architectures, power-efficient mobile computing, microprocessor reliability, and benchmarking for performance evaluation.

Aqeel served as a research assistant in the Advanced Computing Systems and later the Rigel research groups, led by Professor Sanjay J. Patel. He also served as a teaching assistant in courses on computer architecture and computational theory. During his studies, he performed internships at Procket Networks in Milpitas, CA, at NVIDIA Corporation in Santa Clara, CA, and at Intel Corporation in Hillsboro, OR.