Architectures and Limits of GPU-CPU Heterogeneous Systems

by

Henry Ting-Hei Wong

B.A.Sc., University of Toronto, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2008

Abstract

As we continue to be able to put an increasing number of transistors on a single chip, the answer to the perpetual question of what the best processor we could build with the transistors is remains uncertain.

Past work has shown that heterogeneous multiprocessor systems provide benefits in performance and efficiency. This thesis explores heterogeneous systems composed of a traditional sequential processor (CPU) and highly parallel graphics processors (GPU). This thesis presents a tightly-coupled heterogeneous chip multiprocessor architecture for general-purpose non-graphics computation and a limit study exploring the potential benefits of GPU-like cores for accelerating a set of general-purpose workloads.

Pangaea is a heterogeneous CMP design for non-rendering workloads that integrates IA32 CPU cores with GMA X4500 GPU cores. Pangaea introduces a resource partitioning of the GPU, where 3D graphics-specific hardware is removed to reduce area or add more processing cores, and a 3-instruction extension to the IA32 ISA that supports fast communication between CPU and GPU by building user-level interrupts on top of existing cache coherency mechanisms.

By removing graphics-specific hardware on a 65 nm process, the area saved is equivalent to 9 GPU cores, while the power saved is equivalent to 5 cores. Our FPGA prototype shows thread spawn latency improvements from thousands of clock cycles to 26. A set of non-graphics workloads demonstrate speedups of up to 8.8×.

This thesis also presents a limit study, where we measure the limit of algorithm parallelism in the context of a heterogeneous system that can be usefully extracted from a set of general-purpose applications. We measure sensitivity to the sequential performance (register read-after-write latency) of the low-cost parallel cores, and latency and bandwidth of the communication channel between the two cores. Using these measurements, we propose system characteristics that maximize area and power efficiencies.

As in previous limit studies, we find a high amount of parallelism. We show, however, that the potential speedup on GPU-like systems is low (2.2× - 12.7×) due to poor sequential performance. Communication latency and bandwidth have comparatively small performance effects (<25%). Optimal area efficiency requires a lower-cost parallel processor while optimal power efficiency requires a higher-performance parallel processor than today's GPUs.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my thesis supervisor Tor Aamodt for the guidance throughout my masters career. Thanks also go to Steve Wilton and Sathish Gopalakrishnan for their comments and feedback during the thesis defense process.

Thanks also goes to Wilson Fung, off whom I had bounced many ideas and had many thought-provoking and insightful discussions over the last two years, and for proofreading the thesis.

I would also like to thank Intel Microarchitecture Research Lab for providing me with the opportunity to conduct research in an industrial setting. Particular thanks goes to Hong Wang and Anne Bracy for the guidance throughout the project.

I acknowledge the financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) during my masters program.

Statement of Co-Authorship

Pangaea (Chapter 2) was work done while on internship at Microarchitecture Research Labs, Intel Corp. The architecture was conceived by colleagues at MRL, and refined by me to suit implementation constraints as work proceeded. The FPGA prototype of the heterogeneous system was designed and built by me from existing CPU and GPU code. Data analysis was performed by me. The Pangaea manuscript was co-authored with colleagues at MRL and Tor Aamodt.

The limit study (Chapter 3) experiment was designed collaboratively by Tor Aamodt and I. The research, data collection and analysis, and manuscript preparation were done by me.

Chapter 1

Introduction

1.1 Motivation

As we continue to be able to put an increasing number of transistors on a single chip, what we build using these resources continually changes. One question is whether we should devote increasing resources to improving serial performance using larger processors or to increasing parallelism with more processors.

Amdahl [1] argued in 1967 that the correct method of improving computer performance was to build ever-faster uniprocessors, as the now-famous Amdahl's Law shows that a program's sequential fraction will dominate runtime if sequential performance improvements were neglected. The argument for building large uniprocessors was again made in 1997 [2], arguing that multiple processor cores ought to be placed on separate chips because communication latency is more tolerable *between* processors than *within* a processor, and that we should build the largest uniprocessor that is possible on a single chip, and use multiple chips for parallelism.

Gustafson [3] argues against Amdahl's Law in 1988 that increasing problem sizes result in increasing parallelism with a roughly constant sequential fraction, thus justifying large multiprocessor systems. A more recent appeal for single-chip multiprocessor parallelism was made in 1996 [4], noting that improving uniprocessor performance was achieving diminishing returns.

One example of the effective use of a single-chip multiprocessor system is in commodity 3D graphics accelerators. Graphics processing units (GPUs) are chips specialized to implement 3D graphics acceleration as dictated by 3D APIs like OpenGL or DirectX. The

abundance of parallelism in the workload naturally led to highly-parallel architectures.

GPUs became programmable by adding support for various types of programmable shaders in the graphics APIs and the development of high-level shader programming languages [5, 6]. Being programmable, GPUs began to be interesting for unintended uses [7, 8, 9]. Many of these early general-purpose (GPGPU) applications involved linear algebra.

Originally, GPGPU applications were limited to using graphics APIs, using programmable shaders embedded within the standard graphics pipeline defined by the APIs. For non-graphics purposes, this presented an unnecessary overhead layer between the application and the hardware doing the computation. This was also difficult to use, as it required the developer with no interest in graphics to understand graphics APIs.

Some of the overhead of graphics APIs is reduced with the introduction of compute-oriented APIs like Nvidia's CUDA [10] or ATI's Stream SDK [11], which allow access to the computation cores in a GPU bypassing the graphics API.

These APIs allowed the GPU to be programmed directly and relatively easily, but the GPU still is treated as a device, its memory space separate from that used by the CPU. GPUs are still difficult to program as the developer needs to manually copy data between the CPU and GPU, and programs on the GPU can not invoke operating system services. The EXO [12] architecture reduces programmer effort by providing the appearance of a shared memory space and supports exception handling by signaling the CPU to perform address translation and exception handling on behalf of the GPU. The GPU hardware still maintained all the hardware overhead needed to support graphics APIs.

In Chapter 2, we address some of these problems and present a heterogeneous multicore chip architecture, *Pangaea*, that further tightly-couples GPU cores with the CPU. Pangaea proposes to integrate the processing cores from a GPU with a CPU on a single chip, removing hardware support for legacy graphics APIs (reducing overhead), and sharing cache and memory hierarchy between the two core types. Fast communication between CPU and GPU cores would be done using proposed x86 ISA extensions which enable user-

level interrupts implemented on top of existing cache coherence mechanisms. Removing graphics-specific hardware results in significant area and power reductions.

Although we show that Pangaea succeeds in reducing communication latency between CPU and GPU, the performance impact is small for highly-parallel applications that can tolerate communication latency (and memory latency too, as the GPU philosophy goes).

This motivates an exploration of designs *and applications* around the design point presented by Pangaea. We want to find the potential impact of communication and GPU core characteristics on performance, as well as which types of applications are sensitive to these design choices. In our limit study (Chapter 3), we attempt to measure the potential impacts of communication latency and bandwidth for general-purpose applications, without requiring that they be manually parallelized.

Our limit study finds that when algorithms are optimally partitioned between GPU- and CPU-like processors, the performance impacts of communication latency and bandwidth are not large. Despite traditional wisdom, we notice many implementations of applications for GPUs do not report CPU-GPU communication as a major limiter of performance [13]. We also find that despite the optimistic assumptions made in our limit study, many general-purpose applications do not have sufficient parallelism to create thousands or millions of threads to be used by the GPU to tolerate long register read-after-write latencies, leaving GPUs under-utilized due to insufficient parallelism in the application.

1.2 Thesis Contributions

We summarize the contributions of this thesis here:

- We propose an architecture for GPU compute only, which achieves tighter coupling with the CPU by sharing memory hierarchy and leveraging cache coherency mechanisms for fast user-level signaling. (Chapter 2)
- We prototype a Pangaea design on an FPGA using RTL code from an IA32 CPU core and the Intel GMA X4500 GPU.

- We show that removing support for the legacy graphics pipeline and sharing of memory hierarchy with the CPU result in significant area and power savings, and significantly reduced communication latency.

- To explore the design and application space around Pangaea, our limit study abstractly models a GPU and CPU heterogeneous system, using an algorithm that extracts parallelism from applications and optimally schedules instructions onto the two processor types. (Chapter 3)

- Our limit study models GPU register read-after-write latency and a constrained CPU-GPU communication channel, and shows that the register read-after-write latency of GPU cores is a significant limiter of achievable speedup, while CPU-GPU communication latency and bandwidth only have minor effects on achievable performance.

- We use area and power efficiency metrics with our limit study data to propose designs for GPU core properties that maximize efficiency. We observe that for area efficiency, GPU cores with higher register read-after-write latency than today's GPU cores are optimal, while optimal power efficiency requires GPU cores with lower read-after-write latency.

The rest of the thesis is structured as follows. We present a background on GPU computation and related work in the following sections (Sections 1.3 and 1.4). In Chapter 2 we propose and evaluate the Pangaea architecture. In Chapter 3 we explore the CPU-GPU design space in our limit study. We conclude in Chapter 4.

1.3 Background on GPU Computation

1.3.1 GPU Organization

Figure 1.1 shows the general system organization of a modern discrete GPU, used in conjunction with a traditional CPU-based system.

The intended use of graphics processors is to accelerate 3D graphics rendering. This is often done using a standard graphics API like OpenGL or DirectX [14]. Graphics APIs present a pipeline where vertices are manipulated, rasterized, textured, and output to the frame buffer to be displayed on a bitmapped display. The DirectX 10 [14] pipeline stages (*Input Assembler* through *Output Merger*) are shown in Figures 1.1 and 1.2. Although it is possible to render 3D graphics in software on the CPU, the processing performed by the rendering pipeline is usually offloaded to the graphics processor (GPU), which is designed specifically to implement the rendering pipeline, and provides a large performance improvement.

When rendering, the graphics driver running on the host operating system is responsible for interfacing between the graphics API runtime and the GPU hardware. Data and commands are sent from driver software running on the CPU to the GPU over the PCI Express bus, which is then processed by the hardware.

1.3.2 Programmability

Modern rendering pipelines have increased in programmability. Figure 3.1 shows the rendering pipeline stages which are programmable. The *Vertex Shader* and *Geometry Shader* stages take each vertex as input then runs a short program transforming each one. The *Pixel Shader* stage takes each pixel fragment as input, and likewise runs a short program on each to transform it. As vertices and pixels are essentially mutually independent, multiple instances of shaders can execute in parallel, which encourages highly parallel hardware designs. In a typical GPU, there is an array of multi-threaded, SIMD processing cores which executes the shader programs. These cores are architected to exploit the available parallelism to get the most throughput in the least area. The graphics driver is responsible for translating vendor-neutral shader code into the device-specific instruction set.

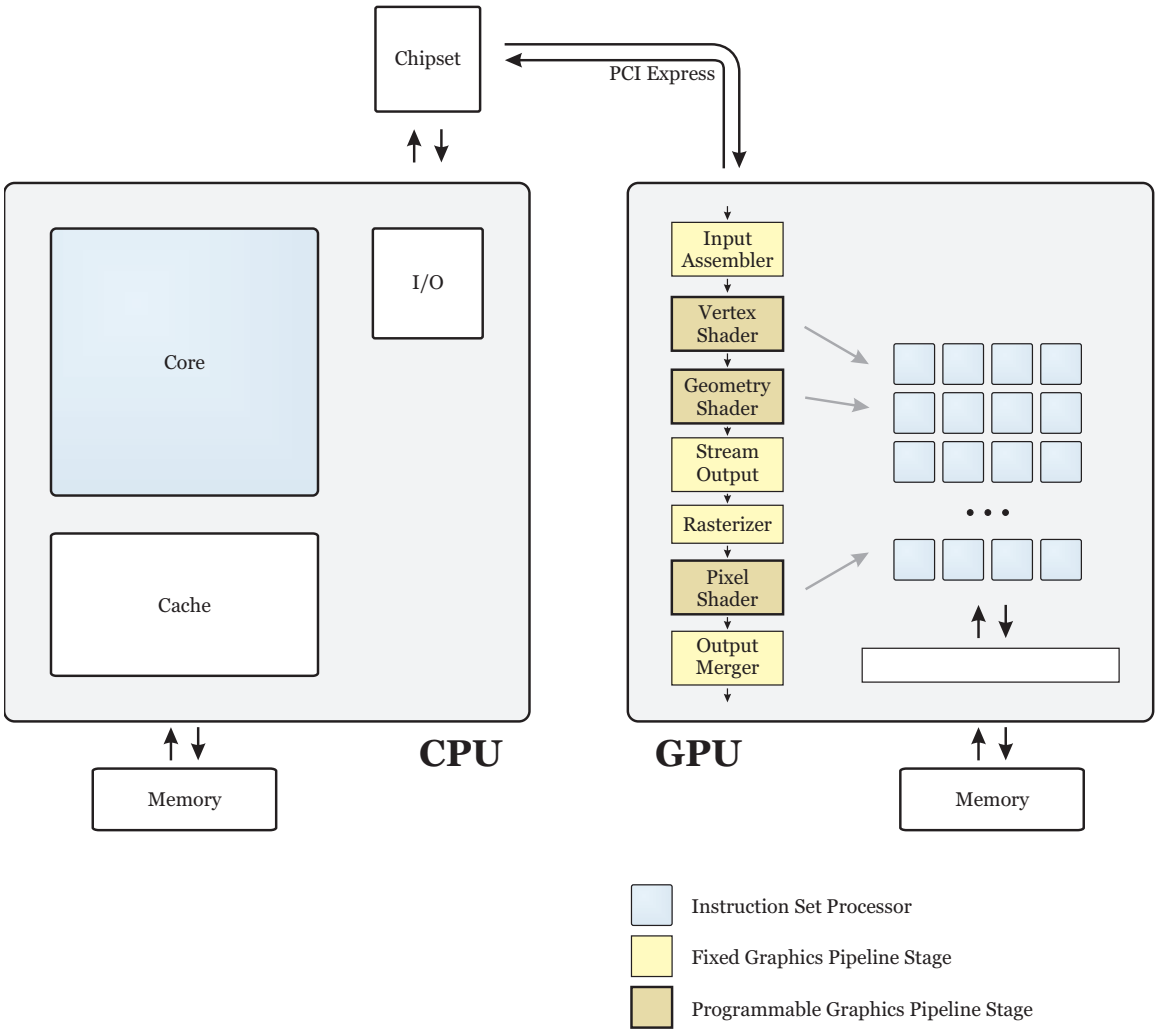

Figure 1.1: Discrete GPU Organization

1.3.3 Computation

Early efforts at GPU computation took advantage of the programmable shaders. Computation problems were mapped into graphics rendering problems, and shaders were written which achieve the desired computation rather than output a rendered image. This approach leaves the hardware organization unchanged. Drawbacks include the requirement that a computation be mapped onto vertex and pixel primitives, and that some stages of the rendering pipeline are irrelevant to the desired computation and is thus wasted area, power, and possibly performance.

1.3.4 CUDA, ATI Stream

Some of the problems of mapping computations to shaders can be avoided by allowing the use of the array of cores by programs without using the graphics API, allowing computation to be specified directly in programs without remapping into vertices and pixels [10, 11]. The hardware organization remains nearly unchanged from Figure 1.1, but a new mechanism to use the array of cores needs to be added (not shown). Since graphics remains the main usage of GPUs, all of the hardware to support graphics is left intact and simply left unused as appropriate.

1.3.5 CPU-GPU Integration

As seen in Figure 1.1, a discrete GPU is typically attached to the CPU system over an off-chip PCI Express bus. Discrete GPUs typically have their own memory, physically separate from the CPU's memory, and data is transferred by copying over the PCI Express bus using DMA. Current integrated GPUs are integrated into the chipset and use a portion of the CPU's memory instead of having its own dedicated memory. This results in lower cost in exchange for a performance penalty. The general architecture is otherwise similar.

One issue with off-chip buses is their high latency and limited bandwidth [15]. This thesis explores in Chapter 3 the impact that this latency and limited bandwidth can have on performance. One solution to this issue is to integrate the GPU on the same chip as the

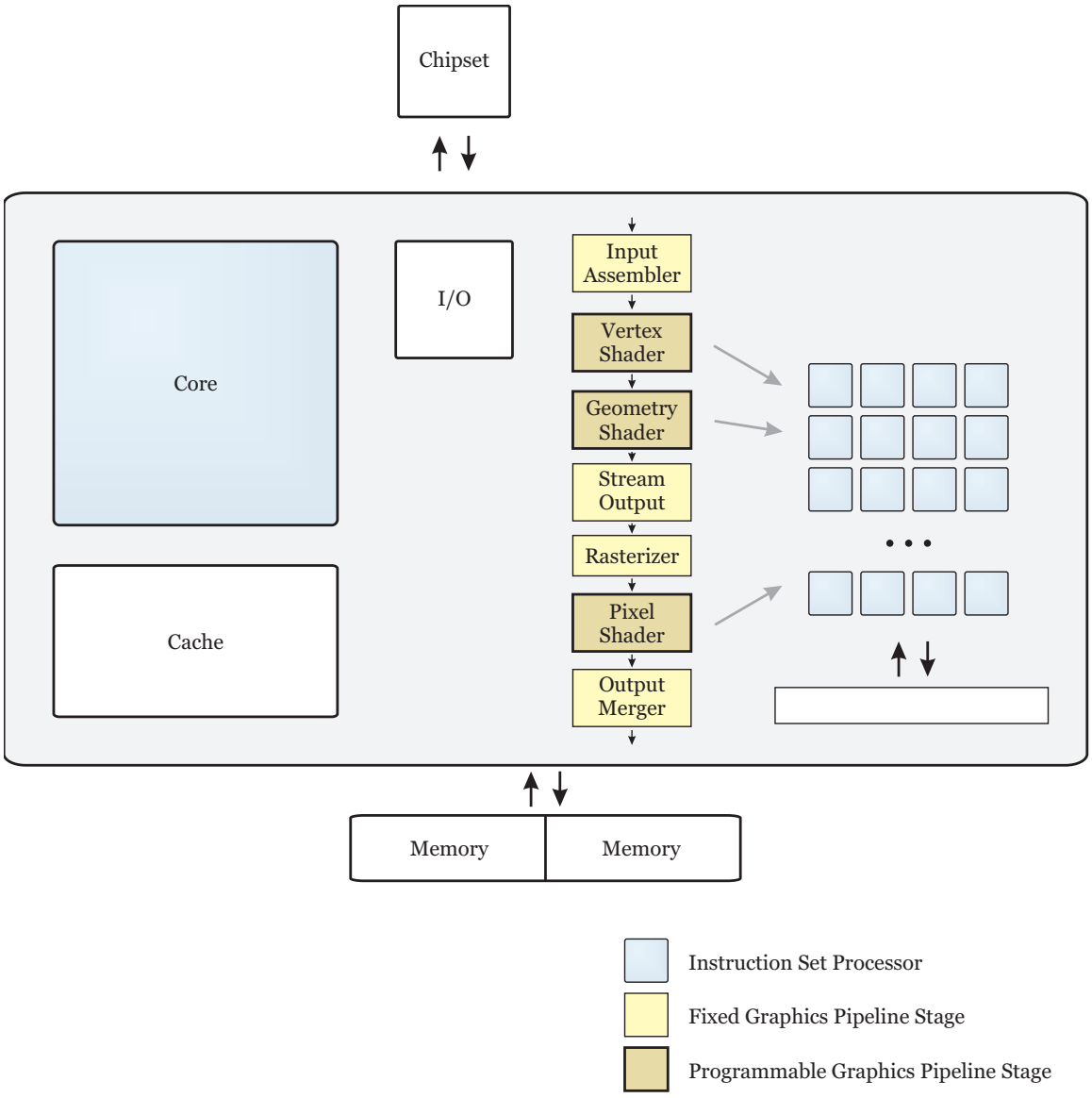

Figure 1.2: Single-chip "Fused" CPU-GPU Organization

CPU.

There are planned products which integrate CPU and GPU on the same chip [16]. One possible arrangement is shown in Figure 1.2. Unlike the discrete GPU, communication between CPU and GPU is now on chip, and no longer across an off-chip PCI Express bus. As the GPU still appears to the system as a graphics device, memory is logically separate, although it would likely share the physical memory array with the CPU. Graphics-specific hardware is left intact, as such a system is intended to support 3D rendering. This arrangement is referred to as the *fused* GPGPU organization in Chapter 2.

The Nvidia CUDA and AMD Stream enhancements allowing direct use of the core array are also applicable to single-chip CPU-GPU multiprocessors.

1.3.6 Pangaea Preview

Extending the integration further, Pangaea proposes to support only general-purpose computation, and removes graphics-specific hardware to save area and power. It is also proposed that memory address space be shared between the parallel cores and the CPU, and that communication occur through cache coherence mechanisms. Sharing a cache allows fast data sharing and reuse between CPU and the GPU-like core array. See Chapter 2 for details.

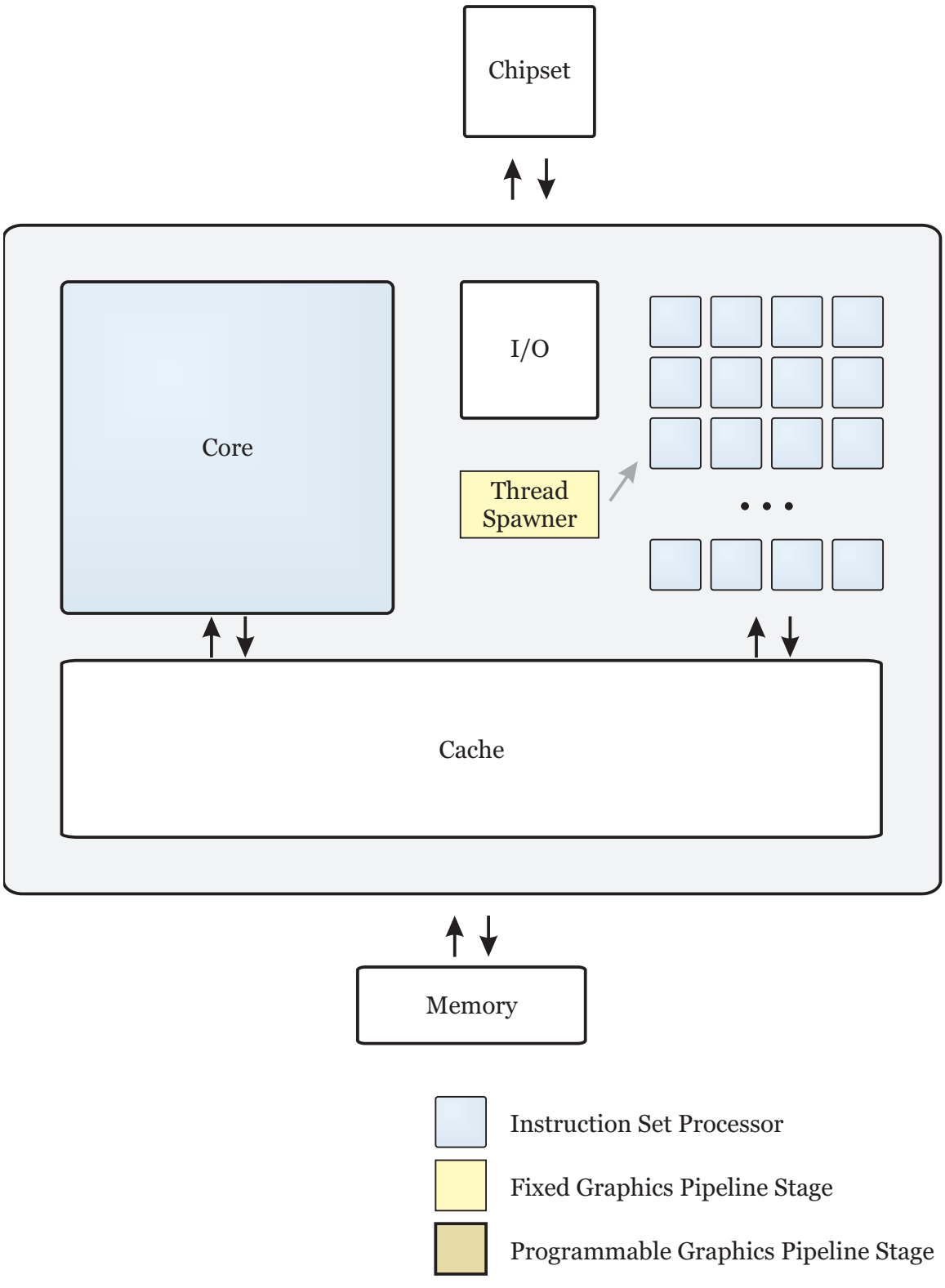

Figure 1.3: Single-chip Pangaea CPU-GPU Organization

1.4 Related Work

1.4.1 Multicore Processors

In the beginning von Neumann [17] created the processor. "I'll take one," Amdahl said [1]. "I'll take more," said Gustafson [3]. "Give me some on a chip," says Olukotun [4]. Patt objected [2], "Only one on a chip." AMD [18] and Intel [19] disagreed and gave everyone two on a chip. And then more [20], perhaps even a thousand someday [21].

Homogeneous Multicore Current personal computer systems use homogeneous multicore processors, where two or more processors of the same microarchitecture are placed on a single chip. Oluktun et al. [4] make the argument that given diminishing returns of improving single-processor performance, multiple processors on a chip provide better performance in many cases.

Indeed, single-processor scaling has been shown historically to be fairly poor, with performance increasing approximately proportional to the square root of resources (area, power) used. This is Pollack's Rule [21]. Multiple cores have the potential to scale throughput linearly as the area and power consumed.

Heterogeneous Multicore As opposed to homogeneous systems, heterogeneous systems use two or more types of processor cores in a system. As programs have both parallel and sequential phases, it makes intuitive sense that in order to prevent Amdahl's Law from limiting speedup, a fast sequential processor should exist to run sequential portions of programs.

Kumar *et al.* showed that single-ISA heterogeneous systems using different microarchitectures implementing the same instruction set can reduce power consumption [22] at little performance loss compared to a single high-performance processor. They also showed improved performance [23] when running multiprogrammed workloads compared to a homogeneous multicore system of equivalent area.

Later work arrive at similar conclusions. Growchowski *et al.* [24] report improved performance under a fixed power budget, and Annavaram *et al.* [25] demonstrate a prototype.

Kumar *et al.* also try to choose processor core sizes in a four-core single-ISA heterogeneous system to optimize for one set of multiprogrammed workloads [26]. They conclude that a homogeneous multiprocessor system outperforms the best homogeneous system, and chooses parameters that optimize the cores for the chosen workload.

Past work has focused mainly on single-ISA systems and single-threaded or multiprogrammed workloads with few cores. In CPU-GPU heterogeneous systems the ISAs of the two processors differ: The GPU uses an ISA specialized for supporting graphics. The GPU is also far more parallel than the ~4 thread contexts considered in earlier work, while the single-thread sequential performance of GPU relative to the CPU is far greater than between CPUs of varying microarchitectures (typically 100x between GPU and CPU).

This thesis concerns heterogeneous systems with a single-threaded CPU core and a highly parallel GPU-like core, with parallelism and performance characteristics more extreme than earlier work considers.

1.4.2 GPU Compute

Ever since graphics processors (GPU) became programmable, there has been interest in using it for computation beyond its intended use of 3D graphics rendering. Initially, custom computation was essentially hacked into the standard graphics pipeline by transforming the desired computation into a rendering computation, but recent hardware has allowed programming the GPU directly.

GPGPU using shaders General-purpose programming of GPUs started when GPUs incorporated various programmable "shaders" into the graphics pipeline. OpenGL and Direct3D defined virtual machine-like models for shaders. High-level languages were also defined to ease programming. Examples include GLSL [6] for use with OpenGL, HLSL for Direct3D, and Nvidia's Cg [5] which can target both APIs.

Early GPGPU work usually involved building algorithm primitives like sorting or applications using linear algebra [27, 28]. Sparse matrix solvers [7, 8] seemed especially popular. One example of an application of sparse matrix solutions is quadratic cell placement [9].

General-purpose Languages Programming within a graphics model when working on a general-purpose program is inconvenient. There have been some attempts at abstracting away the graphics pipeline by providing a more general programming environment (usually C-like) that is then compiled for execution by the GPU shaders. Examples include Brook for GPUs [29] and Accelerator [30]. Note that although these languages can abstract away the graphics pipeline from the developer, the compiled program is still compiled to shader programs within a graphics pipeline, and retain any related overheads.

ClawHMMER [31] is an example application using Brook.

Unified Shaders The increasing number stages of the graphics pipeline that became programmable (vertex, pixel, and, in DirectX 10, geometry shaders) eventually led to the definition of a unified shader model, instead of different shader models for each type of shader. DirectX 10's Shader Model 4.0 [14] requires unified shaders.

GPU Compute Hardware Unified shaders with increasing programmability naturally led to the desire to use them outside the graphics pipeline. Using the shader cores directly can improve performance by avoiding translation/compilation from vendor-neutral shader code into the native instruction set of the GPU cores. It also avoids the programming complexity of trying to fit a general-purpose problem into a graphics-specific programming model.

Nvidia's CUDA [10] and ATI's CTM [11] allow programming of their respective GPUs without using the graphics API. CUDA presents a C-like programming environment, reducing programming complexity. However, much of the complexity caused by constraints inherent in the GPU cores' architecture remains that the developer still needs to consider,

for example SIMD width, VLIW scheduling of instructions, or matching the number of threads and registers used to hardware capabilities.

The relative ease of use of CUDA led to many applications [13, 32, 33]. It is interesting to note that the communication latency and bandwidth over PCI Express is not a major limiter on performance [13], an observation also seen in our limit study (Chapter 3).

Unified Programming Model Even with CUDA, programming GPUs is not easy. As the GPU is viewed as a device in a CPU-based system, memory spaces are separate and the developer needs to manage data copying between CPU and GPU memory. Programs on GPUs also are unable to use OS services on their own.

Efforts to improve ease of programming have involved defining new programing model that can target both CPU and GPU implementations [34, 35], so that a program need only be written once. Another approach involves making the GPU programming model more similar to the existing CPU model, for example the Exoskeleton Sequencer (EXO) architecture [12]. EXO gives the illusion of a shared memory space between GPU and CPU and allows exception handling in GPU code using Address Translation Remapping and Collaborative Exception Handling, where the CPU handles page faults and other exceptions on behalf of the GPU.

Unified Hardware: CPU-GPU integration Commercial products integrating the GPU and CPU (single package or single chip) are expected to be available soon [16]. Although likely lower in raw performance than a discrete GPU due to more stringent area and power constraints, CPU-GPU integration offers interesting possibilities in supporting a unified programming model.

Pangaea goes further than simple integration of GPU and CPU, but proposes that the GPU be specialized for parallel compute only. Pangaea follows the EXO shared-memory model, but also physically shares parts of the memory hierarchy, and adds mechanisms for fast communication between CPU and GPU.

1.4.3 Limit Studies on parallelism

Single Processor One of the early limit studies on available parallelism in programs was done by Wall [36]. He reports fairly low available parallelism (~5-7), due mainly to branch mispredictions. The poor results due to branch prediction prompted Lam and Wilson's study [37] on the effects on available parallelism of various control flow constraints (prediction and/or condition dependence) and allowing multiple fetch streams, using a similar methodology. This study finds that multiple fetch streams and condition dependence analysis unlock much of the parallelism available (40 for integer and 561 for floating-point applications).

Thread-Level Speculation There have also been limit studies modeling the potential benefits of thread-level speculation machines [38, 39]. These placed additional constraints on the model used by Lam and Wilson's work.

Homogeneous Multiprocessor With the recent trend toward multicore processors, there is an interest in evaluating the potential performance in parallelizing existing applications. The limit study by Vachharajani *et al.* [40] uses a similar methodology to earlier work, but adds various scheduling algorithms that distribute instructions from a single thread among processors in a homogeneous multiprocessor system. They find that there is significant parallelism in ordinary applications when parallelized for homogeneous multiprocessor systems, but find that communication latency between cores is a significant limiter of performance.

We are not aware of prior limit studies that attempt to model *heterogeneous* systems where some processors are faster than others, which is characteristic of GPU-based compute systems.

References

[1] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings vol. 30*, pages 483–485, 1967.

[2] Y.N. Patt, S.J. Patel, M. Evers, D.H. Friendly, and J. Stark. One Billion Transistors, One Uniprocessor, One Chip. *Computer*, 30(9):51–57, Sep 1997.

[3] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of ACM*, 31(5):532–533, 1988.

[4] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proc. 7th international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM.

[5] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[6] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language, version 1.20. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf, Sept. 2006.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 917–924, New York, NY, USA, 2003. ACM.

[8] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Guilherme Flach, Marcelo Johann, Renato Hentschke, and Ricardo Reis. Cell placement on graphics processing units. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 87–92, New York, NY, USA, 2007. ACM.

[10] Nvidia. Compute Unified Device Architecture Programming Guide Version 2.0. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf.

[11] J. Hensley. Close to the Metal. ACM SIGGRAPH 2007 courses, course 24 article 7, 2007.

[12] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

[13] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[14] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[15] B. Holden. Latency Comparison Between HyperTransport and PCI-Express in Communications Systems. http://www.hypertransport.org/docs/wp/Latency_Comparison_HyperTransport_PCIe_in_Communications_Systems.pdf.

[16] AMD. 2007 Financial Analyst Day. http://download.amd.com/Corporate/MarioRivasDec2007AMDAnalystDay.pdf, 2007.

[17] John von Neumann. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pensylvania, June 1945.

[18] AMD. AMD "Shatters The Hourglass" With The Arrival Of The AMD Athlon 64 X2 Dual-Core Processor. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~98647,00.html, May 2005.

[19] O. Wechsler. Inside Intel Core Microarchitecture: Setting New Standards for Energy-efficient Performance. ftp://download.intel.com/technology/architecture/new_architecture_06.pdf, 2006.

[20] AMD. Family 10h AMD Phenom Processor Product Data Sheet. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf, November 2007.

[21] S. Borkar. Thousand Core Chips — A Technology Perspective. In *Proc. 44th Annual Conference on Design Automation*, pages 746–749, 2007.

[22] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi,

and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[24] E. Grochowski, R. Ronen, J. Shen, and Hong Wang. Best of both latency and throughput. *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 236–243, Oct. 2004.

[25] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI Throttling. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[26] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM.

[27] Dinesh Manocha. General-Purpose Computations Using Graphics Processors. *Computer*, 38(8):85–88, 2005.

[28] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. EUROGRAPHICS 2005 STAR - State of The Art Report A Survey of General-Purpose Computation on Graphics Hardware, aug 2005.

[29] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[30] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to

program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, 2006.

[31] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementatio. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 261–272, New York, NY, USA, 2008. ACM.

[33] L. Nyland, M. Harris, J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems 3*, 2007.

[34] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM.

[35] Michael D. McCool, Kevin Wadleigh, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of GPUs using the RapidMind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[36] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM.

[37] M. S. Lam, R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th International Symposium on Computer Architecture*, pages 46–57, 1992.

[38] Nakajima Akio, Kobayashi Ryotaro, ANDO HIDEKI, and SHIMADA TOSHIO. Limits of Thread-Level Parallelism in Non-Numerical Programs. *IPSJ Transactions on Advanced Computing Systems*, 47(SIG 7(ACS14)), May 2006.

[39] Katsuhiko Metsugi and Kazuaki Murakami. Limits of Parallelism on Thread-Level Speculative Parallel Processing Architecture. In *International Workshop on Information and Electrical Engineering (IWIE2002)*, May 2002.

[40] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, D. Connors. Chip multi-processor scalability for single-threaded applications. *ACM SIGARCH Computer Architecture News*, 33(4):44–53, 2005.

Chapter 2

Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor[1]

2.1 Introduction

As Moore's Law pushes for a more rapid pace of silicon development and even higher degree of on-die integration, the number of cores in future multi-core designs will continue to increase. As the microprocessor industry rapidly marches into the era of multi-core design, the future generation of multi-core processors will essentially become an integration platform with not only numerous cores, but also different types of cores varying in functionality, performance, power, and energy efficiency [1]. Fundamentally, ultra low EPI (Energy Per Instruction) cores are essential to scale multi-core processor designs to incorporate a large number of cores. One approach to improving EPI by an order of magnitude is through heterogeneous multi-core designs, which have a small number of large, general-purpose cores optimized for instruction-level parallelism (ILP) and many more special-purpose cores optimized for data-level parallelism (DLP) and thread-level parallelism (TLP). Such a multi-core processor offers opportunities for non-graphics application software and usage models [2, 3, 4, 5, 6, 7] to aggressively exploit the combination of ILP, DLP and TLP.

In this paper we present Pangaea, a synthesizable design of a heterogeneous chip mul-

[1] To appear in *Proceedings of Parallel Architectures and Compilation Techniques (PACT 08), Toronto, Ontario, Canada*
Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, Hong Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Proceedings of Parallel Architectures and Compilation Techniques*, October 2008, Toronto, Ontario, Canada.

tiprocessor (CMP) that integrates IA32 CPU cores with GPU multi-cores. Architected to support general-purpose parallel computation, Pangaea goes beyond the current state-of-the-art CPU-GPU integration that physically "fuses" an existing CPU design and an existing GPU design on the same die. In Pangaea, new enhancements are introduced to both the CPU and GPU to support tighter architectural integration, improved area and power efficiency, and scalable modular design. On the CPU side, a three-instruction extension to the IA32 ISA supports a fly-weight communication mechanism between the CPU and the GPU and a fine-grain shared memory collaborative multithreading environment between the IA32 CPU cores and the GPU multi-cores. This ISA enhancement allows an IA32 thread to directly spawn user-level threads to the GPU cores, bypassing most of the legacy graphics specific fixed-function hardware (*e.g.*, input assembler, vertex shader, rasterization, pixel shader, output merger [8]) found in a modern GPU design. This can achieve a two-order of magnitude reduction in thread spawning latency. On the GPU side, a state-of-the-art existing GPU design (Intel GMA X4500 [9]) is rearchitected to significantly reduce the fixed-function hardware, which is traditionally dedicated to support 3D-specific graphics processing. The legacy front-end is replaced with a small FIFO controller that can buffer and dispatch GPU threads spawned by the IA32 CPU. The legacy back-end is replaced by sharing the memory hierarchy between the IA32 CPU and the GPU multi-cores. The removal of the legacy fixed-function hardware can result in area savings (on a 65 nm process) equivalent to nine additional GPU cores (of five hardware threads each) and power savings equivalent to five GPU cores.

This paper makes the following contributions:

- We describe the architecture support and microarchitecture reorganization of both CPU and GPU in Pangaea to achieve tighter architecture integration and power and area efficiency of a heterogeneous CMP design.
- We detail a fully functional synthesizable implementation of a Pangaea design, based on production quality RTL from an ILP optimized IA32 core and the GMA X4500 GPU.

- We present an in-depth analysis of architectural tradeoffs between the Pangaea design and a state-of-the-art design that physically fuses existing CPU and GPU on the same die.
- We report significant performance gains for a set of media and non-graphics parallel applications by employing Pangaea to harvest ILP, DLP and TLP, achieving speedups of up to 8.8×.

The rest of the paper is organized as follows. Section 2.2 reviews related work. Section 2.3 provides a background on baseline GPU architecture. Section 2.4 introduces the architectural enhancements to the IA32 CPU and the microarchitectural reorganization of the X4500 GPU to support tighter architectural integration. Section 2.5 details the implementation of Pangaea and assesses the key architectural tradeoffs in terms of power and area savings compared to the state-of-the-art CPU-GPU design with physical fusion. Section 2.6 evaluates the performance of a set of general-purpose applications on a Pangaea hardware prototype on an FPGA-based emulator. Section 2.7 concludes.

2.2 Related Work

We adopt the distinction between *asymmetric* and *heterogeneous* multi-core designs from related work [7, 10]. All cores in an *asymmetric* multi-core design are of the same ISA but differ microarchitecturally. In a *heterogeneous* multi-core design, some cores feature different ISAs in addition to microarchitectural differences. Prior work on multi-core architectures has demonstrated significant benefits for both power/performance and area/performance efficiency [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. However, those studies primarily focus on asymmetric rather than heterogenous multi-core design.

Heterogeneous multi-core designs integrate cores of different ISAs and functionalities and can potentially lead to even further improvement in power/area/performance efficiency. IBM Cell's heterogeneous architecture [20] offers a mix of execution elements optimized for a spectrum of functions. Applications execute on this system, rather than a col-

lection of individual cores, by partitioning the application and executing each component on the most appropriate execution element. The *exoskeleton sequencer* (EXO) architecture [7] presents heterogeneous cores as MIMD function units to the IA32 CPU and provides architectural support for shared virtual memory, ensuring efficient data sharing across the heterogeneous execution elements.

Recently, both AMD and Intel have made public announcements on their upcoming mainstream heterogeneous processor designs for the 2009-10 timeframe. These processors will be on-die integrations of the IA32 CPU and their respective GPUs, which are traditionally found on the chipset or in discrete GPU cards. The so-called *fusion* integration physically connects existing CPU and GPU designs and supports some level of cache sharing between them, while the designs themselves remain unchanged. Although the integrated GPU is intended to run the legacy graphics software stack, there has been growing interest in harvesting such heterogeneous multi-core processors to accelerate non-graphics applications. Furthermore, there have been extensive efforts to provide programming model abstractions and runtime support to ease the otherwise daunting task for programmers to use heterogeneous multi-cores [4, 5, 6, 21].

Although heterogeneous integration is key to Pangaea, Pangaea is different than fused designs in that it supports a tighter-coupled integration through lightweight user-level interrupts. Bracy *et al.* discuss these lightweight user-level interrupts and utilize existing coherency logic to provide simple, preemptive, low-latency communication between cores [22]. Many other microarchitectures also support preemptive communication [23, 24, 25, 26, 27, 28, 29, 30, 31].

2.3 Background

This section provides some necessary background on GPU architecture and defines terminology that will be used in the following sections. Figure 2.1 depicts an architectural organization of a modern GPU. It consists of three major components (from left to right):

- **Front-end:** a graphics-specific pipeline ensemble of fixed-function units, each corre-

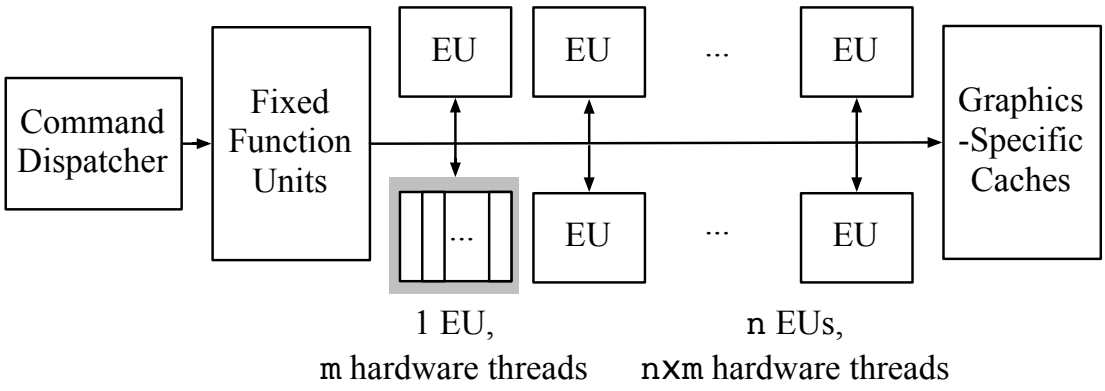

Figure 2.1: Organization of the Intel GMA X4500.

sponding to a certain phase of the pixel and vertex processing primitives, *e.g.*, command streamer, vertex fetcher, vertex shader, clipper, strip/fan, windower/masker, roughly in correspondence to DirectX's input assembler, vertex shader, rasterization, pixel shader, and output merger [8], respectively. The front-end translates graphics commands into threads that can be run by the processing cores.

- **Processing multi-cores:** hereafter referred to as Execution Units (EU). This is where most GPU computations are performed. Each EU usually consists of multiple SMT hardware threads, each implementing a wide SIMD ISA. In the GMA X4500, each thread supports 8-wide SIMD operations.
- **Back-end:** consists of graphics-specific structures like render cache, etc., which are responsible for marshalling data results produced by the EUs back to the legacy graphics pipeline's data representation.

Non-graphics communities are understandably interested in harvesting the massive amount of thread level and data-level parallelism offered by the EU to accelerate general-purpose computation, for which the graphics specific hardware front-end and back-end are largely overhead. The GPU is managed by device drivers that run in a separate memory space from applications. Consequently, communication between an application and the GPU usually requires device driver involvement and explicit data copying. This results in

additional latency overhead due to the software programming model.

Pangaea assumes the EXO execution model that supports user-level shared memory heterogeneous multithreading and an integrated programming environment such as *C for Heterogeneous Integration* (CHI) [7] that can produce a single fat binary consisting of multiple code sections of different instruction sets that target different cores. The focus of our study of the Pangaea design space is to investigate architectural improvements beyond the physical on-die fusion of existing CPUs and GPUs and to assess the power/area/performance efficiency using production quality RTL for both an IA32 CPU design and a modern multi-core multithreaded GPU design. The proposed architecture enhancements to both the CPU and GPU can enable much more efficient software management of parallel computation across heterogeneous cores. By minimizing resources dedicated solely to 3D-specific graphics processing, significant improvements in area and power efficiency can be achieved.

2.4 Pangaea Architecture

This section introduces Pangaea's architecture enhancements to the IA32 CPU and architectural reorganization of the X4500 GPU to support tighter architectural integration.

2.4.1 CPU-GPU Integration

Pangaea is a novel CPU-GPU integration architecture design that removes the legacy graphics front-end and back-end of the traditional GPU design to enhance general-purpose (non-graphics) computation. With architectural support for shared memory and a fly-weight user-level inter-core communication mechanism, Pangaea provides a tightly-coupled architectural integration of CPU and GPU EUs to more efficiently support collaborative heterogeneous multithreading between GPU threads and CPU threads.

Figure 2.2 shows a high level diagram of the Pangaea architecture. Pangaea physically couples a set of EUs directly with each CPU via an agile thread spawning interface, but

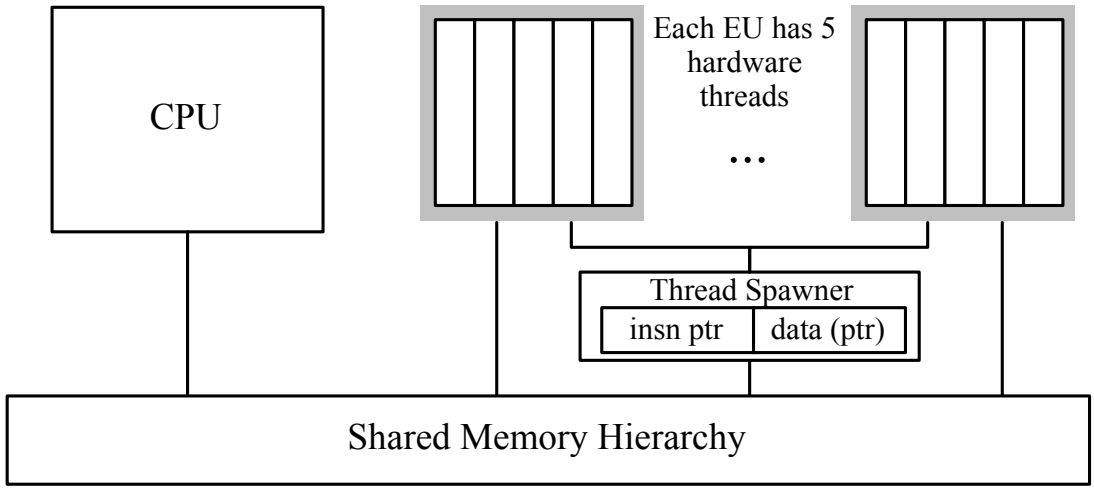

Figure 2.2: Pangaea: Integrated CPU-GPU without Legacy Graphics Front- and Back-End.

without the legacy graphics front-end and back-end. Each EU works as a TLP/DLP co-processor to the CPU. This mechanism allows for a more power and area efficient design, which maximizes the utilization of the massively-parallel ALUs packed in the EUs.

The shared cache supports the collaborative multithreading relationship (peer-to-peer or producer-consumer) between the CPU and EUs. Both CPU and EU cores fetch their instructions and data from the shared memory. The common working sets between CPU threads and EU threads benefit from the shared cache. Enabling a coherent shared address space also make it easier to build a simple communication mechanism between the CPU and EU cores. The communication mechanism between the CPU and EU cores is introduced as an ISA extension.

In Panagea, the EUs appear as additional function units to which threads can be dispatched from the CPU. The CPU is responsible for both assigning and monitoring the GPU's work. The CPU can receive results from the GPU as soon as they are ready and schedule new threads to the GPU as soon as EU cores become idle. Inter-processor interrupts (IPIs) have often been leveraged for cross-core communication, but they introduce performance overheads that are not appropriate in the intended fine grained multi-

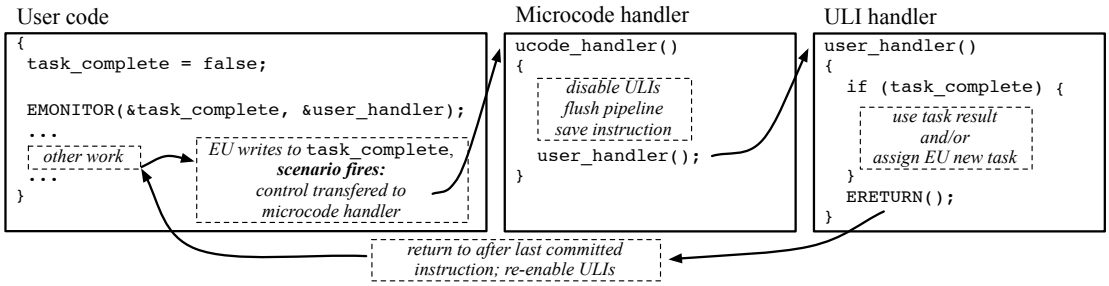

Figure 2.3: Example of User-Level Interrupt (ULI).

threaded environment of Pangaea. Instead of using IPIs, Pangaea leverages simple and fast *user-level interrupts* (ULIs) which are discussed in the next section. A fast mechanism is desirable as the EU threads are short lived and each EU thread processes only a small amount of data. The CPU spawns a large number of threads to increase the resource utilization of the EUs which are optimized for DLP and TLP.

Sections 2.4.2 and 2.4.3 describe the IA32 ISA extension that supports a user-level communication mechanism between the CPU and EUs. Section 2.5 presents an analysis of the power and area efficiency of Pangaea versus the fusion design.

2.4.2 ISA Extension for User-level Interrupts

Pangaea introduces a three-instruction IA32 ISA extension that supports communication between heterogeneous cores. The three instructions are **EMONITOR**, **ERETURN**, and **SIGNAL**. The communication mechanism is as follows.

A *scenario* is a particular machine event that may occur (or *fire*) on any core. Example scenarios include an invalidation of a particular address, an exception on an EU, or termination of a thread on an EU. **EMONITOR** allows application software to register interest in a particular scenario and to specify a user-defined software handler to be invoked (via user-level interrupt (ULI)) when the scenario fires. This scenario-to-handler mapping is stored in a new form of user-level architecture register called a *channel*. Multiple channels allow multiple scenarios to be monitored simultaneously.

When the scenario fires, the microcode handler disables future ULIs, flushes the pipeline, pushes the current interrupted instruction pointer onto the stack, looks up the instruction pointer for the user-defined handler associated with the channel, and redirects program flow to that address. The change in program control flow is similar to what happens when an interrupt is delivered. The key difference is that the ULI is handled completely in user mode with minimal state being saved/restored when the user-level interrupt handler is invoked.

ERETURN is the final instruction of the user-defined handler. It pops the stack and returns the processor to the interrupted instruction while re-enabling ULIs.

Figure 2.3 shows an example of using ULIs. On the left and right is code provided by software. In the middle is the microcode handler. Software activates a channel by executing the EMONITOR instruction, registering its interest in invalidations to the `task_complete` variable and providing the handler that should be called when the invalidation occurs. In this example—one of many possible usage models—the user code spawns a task to the EU and then performs other work. When the EU completes its task, it writes to the variable `task_complete` which is being monitored and the scenario fires. The microcode handler invokes the user-defined interrupt handler. The user's handler can use the result of the EUs immediately and/or assign the EU another task. The user's handler ends with ERETURN. The program then returns to the instruction just after the last committed instruction prior to the interrupt and the user code continues its work. Other usage models might have the EU's task completion affect the user code's behavior upon returning from the interrupt.

To spawn a thread to the EU, the CPU stores the task (including an instruction pointer to the task itself and a data pointer to the possible task input) at an address monitored by the *Thread Spawner*, shown in Figure 2.2. The Thread Spawner is directly associated with the thread dispatcher hardware on the EUs. The CPU then executes the **SIGNAL** instruction—the third ISA extension—to establish the signaling interface between the CPU and EU.

As in related work [10], the SIGNAL instruction is a special store to shared memory

that the CPU uses to spawn EU threads. Using SIGNAL, the EUs can be programmed to monitor and snoop a range of shared addresses similar to SSE3's MONITOR instruction [32]. Upon observing the invalidation caused by the CPU's SIGNAL, the Thread Spawner loads the task information from the cache-line payload. The Thread Spawner then enqueues the EU thread into the hardware FIFO in the EU's thread dispatcher, which binds a ready thread to a hardware thread core (EU), and then monitors the completion of the thread's execution.

Upon recognizing the completion of a thread, the Thread Spawner performs a final store (here, writing to `task_complete`) that results in the scenario firing, as shown in Figure 2.3. The CPU thread can schedule and dispatch more EU threads in response (not shown).

Because the thread spawning and signaling interface between the CPU and EUs leverages simple loads and stores, it can be built as efficiently as regular cache coherence with very low on-chip latencies.

A similar fly-weight signaling mechanism is also used in hardware to implement the exoskeleton *proxy execution* mechanism [7]. In Pangaea the IA32 CPU handles exceptions and faults incurred on the GMA X4500 cores for *address translation remapping* and *collaborative exception handling* using proxy execution. These mechanisms are essential to support a shared virtual address space between the IA32 CPU and the GMA X4500 cores.

Figure 2.4 shows the microarchitecture block diagram of the IA32 core used for this study. The darkened units were modified to support ULIs. First, new registers are introduced to support multiple channels (shown in Figure 2.4(a)). Each channel holds a mapping between a user handler's starting address and the associated ULI scenario. A register is used to hold a blocking bit which specifies if ULIs are temporarily disabled. Since the channel registers store application specific state, these registers need to be saved and restored across OS thread context switches along with any active EU thread context. Existing IA32 XSAVE/XRSTOR instruction support can be modified to save and restore additional state across context switches [33]. These registers can be read and written under

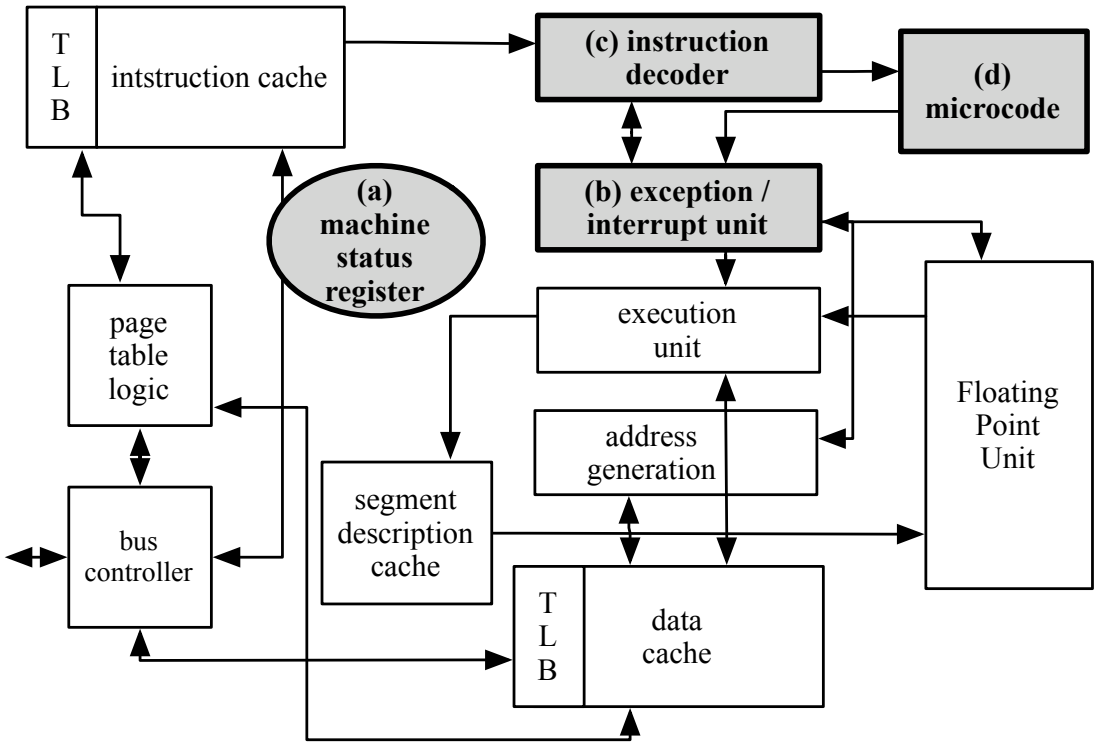

Figure 2.4: IA32 CPU Block Diagram. Shaded blocks indicate modifications to support ULI.

the control of microcode.

The exception/interrupt unit (shown in Figure 2.4(b)) handles all interrupts and faults, and determines whether instructions should be read from the instruction decoder or the microcode. This unit is modified to recognize ULI scenarios. A new class of interrupt request, **ULI-YIELD**, triggers at the firing of a scenario and requests a microcode control-flow transfer to the ULI microcode handler. This interrupt is handled in the integer pipeline. All state logic associated with the ULI-YIELD, determining when an ULI-YIELD should be taken, and saving pending ULI-YIELD events is found here. Because the ULI-YIELD request has the lowest priority of all interrupt events, ULIs do not interfere with traditional interrupt handling. Once the ULI-YIELD has priority, the exception/interrupt unit flushes the pipeline and jumps to the ULI microcode handler. If multiple channels are im-

plemented, when multiple instances of ULI-YIELD interrupts simultaneously occur, lower indexed channels have higher priority over higher indexed channels.

The instruction decoder (shown in Figure 2.4(c)) is responsible for decoding instructions and providing information needed for the rest of the CPU to execute the instruction. The decoder is modified to add entry points for the new IA32 instructions EMONITOR, ERETURN and SIGNAL. These changes map the CPU instructions to the corresponding microcode flows in the microcode. The microcode (shown in Figure 2.4(d)) is modified to contain the ULI microcode handler and the microcode flows for EMONITOR, ERETURN and SIGNAL. The ULI microcode handler flow saves the current instruction pointer by pushing it onto the current stack, sets the blocking bit to prevent taking recursive ULI events, and then transfers control to the user-level ULI handler. The EMONITOR microcode flow registers a scenario and the user handler instruction pointer in the ULI channel register. The ERETURN microcode flow pops the saved instruction pointer off the stack, clears the blocking bit and finally transfers control to the main user-code where it starts re-executing the interrupted instruction.

In Pangaea, we introduce a ULI scenario, **ADDR-INVAL**, which architecturally represents an invalidation event incurred on a range of addresses, which resembles the behavior of a user-level version of the MONITOR/MWAIT instruction in SSE3. Unlike MWAIT [32], when the IA32 CPU in Pangaea snoops a store to the monitored address range, the CPU will activate the ULI microcode handler and transfer program control to the user-level ULI handler. To implement a producer-consumer workload using a traditional polling model, the producer regularly reads a designated semaphore address, checking for a value indicating that the consumer has completed its task. With the ADDR-INVAL ULI, the producer sets up a ULI channel to monitor future asynchronous updates to a semaphore and then proceeds to work on other tasks in parallel while the hardware performs the monitoring. When a consumer writes to the semaphore indicating task completion, this triggers the ADDR-INVAL ULI scenario and the producer is informed of this asynchronously. This ULI scenario is used for the signaling between the IA32 CPU cores, the thread spawner,

and the GMA X4500 EUs by leveraging the existing cache coherence protocol support, which is much more efficient than traditional IPI mechanisms that are sent via the interrupt controller. The address range that needs to be monitored is set up using the SIGNAL instruction which directly communicates with the thread spawner.

2.4.3 User-level Interrupt Handler

Certain precautions need to be taken in designing and writing a user-level interrupt handler as it runs in the context of the monitoring software thread. The monitoring software thread is the thread that executes the EMONITOR instruction and monitors the execution of the EU threads. The monitoring software thread runs on the IA32 CPU concurrently with the EU threads that run on the GPU. The user-level interrupts are delivered in the context of the monitoring thread without operating system intervention and they pre-empt the execution of the monitoring thread. Due to the pre-emptive nature of the user-level interrupt the user-defined interrupt handler should avoid attempting to acquire locks or invoke system calls that acquire locks as the monitoring thread may be executing in the middle of a critical section when it is pre-empted to execute the user-level interrupt handler. If the user-level interrupt handler attempts to acquire the same lock that has already been acquired then a deadlock results. An ideal user-level interrupt handler does not need to be complex or invoke system calls as the user-level interrupt handler is responsible for dispatching a new set of threads to the EU or resolving exception conditions for the EU threads to make forward progress. The user-level interrupt handler usually sets flags that are checked by the monitoring thread when exception conditions have to be resolved. An example of this is shown in Figure 2.3.

The user-level interrupt serves as a notification mechanism of a exception that needs to be resolved for the EU threads to make forward progress or to inform the monitoring thread about the termination of a group of EU threads. The monitoring thread can resolve the exception condition and then resume the EU thread at a later point in time. The interrupt mechanism is optional and the monitoring thread can always use the polling

Parameter	Configuration
IA32 CPU	2-issue, in-order, 4-wide SIMD capabilities, optimistically giving 4x speedup over non-SIMD
CPU-only L1 Caches	8KB 2-cycle access write-back data cache, 8KB Instruction cache, 2-way set associative
EUs	2 EUs, 5 hardware threads each, 8-wide SIMD ISA, 4-wide SIMD execution unit, 0 latency thread switch, 64 256-bit registers per thread. Same clock speed as CPU
EU-only Instruction Cache	4KB shared instruction cache, 4-way set associative
Shared L2 Cache	256KB shared with EU for EU instructions and data, 32-bits/clock bandwidth, configurable access latency by EU (2 to >100 cycles)

Table 2.1: One Pangaea Prototype Configuration that fits one Xilinx Virtex-5.

mechanism to poll on the status of the EU threads by reading the channel registers which contain the scenarios that are being monitored as well as the current status of the scenario. The monitoring thread may attempt to just poll the channel registers when there is no more concurrent work to do or there is a need for a barrier synchronization between the monitoring thread and the EU threads.

The user-level interrupt handler is also responsible for saving and restoring the register state that is not saved/restored by the microcode handler. Since the user-level interrupt handler runs in the context of the monitoring thread it is safe to assume that the code segment or stack segment registers do not change after the monitoring thread executes the EMONITOR instruction as segmentation is not normally used for virtual memory management in modern operating systems. The only exception to this assumption is when the monitoring thread is running in compatibility 32-bit mode under a wrapper on a 64-bit operating system. A change in the code and stack segment occurs during transition from compatibility 32-bit mode to 64-bit mode in user space. The microcode handler is modified to suppress any user-level interrupts to be delivered when the code segment values do not match what was recorded when the EMONITOR instruction is executed. The delivery of the user-level interrupt is frozen for the duration of execution in 64-bit user mode. The EU threads that do not need to report any exceptions or terminate can continue to execute

even when the monitoring thread is executing in 64-bit user mode. When the monitoring thread returns from executing in 64-bit mode back to 32-bit mode the microcode detects the pending user-level interrupt and invokes the user-level interrupt handler. This simple mechanism is sufficient to allow 32-bit applications to continue to work when migrated to run on a 64-bit operating system that runs the application in compatibility mode.

The user-level interrupt mechanism provides a simple, fast and efficient core-to-core communication mechanism without having to introduce new interrupts that need device driver management or major changes to the interrupt controller.

2.5 Pangaea Implementation

To assess its power/area/performance efficiency, we implement a synthesizable design of Pangaea using production quality RTL for both an IA32 CPU design and a modern multi-core multithreaded GPU design. This section describes the Pangaea implementation and prototyping on an FPGA. We also discuss the power/area efficiency analysis. Section 2.6 presents a performance evaluation of Pangaea using a set of non-graphics parallel workloads.

2.5.1 Pangaea's Synthesizable RTL Design

We build a prototype of the proposed Pangaea architecture by implementing synthesizable RTL of a fully functional single-chip heterogeneous CMP consisting of an IA32 CPU and GMA X4500 multi-cores (*i.e.*, EUs). The CPU used in our prototype (shown in Figure 2.4) is a production two-issue in-order IA32 processor equivalent to a Pentium with a 4-wide SSE enhancement. The EU is derived from the RTL for the full GMA X4500 production GPU. We configure our RTL to have two EUs, each supporting five hardware threads. While the baseline design is the physical fusion of the existing CPU and full GPU, in Pangaea much of the front-end and back-end of the GPU have been removed, keeping only the EUs and necessary supporting hardware. By attaching the EU onto the memory hierarchy of the

	LUTs	Registers	Block RAMs	DSP48 blocks
IA32 CPU	50621	24518	118	24
EU Subsystem	84547	36170	67	64
Other	1604	591	91	2

Table 2.2: Virtex-5 FPGA Resource Usage for the Pangaea configuration in Table 2.1.

CPU (sharing of the last-level cache), we no longer need to duplicate the hardware required for accessing and caching memory on the GPU. This prototype design provides means to adjust various configuration parameters, including capacities and access latencies for the memory hierarchy, number of EUs and number of hardware threads per EU. The RTL can be synthesized to either ASIC or FPGA targets.

Table 2.1 shows one particular design that can be synthesized to a Xilinx Virtex-5 XC5VLX330 FPGA using Synplify Pro 9.1 and Xilinx ISE 9.2.03i. Table 2.2 shows the resource usage as reported by Synplify Pro for our FPGA prototype. The IA32 core is larger than one EU, taking up approximately 24% of the 207,360 available FPGA 6-LUTs. As the table shows, the EU subsystem with 2 EUs is less than double the area IA32 CPU in our prototype. The impact from the modifications to the CPU to support ULIs (not shown) is negligible—on the order of 50 LUTs. The logic added to support the thread spawner (not shown) is only 2% of a single EU.

The prototype can fit just two EU cores and occupies 66% of the 6-LUTs available on the Virtex-5 LX330. Larger configurations consisting of multiple EUs have been evaluated in RTL simulation. For parallelizable workloads evaluated in this paper (see Section 2.6), we expect throughput performance to scale roughly with the number of EUs. The critical timing path within the EU allows us to clock the Pangaea prototype system at a maximum of 17 MHz without any special tuning. Similar to [34], the FPGA system on chip is mounted on an adapter that sits in a standard Intel Pentium motherboard with 256MB DRAM. Because of the critical path in our FPGA prototype, we underclocked the motherboard to 17 MHz, down from the original 50 MHz. Note that by underclocking the entire board, the relative speeds between all parts of the system remain unchanged, in-

cluding processor, RAM and cache. The main advantage of an FPGA prototype compared to RTL simulation is the ability to execute orders of magnitude faster. Even at 17 MHz, the FPGA emulation speed is quicker than fast IA32 platform functional simulators such as SoftSDV [35]. This allows our prototype to run off-the-shelf operating system software, including Windows XP and Linux, and execute fat binaries of heterogeneous multithreaded programs produced by frameworks similar to EXOCHI [7].

2.5.2 Area Efficiency Analysis

To assess the area efficiency of Pangaea versus the baseline fusion design, we use the area data collected from the ASIC synthesis of the baseline GMA X4500 RTL code. This ASIC synthesis result corresponds to a processor built on a 65 nm process. The left column of Table 2.3 shows the area distribution of a fusion-styled design with two EUs, including both legacy graphics front- and back-ends. The total area used for graphics-specific legacy hardware (the front- and back-ends) is 81%—the equivalent of over nine EUs. Even if this cost were amortized across more EUs, the overhead remains significant. With 32 EUs, for example, the front- and back-ends still occupy 23% of the chip area.

	2-EU GPU	2-EU Pangaea
Processing	17%	94%
Thread Dispatch	1%	5%
Front-End	34%	–
Memory Interface	1%	–
Back-End	47%	–
Interfacing Logic	–	1%

Table 2.3: Area distribution of two-EU systems.

The right column of Table 2.3 depicts the distribution of chip area of the Pangaea configuration shown in Table 2.1. Unlike the two EU GPU in a fusion design, a two EU Pangaea design has much higher area efficiency. A majority (94%) of the area is used for computation. The extra hardware added to implement the thread spawner and its interface to the interconnection fabric is minimal, amounting to 0.8% of the two-EU system, and easily becomes negligible in a system with more EUs. This significantly reduced overhead allows

us to efficiently use EUs as building blocks for DLP/TLP and couple them with the IA32 cores in a heterogeneous multi-core system.

2.5.3 Power Efficiency Analysis

Table 2.4 shows the total power consumption distribution for a two-EU GPU including both dynamic power and leakage power. Like our area analysis, we use power data based on ASIC synthesis. Most noticeable is that the legacy graphics front-end contributes a lower proportion of power relative to its area. This is mainly due to extensive use of clock-gating that results in reduced dynamic power consumed by the front-end, since only the fixed-functions in the front-end that relate to the current task are switched on. We estimate that removing the legacy graphics-specific hardware would result in the equivalent of five EUs of power savings.

Processing	29%
Thread Dispatch	0.5%
Front-End	14%
Memory Interface	0.5%
Back-End	57%

Table 2.4: Power distribution of a two-EU GPU.

Because of the reduced front-end power, the power overhead for keeping the front-end and back-end in the design is lower than the area overhead. Despite that, the power overhead is still significant for a large number of EUs per GPU, and prohibitive for a small number of EUs. For a two-EU Pangaea (not shown), the power increase due to the thread spawner and related interfacing hardware is negligible compared to the amount of power saved by removing the legacy graphics specific front- and back-ends of the two-EU GPU.

2.5.4 Thread Spawn Latency

Table 2.5 compares the latency of spawning a thread in fusion CPU-GPU integration versus Pangaea. The thread spawn latencies are collected from RTL simulations of the two configurations. The latencies reported are for the hardware only. For the baseline GPGPU

case, thread spawn latency is measured from the time the GPU's command streamer hardware fetches a graphics primitive from the command buffer until the first EU thread performing the desired computation requests is scheduled on an EU core and performs the first instruction fetch. For the Pangaea case, we measure the time from when the IA32 CPU writes the thread spawn command to the address monitored by the thread spawner set up by the SIGNAL instruction, until the thread spawner dispatches the thread to an EU core and the first instruction is fetched. The latency in the GPGPU case is approximate, as the amount of time spent in the 3D pipeline varies somewhat depending on the graphics primitive performed.

GPGPU		**Pangaea**	
3D pipeline	~ 1500	Bus interface	11
Thread Dispatch	15	Thread Dispatch	15
Total	~ 1515	Total	26

Table 2.5: Thread Spawn Latency in cycles.

Unlike the Pangaea case, the measurement for the GPGPU case is optimistic since (1) the latency numbers apply only when the various caches dedicated to the front-end all hit, and (2) the measurement does not take into account of the overhead incurred by the CPU to prepare command primitives. In the GPGPU case, the CPU needs to do a significant amount of work before the GPU hardware can begin processing. For example, when the GPGPU parallel computation is expressed in a shader language, the CPU needs to first convert the device independent shader byte code into native graphics primitives, place the appropriate commands into the command buffer, and notify the GPU that there is new data in the command buffer. Since CPU and GPU operate in separate address spaces, the CPU would also need to go through the device driver interface to copy the code and data into non-cacheable memory the GPU can access. This process is usually inefficient due to the involvement of privilege level ring transitions and data movement between cacheable and non-cacheable memory regions. In effect, the 1515 cycle latency for GPGPU assumes 0-cycles of CPU work. In contrast, the Pangaea case simply involves a user-level 32-bit store containing the instruction pointer of the EU thread to be spawned to the EU core.

Kernel	Description	EU-kernel code size	Data Size	Threads	Icount/ thread
Linear filter 1,2	computes average of pixel and 8 neighbors	2.5 KB	**1:** 640x480 24-bit image	6,480	159
			2: 2000x2000 24-bit image	83,500	159
Sepia Tone 1,2	modifies RGB values of each pixel	4.0 KB	**1:** 640x480 24-bit image	4,800	247
			2: 2000x2000 24-bit image	62,500	247
Film Grain Technology (FGT)	applies artificial film grain filter from H.264 standard	6.6 KB	1024x768 image	96	15,200
Bicubic Scaling	scales YUV image using bicubic filtering	6.1 KB	360x240 → 720 x 480	2,700	691
k-means	k-means clustering of uniformly distributed data	1.5 KB	k=8, 100,000x8	200,000	94
SVM	kernel from SVM-based face classifier	3.6 KB	704x480 image	1,320	11248

Table 2.6: Benchmark Suites

Much of the latency for the GPGPU case comes from needing to map the computation to the 3D graphics processing pipeline. Most of the work performed in the 3D pipeline is not relevant to the computation, but is necessary if the problem were formulated as a 3D computation. By bypassing the front-end of the 3D pipeline, we have successfully reduced the thread spawning latency. With spawning latency reduction of two orders of magnitude, Pangaea can enable more versatile exploration of ILP, DLP and fine grain TLP through tightly-coupled execution on the heterogeneous multi-cores. In Section 2.6, we will study a set of workloads with varying degrees of ILP, DLP and TLP.

2.6 Performance Evaluation

This section evaluates the performance of Pangaea. Our benchmarks are run on the FPGA prototype with the configuration described in Table 2.1, under Linux, compiled using a production IA32 C/C++ compiler that supports heterogeneous OpenMP with the CHI runtime [7]. For the benchmarks, we select four product quality media processing kernels and 2 informatics kernels that are representative of highly parallel compute-intensive workloads rich in ILP, DLP and TLP. These benchmarks have been optimized to run on the IA32 CPU alone (with 4-way SIMD) as the baseline, and on Pangaea to use both the IA32 CPU and the GMA X4500 EUs in parallel whenever applicable, including leveraging the new IA32 ISA extension to support user-level interrupts. Table 2.6 gives a brief description of the benchmarks. While FGT and SVM have relatively few threads of coarser granularity,

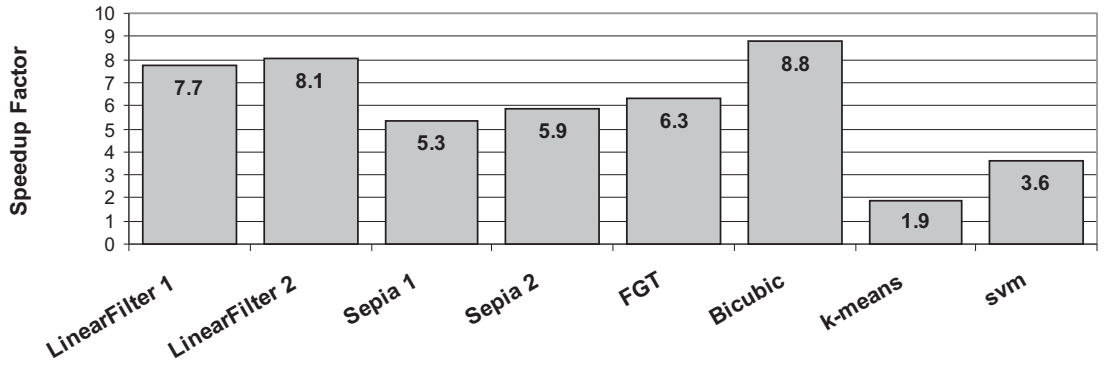

Figure 2.5: Pangaea speedup vs. CPU w/ SSE alone.

the rest have many more threads of fine granularity.

Figure 2.5 shows the speedups of Pangaea relative to a CPU only case. Despite each EU being slightly smaller in area than the CPU, running highly parallel workloads on Pangaea rather than the IA32 CPU alone results in significant performance improvements, ranging from 1.9 to 8.8× improvement on a two-EU Pangaea system.

The first four benchmarks are implementations of several key image and video processing algorithms. They operate on image frames and tend to be highly parallelizable, because an input image can usually be divided into independent macro-blocks (*e.g.*, 8 by 8 pixels in dimension) which can be processed independently. Consequently, many parallel threads can be created, each corresponding to a macro-block. Each thread can be further optimized to exploit 8-wide SIMD operations. Between threads, spatial or temporal locality can also be exploited. For example, in some video processing algorithms, adjacent macro-blocks along x-, or y- or the diagonal dimension may have overlapping stripe or mini-blocks. It is advantageous to schedule the corresponding threads back-to-back so that the overlapped data fetched by the first thread can be reused by the second thread. With architectural support for fly-weight thread spawning and inter-core signaling, Pangaea can efficiently support agile user-level thread scheduling. With these optimizations, the benchmarks show impressive speedups. Linear filter computes the average pixel val-

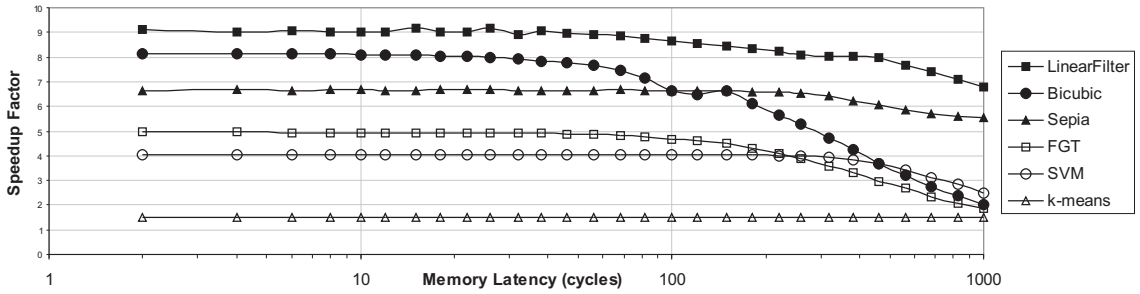

Figure 2.6: Tolerance of Pangaea to Different Memory Access Latencies.

ues of a pixel with its 8 neighbors. Sepia tone modifies each pixel's RGB values, dependent only on the same pixel's original RGB values. FGT applies an artificial film grain filter. Bicubic performs a bicubic-filtered image scaling.

Although similar to Sepia tone, Linear filter sees a larger speedup mainly because Linear filter makes references to neighboring pixels, which the CPU cannot store entirely in architectural registers, requiring cache accesses. When executed on the EU, an entire block of pixels can be stored and manipulated in the EU's large register file. The other two benchmarks are classic machine learning informatics benchmarks that focus on either clustering (k-means) or segregating (SVM) classes of high dimensional data. K-means clustering finds k clusters in a set of points by finding the set of points closest to randomly-generated centroids, then iteratively moving the centroid to be the mean of the set of points that belongs to it. This benchmark is partially parallelized, and cooperatively executes on both the CPU and EU simultaneously. Finding which cluster each point belongs to is parallel and runs on the EU, and computing the mean is performed serially, on the CPU. The serial portion is the bottleneck in this benchmark, resulting in a small 1.9× speedup. The transition between parallel and serial sections of the computations is made more efficient through the fly-weight thread spawning and signaling between the CPU and the EU. The Support Vector Machine (SVM) kernel performs the dot product of blocks of pixels with an array of constant values. Unlike k-means, there is no significant serial portion to the

code, and a speedup of 3.6× is achieved.

While it may seem that achieving almost a 9× speedup with only twice the number of functional units is unrealistic, multiple architectural features combine to allow the EUs to operate much more efficiently than the CPU's SIMD unit and result in larger than expected speedup. As discussed in Section 3, Pangaea utilizes not only DLP but also TLP. When multiple threads exist, multithreading significantly increases utilization of the EU's functional units (*e.g.* 92% on the EUs vs 65% on the CPU in Linear filter). Additional performance improvement is attributable to the EUs' ISA. The EU's SIMD-8 instructions allow a large reduction in the instruction count for these data parallel workloads. Furthermore, the EU's large register file minimizes spilling of registers to memory (57% of CPU instructions in bicubic reference memory, whereas only 7.4% of the EU instructions are loads and stores). Bicubic also heavily uses the multiply accumulate instruction and the low latency accumulator registers (55% of EU instructions), which the CPU does not support, giving this benchmark a particular advantage on the EUs.

To further explore the performance aspects of Pangaea, we assess its sensitivity to the latency of the shared memory hierarchy. Here we vary the latency it takes the EU hardware thread to access shared memory from 2 to 1000 cycles. Figure 2.6 shows the results of this experiment. This experiment sheds light on the impact of not only different access times for the shared L2, but also different shared memory configurations. While a latency of between 50 and 100 cycles might simulate a shared last level cache, latencies exceeding 100 cycles can indicate the performance impact of configuration where CPU and EUs share no caches at all. Although the "performance knee" varies for each benchmark, performance is insensitive to access latency up to approximately 60 cycles for all benchmarks. Once access time exceeds 100-200 cycles, performance improvement slowly diminishes, but even at 1000 cycles, speedups are still anywhere from 1.9× to 5.9×. Bicubic and FGT are the most sensitive to access latency due to the fact that the EU's instruction cache is only 4KB, and each of these kernels is over 6KB in size (see Table 2.6). Consequently, higher memory latency affects not only data accesses, but also the instruction supply. K-means shows the

least sensitivity to memory latency. This is because the serial portion of the algorithm (the part run on the CPU) continues to be the performance bottleneck.

The results of this sensitivity study indicate that a variety of shared cache configurations and access times will produce similar speedups. The performance of the Pangaea architecture does not depend entirely on sharing the closest level cache; the choice of which level of memory hierarchy to share can be traded off with margins for ease or efficiency of implementation without noticeably degrading performance.

2.7 Conclusion and Future Work

In this paper, we present Pangaea, a heterogeneous multi-core design, including its architecture, an implementation in synthesizable RTL and an in-depth evaluation of power, area, performance efficiency and tradeoffs. We demonstrate the potential to significantly improve power/area/performance efficiency for heterogeneous multi-core designs, should they be targeted for a general-purpose heterogeneous multithreading model beyond legacy graphics. As long as Moore's Law continues at its current pace, the level of integration in mainstream microprocessors will continue to increase in terms of quantity and diversity of heterogeneous building blocks, so will the need to achieve higher power/area efficiency. It is advantageous to represent these heterogeneous building blocks as additional architectural resources to the general-purpose CPU. Such tighter architectural integration will allow ease of programming and the use of these new building blocks without requiring drastic changes in the software ecosystem (*e.g.*, the OS). In turn, the software ecosystem will continue to innovate and harvest the parallelism offered by the hardware more efficiently. Even for graphics, leading researchers [36, 37] are actively investigating opportunities beyond today's brute-force, unidirectional rendering pipeline. They have proposed programmable graphics and interactive rendering techniques to design adaptive, demand-driven renderers that can efficiently and easily leverage all processors in heterogeneous parallel systems and tightly couple the distinct capabilities of the ILP-optimized CPU and

DLP/TLP-optimized GPU multi cores to generate far richer and more realistic imagery. Like the famed wheel of reincarnation [38], an efficient heterogeneous multi-core design like Pangaea potentially offers opportunities to significantly accelerate parallel applications like interactive rendering. We continue to actively investigate these opportunities in our on-going exploration.

Acknowledgments

We would like to thank Prasoonkumar Surti, Chris Zou, Lisa Pearce, Xintian Wu, and Ed Grochowski for the productive collaboration throughout the Pangaea project. We also appreciate the support from John Shen, Shekhar Borkar, Joe Schutz, Tom Piazza, Jim Held, Ketan Paranjape, Shiv Kaushik, Bryant Bigbee, Ajay Bhatt, Doug Carmean, Per Hammarlund, and Dion Rodgers. In addition, we would like to thank the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper. Henry Wong and Tor Aamodt are partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] E. Grochowski and M. Annavaram. Energy per Instruction Trends in Intel Microprocessors. *Technology@Intel Magazine*, March 2006.

[2] *GPGPU: General Purpose Computation using Graphics Hardware*. http://www.gpgpu.org.

[3] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *Proc. 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[4] Nvidia. *Compute Unified Device Architecture (CUDA)*. http://developer.nvidia.com/object/cuda.html.

[5] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[6] Peakstream Inc. *The PeakStream Platform: High Productivity Software Development for Multi-core Processors*, 2006.

[7] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, 2007.

[8] Microsoft. *A Roadmap for DirectX*. http://msdn.microsoft.com/en-us/library/bb756949.aspx.

[9] Intel. *G45 Express Chipset*. http://www.intel.com/Assets/PDF/prodbrief/319946.pdf.

[10] Richard A. Hankins, Gautham N. Chinya, Jamison D. Collins, Perry H. Wang, Ryan Rakvic, Hong Wang, and John P. Shen. Multiple Instruction Stream Processor. In *Proc. 33rd International Symposium on Computer Architecture*, 2006.

[11] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. 32nd International Symposium on Computer Architecture*, pages 298–309, 2005.

[12] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. 32nd International Symposium on Computer Architecture*, pages 506–517, Jun. 2005.

[13] Soraya Ghiasi. Aide de Camp: Asymmetric Multi-core Design for Dynamic Thermal Management. Technical Report TR-01-43, 2003.

[14] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of Both Latency and Throughput. In *Proc. IEEE International Conference on Computer Design*, 2004.

[15] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. 36th International Symposium on Microarchitecture*, Dec. 2003.

[16] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. 31st International Symposium on Computer Architecture*, Jun. 2004.

[17] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proc. 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[18] T. Morad, U. Weiser, and A. Kolodny. ACCMP - Asymmetric Cluster Chip-Multiprocessing. Technical Report 488, CCIT, 2004.

[19] Tomer Morad, Uri Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Computer Architecture Letters*, 5(1), 2006.

[20] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM Transactions on Graphics*, volume 23, pages 777–786, 2004.

[22] Anne Bracy, Kshitij Doshi, and Quinn Jacobson. Disintermediated Active Communication. *IEEE Computer Architecture Letters*, 5(2):15, 2006.

[23] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. 17th International Symposium on Computer Architecture*, pages 104 – 114, May 1990.

[24] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a Message-Driven Processor. In *Proc. 14th International Symposium on Computer Architecture*, pages 189 – 196, 1987.

[25] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, 1992.

[26] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proc. 23rd International Symposium on Computer Architecture*, pages 244–255, May 1996.

[27] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, 1994.

[28] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. 23rd International Symposium on Computer Architecture*, pages 179–188, 1996.

[29] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. 23rd International Symposium on Computer Architecture*, 1996.

[30] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, 1994.

[31] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19th International Symposium on Computer Architecture*, pages 430–440, May 1992.

[32] Intel. *Use MONITOR and MWAIT Streaming SIMD Extensions 3 Instructions*. http://softwarecommunity.intel.com/Wiki.

[33] Intel. *IA Programmers Reference Manual 2008*. http://www.intel.com/products/processor/manuals/index.htm.

[34] Shih-Lien L. Lu, Peter Yiannacouras, Rolf Kassa, Michael Konow, and Taeweon Suh. An FPGA-based Pentium in a Complete Desktop System. In *International Symposium on Field-Programmable Gate Arrays*, pages 53–59, 2007.

[35] Richard Uhlig, Roman Fishtein, Oren Gershon, Israel Hirsh, and Hong Wang. SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture. *Intel Technology Journal*, (Q4):14, 1999.

[36] Eric Haines. An Introductory Tour of Interactive Rendering. *IEEE Computer Graphics and Applications*, 26(1), 2006.

[37] Matt Pharr, Aaron Lefohn, Craig Kolb, Paul Lalonde, Tim Foley, and Geoff Berry. Programmable graphics: the future of interactive rendering. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–6, 2008.

[38] T. H. Myer and Ivan E. Sutherland. On the Design of Display Processors. *Communications of ACM*, 11(6):410–414, 1968.

Chapter 3

The Performance Potential for Single Application Heterogeneous Systems[2]

3.1 Introduction

As we blindly march towards thousands [1] of cores, one might be tempted to ask how useful such systems might be for the average application. Achieving linear speedups with more cores, like all good things, must eventually come to an end [2]. Gustafson [3] offers a dissenting view, where applications will scale to utilize all available hardware.

Whereas architects have traditionally been the ones faced with the problem of limited scalability [1, 4], the current trend of increasing cores lifts the performance scalability burden from architects and places it onto algorithm designers, via restrictive programming models [5]. It's not entirely clear how well algorithms can deal with this problem. Amdahl makes the assumption that algorithms have inherently serial portions that will limit performance on highly-parallel systems, whereas Gustafson assumes that algorithms can always be scaled to increase parallelism if problem sizes are allowed to increase.

While there undoubtedly has been successes in achieving speedups using highly-parallel GPU systems [6, 7], most applications do not have large amounts of easily-extracted parallelism.

Like earlier concerns about how much parallelism exists in applications that can be extracted by extra hardware [8, 9, 10, 11] and the hardware area and power costs required to extract it, we should also ask the same question in about many-core systems: How much

[2]Submitted for review to *The 15th International Symposium on High-Performance Computer Architecture*. Henry Wong, Tor M. Aamodt. The Performance Potential for Single Application Heterogeneous Systems.

parallelism exists in applications, and what does it cost to extract it with more hardware? We make an attempt at answering this question in this paper.

We focus our analysis on heterogeneous systems. Heterogeneous systems offer better performance [12, 13] and power efficiency [14] than homogeneous many-core systems.

Typically, heterogeneous systems perform computation using two types of processors. There have been some commercial implementations [5, 15, 16]. These systems use a traditional microprocessor for serial tasks, while offloading parallel sections of algorithms to an array of smaller cores to efficiently exploit the available parallelism. The Cell processor [15] is a heterogeneous multi-core system, where a traditional PowerPC core resides on the same die as an array of 8 smaller cores. GPU compute systems [5, 16] are typically off-chip, attached to the CPU system on a card on a PCI Express bus, although commercial CPU-GPU single-chip systems have been announced [17]. Whether a system is multi-core or off-chip affects the performance of the communications channel (latency and bandwidth) between the large cores and the array of small cores.

One common characteristic of heterogeneous multi-core systems is that the small multi-cores for exploiting parallelism are unable to execute a single thread of execution as fast as the larger sequential processor in the system. The Nvidia G80 series, for example, has a register to register read-after-write latency of 24 shader clock cycles [5]. Our limit study is designed to capture this effect.

There have been previous limit studies on parallelism in the context of single-threaded machines [8, 9, 10], and homogeneous multi-core machines [11].

A heterogeneous system presents a somewhat different trade-off, as it is no longer just a question of how much parallelism can be extracted in software, but also whether the parallelism is worth extracting in the face of slower sequential performance (higher register read-after-write latency) and finite communication channels between processors.

We have avoided focus on applications with high thread-level parallelism, as those have been studied well as examples of what works effectively on existing GPU compute systems.

This limit study makes the following contributions:

- We model an optimistic heterogeneous system consisting of a serial processor and a parallel processor, modeling a traditional CPU and an array of cores for exploiting parallelism, using an algorithm that can automatically extract parallelism across function calls and loop iterations and optimally schedule instructions for the two processors.
- We evaluate the effect of the parallel processor array's sequential performance on the ability of a heterogeneous system to accelerate a set of general-purpose applications. We find sequential performance is a significant limitation on achievable speedup.
- We also evaluate the effects of constraining the communication channel between the two processors (latency and bandwidth). We find that latency and bandwidth have comparatively minor effects on speedup.
- We then look at the area and power efficiency of the heterogeneous system and propose design parameters to optimize those two metrics. We observe that the optimal area-efficient design uses a lower sequential performance parallel processor, while the optimal power-efficient design uses a higher sequential performance parallel processor, relative to our model of current GPUs.

In the case of a heterogeneous system using a GPU-like parallel processor, speedup is limited to only 12.7x for SPECfp 2000, 2.2x for SPECint 2000, and 2.5x for PhysicsBench [18]. When connecting the GPU to the system using an off-chip PCI Express-like bus, SPECfp achieves 74%, SPECint 94%, and PhysicsBench 82% of the speedup achievable without latency and bandwidth limitations.

We review previous limit studies in Section 2, present our model in Section 3, methodology in Section 4, analyze our results in Section 5, and conclude in Section 6.

3.2 Related Work

There have been many limit studies on the amount of parallelism within sequential programs.

Wall [8] studies parallelism in SPEC92 under various limitations in branch prediction, register renaming, and memory disambiguation. Lam et al. [9] studies parallelism under branch prediction, condition dependence analysis, and multiple-fetch. Postiff et al. [10] perform a similar analysis on the SPEC95 suite of benchmarks. These studies showed that significant amounts of parallelism exist in typical applications under optimistic assumptions. These studies focused on extracting instruction-level parallelism on a single processor. As it becomes increasingly difficult to extract ILP out of a single processor, performance increases often comes from multi-core systems.

As we move towards multi-core systems, there are new constraints, such as communication latency, that are now applicable. Vachharajani et al. [11] studies speedup available on homogeneous multiprocessor systems. They use a greedy scheduling algorithm to assign instructions to cores. They also scale communication latency between cores in the array of cores and find that it is a significant limit on available parallelism.

In our study, we extend these analyses to heterogeneous systems, where there are two types of processors. Vachharajani examined the impact of communication between processors within a homogeneous processor array. We examine the impact of communication between a sequential processor and an array of cores. In our model, we roughly account for communication latency between cores within an array of cores by using higher instruction read-after-write latency.

Heterogeneous systems are interesting because they are commercially available [5, 15, 16] and, for GPU compute systems, can leverage the existing software ecosystem by using the traditional CPU as its sequential processor. They have also been shown to be more area and power efficient [12, 13, 14] than homogeneous multi-core systems.

Hill [12] uses Amdahl's Law to show that there are limits to parallel speedup, and

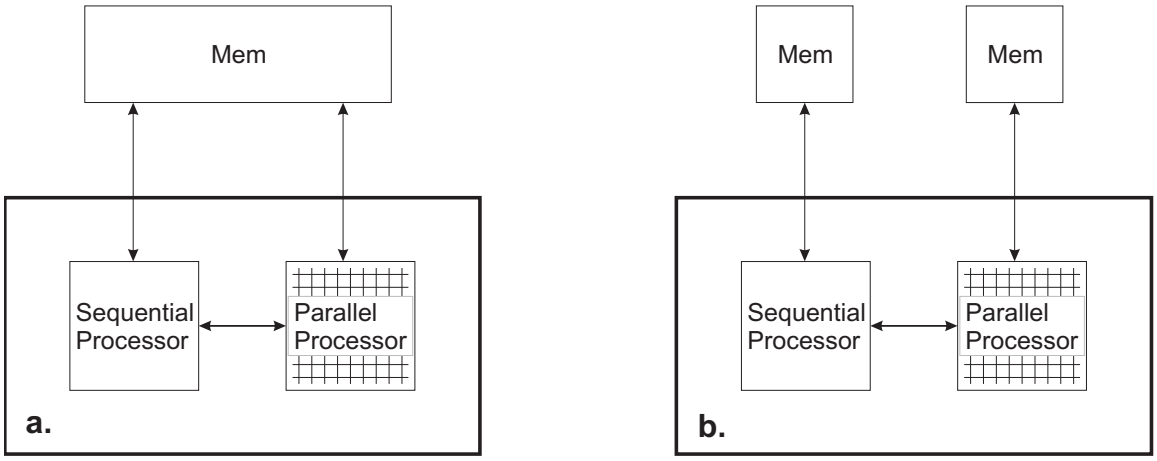

Figure 3.1: Conceptual Model of a Heterogeneous System. Two processors with different characteristics may (a) or may not (b) share memory, affecting whether data needs to be copied over the communication channel connecting them.

makes the case that when one must trade per-core performance for more cores, heterogeneous multiprocessor systems perform better than homogeneous ones because non-parallelizable fragments of code do not benefit from more cores, but do suffer when all cores are made slower to accommodate more cores. Arguments based on Amdahl's Law ignore communication latencies and assume that programs have one fraction that is infinitely parallel, while the remaining fraction is serial. We refine these simplistic assumptions with analysis of real workloads and their behavior scheduled on an idealized machine that also models communication latency and bandwidth limits.

3.3 Modeling a Heterogeneous System

We model heterogeneous systems as having two processors with different characteristics (Figure 3.1). The sequential processor models a traditional sequential processor, while the parallel processor models an array of cores for exploiting parallelism. The parallel processor models an array of low-cost cores by allowing parallelism, but with a longer register read-after-write latency than the sequential processor. The two processors may communicate over a communication channel whose latency and bandwidth we can limit.

We assume that the processors are attached to ideal memory systems. We model a system with private memory (Figure 3.1(b)) by limiting the communication channel's bandwidth when data needs to be copied between processors, while a system with shared (Figure 3.1(a)) memory only needs to consider communication latency, as no data needs to be copied across the link between the two processors.

We describe in Section 3.3.3 our algorithm for partitioning and scheduling an instruction trace to optimize its runtime on the serial and parallel processors.

The following sections describe each portion of our model in more detail.

3.3.1 Serial Processor

We model a serial processor as being able to execute one instruction per cycle. A simple model has the advantage of having predictable performance characteristics that make the optimal scheduling (Section 3.3.3) of work between serial and parallel processors feasible. A simple model preserves the essential characteristic of high-ILP processors that a program is executed serially, while avoiding the modeling complexity of a more detailed model. Although this simple model does not capture the CPI effects of a serial processor which exploits ILP, we are mainly interested in the relative speeds between the sequential and parallel processors. We account for sequential processor performance due to ILP by making the parallel processor relatively slower. In the remainder of this paper, all time periods are expressed in terms of the sequential processor's cycle.

3.3.2 Parallel Processor

We model the parallel processor as a dataflow processor, where a data dependency takes multiple cycles to resolve. Using a dataflow model, we avoid the requirement of partitioning instructions into threads, as done in the thread-programming model. This allows us to model the upper bound of parallelism for future programming models that may be more flexible than threads.

The parallel processor can execute multiple instructions in parallel, provided data de-

pendencies are satisfied. Slower sequential performance of the parallel processor is modeled by increasing the latency from the beginning of an instruction's execution until the time its result is available for a dependent instruction.

We do not limit the parallelism that can be used by the program, as we are interested in the amount of parallelism available in algorithms.

Our model can represent a variety of methods of building parallel hardware. In addition to an array of single-threaded cores, it can also model cores using fine-grain multithreading, like current GPUs.

In GPUs, fine-grain multithreading creates the illusion of a large amount of parallelism (>10,000s of threads) with low per-thread performance, although physically there is a lower amount of parallelism (100s of operations per cycle), high utilization of the ALUs, and frequent thread switching. GPUs use the large number of threads to "hide" register read-after-write latencies and memory access latencies by switching to a ready thread. From the perspective of the algorithm, a GPU appears as a highly-parallel, low-sequential-performance parallel processor.

To model current GPUs, we use a register read-after-write latency of 100 cycles. For example, current Nvidia GPUs have a read-after-write latency of 24 shader clocks [5] and a shader clock frequency of 1.3-1.5 GHz [19, 20]. The 100 cycle estimates includes the effect of instruction latency, the difference between the shader clock and current CPU clock speeds (about 2x), and the ability of current CPUs to extract ILP (about 2x).

3.3.3 Heterogeneity

We model a heterogeneous system by allowing an algorithm to choose between executing on the serial processor or parallel processor and to switch between them (which we refer to as a "mode switch"). We do not allow concurrent execution of both processors. This is a common paradigm, where a parallel section of work is spawned off to a co-processor while the main processor waits for the results. The runtime difference for optimal concurrent processing is no better than 2x compared to not allowing concurrency.

We schedule an instruction trace for alternating execution on the two processors. Execution of a trace on each type of core was described in Sections 3.3.1 and 3.3.2. For each mode switch, we impose a "mode switch cost", intuitively modeling synchronization time during which no useful work is performed. We use a dynamic programming algorithm to choose points along the instruction trace where mode switches should occur such that the total runtime of the trace, including the penalties incurred for switching modes, is minimized. The mode switch cost is used to model communication latency, as described in the next section.

The naïve optimal algorithm runs in quadratic time with respect to the instruction trace length. Each instruction along the trace may be a mode switch point, and for each potential mode switch point all future instructions along the trace are examined to find the cumulative cost of the next mode switch point.

For traces of millions of instructions in length, quadratic time is too slow. We make an approximation to reduce the algorithm to run in time linear in the length of the instruction trace. Instead of looking ahead at all future instructions for each potential mode switch point, we only look ahead 30,000 instructions. The modified algorithm is no longer optimal. We mitigate this sub-optimality by first reordering instructions before scheduling, and observed that the amount of sub-optimality is insignificant.

To overcome the limitation of looking ahead only 30,000 instructions in our algorithm, we reorder instructions in dataflow order before scheduling. Dataflow order is the order in which instructions would execute if scheduled with our optimal scheduling algorithm. This linear-time preprocessing step exposes parallelism found anywhere in the instruction trace by grouping together instructions that can execute in parallel.

We remove instructions from the trace that do not depend on the result of any other instruction. Most of these instructions are dead code created by our method of exposing loop- and function-level parallelism, described in Section 3.4.2. Since dead code can execute in parallel, we remove these instructions to avoid having them inflate the amount of parallelism we observe. Across our benchmark set, 27% of instructions are removed by

this mechanism.

3.3.4 Latency

We model the latency of migrating tasks between processors by imposing a constant runtime cost for each mode switch. This cost is intended to model the latency of spawning a task, as well as transferring of data between the processors. If the amount of data transferred is large relative to the bandwidth of the link between processors, this is not a good model for the cost of a mode switch. This model is reasonable when the mode switch is dominated by latency, for example in a heterogeneous multi-core system where the memory hierarchy is shared (Figure 3.1(a)), so very little data needs to be copied between the processors.

As described in Section 3.3.3, our trace scheduling algorithm considers the cost of a mode switch when scheduling the instruction trace. A mode switch cost of zero would allow freely switching between modes, while a very high cost would constrain the scheduler to choose to run the entire trace on one processor or the other, whichever was faster.

3.3.5 Bandwidth

Bandwidth is a constraint that limits the rate that data can be transferred between processors in our model. Note that this does not apply to the processors' link to its memory (Figure 3.1), which we assume to be unconstrained. In our shared-memory model (Figure 3.1(a)) mode switches do not need to copy large amounts of data so only latency (Section 3.3.4) is a relevant constraint. In our private-memory model (Figure 3.1(b)), bandwidth is consumed as a result of a mode switch.

If a data value is produced by an instruction in one processor and consumed by one or more instructions in the other processor, then that data value needs to be communicated to the other processor. A consequence of exceeding the imposed bandwidth limitation is the addition of idle computation cycles while an instruction waits for its required operand to be transferred. In our model, we assume opportunistic use of bandwidth, allowing

communication of a value as soon as it is ready, in parallel with computation.

Each data value to be transferred is sent sequentially and occupies the communication channel for a specific amount of time. Data values can be sent any time after the instruction producing the value executes, but must arrive before the first instruction that consumes the value is executed. Data transfers are scheduled onto the communication channel using an "earliest deadline first" algorithm, which produces a scheduling with a minimum of added idle cycles.

Bandwidth constraints are applied by changing the amount of time each data value occupies on the communication channel. Communication latency is applied by setting the deadline for a value some number of cycles before the value is consumed.

Computing the bandwidth requirements and idle cycles needed, and thus the cost to switch modes, requires a scheduling of the instruction trace, but the optimal instruction trace scheduling is affected by the cost of switching modes. We approximate the ideal behavior by iteratively performing scheduling and bandwidth computation, using bandwidth-caused idle cycles as an input into the scheduling algorithm, until convergence.

3.4 Simulation Infrastructure

We evaluate performance using micro-op traces extracted from execution of a set of x86-64 benchmarks on the PTLsim [21] simulator. Each micro-op trace was then scheduled using our scheduling algorithm for execution on the heterogeneous system.

3.4.1 Benchmark Set

We chose our benchmarks with a focus towards general-purpose computing. We used the reference workloads for SPECint and SPECfp 2000 v1.3.1 (23 benchmarks, except 253.perlbmk and 255.vortex which did not run in our simulation environment), PhysicsBench 2.0 [18] (8 benchmarks), SimpleScalar 3.0 [22] (used here as a benchmark), and four

Benchmark	*Description*
linear	Compute average of 9 input pixels for each output pixel. Each pixel is independent.
sepia	3x3 constant matrix multiply on each pixel's 3 components. Each pixel is independent.
serial	A long chain of dependent instructions, has parallelism approximately 1 (no parallelism).
twophase	Loops through two alternating phases, one with no parallelism, one with high parallelism. Needs to switch between processor types for high speedup.

Table 3.1: Our microbenchmark set. We also employ many real benchmarks. (See Section 3.4.1)

small microbenchmarks (described in Table 1).

We chose PhysicsBench because it contains both sequential and parallel phases in the benchmark, and would be a likely candidate to benefit from heterogeneity, as it would be unsatisfactory if both types of phases were constrained to one processor type [18].

Our SimpleScalar benchmark used the out-of-order processor simulator from SimpleScalar/PISA, running go from SPECint 95, compiled for PISA.

We used four microbenchmarks to observe behavior at extremes of parallelism, as shown in Table 1. Linear and sepia are highly parallel, serial is serial, and twophase has alternating highly parallel and serial phases.

Figure 3.2 shows the average parallelism present in our benchmark set. As expected, SPECfp has more parallelism (611) than SPECint (116) and PhysicsBench (83). Linear (4790) and sepia (6815) have the highest parallelism, while serial has essentially no parallelism.

3.4.2 Traces

Micro-op traces were collected from PTLsim running x86-64 benchmarks, compiled with gcc 4.1.2 -O2. Four microbenchmarks were run in their entirety, while the 32 real benchmarks were run through SimPoint [23] to choose representative sub-traces to analyze. Our traces are captured at the micro-op level, so in this paper instruction and micro-op are

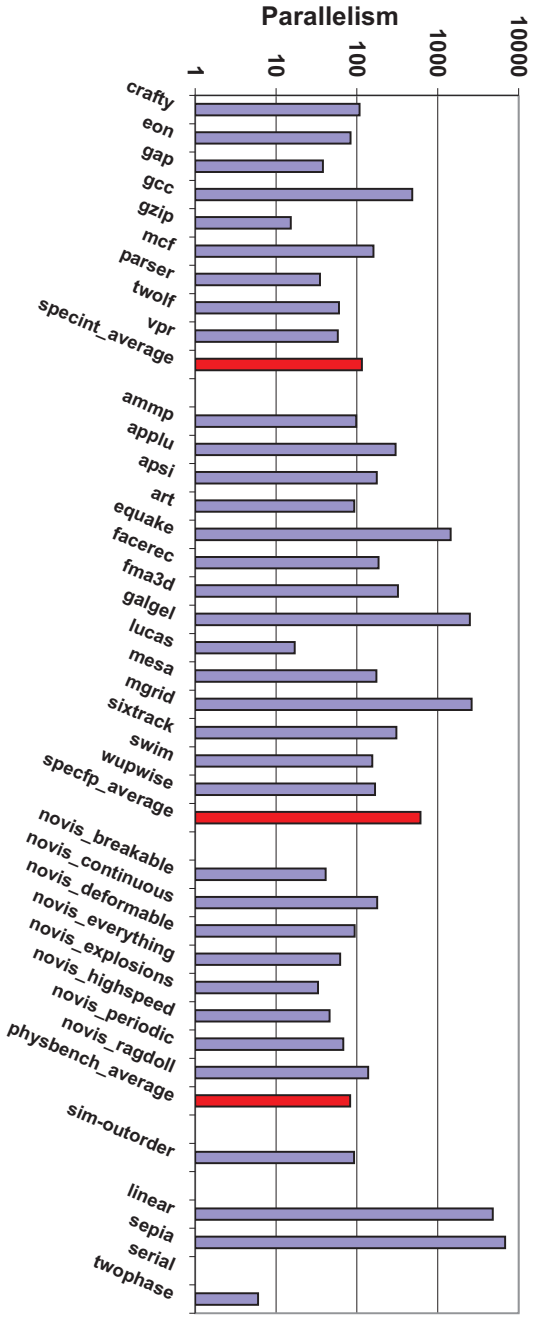

Figure 3.2: Average Parallelism of Our Benchmark Set

used interchangeably.

We used SimPoint to select simulation points of 10-million micro-ops in length from complete runs of benchmarks. As recommended [23], we allowed SimPoint to decide how many simulation points should be used to approximate the entire benchmark run. We averaged 12.9 simulation points per benchmark. This is a significant savings over the complete benchmarks which were typically several hundred billion instructions long. The weighted average of the results over each set of SimPoint traces are presented for each benchmark.

We assume branches are correctly predicted. Many branches, like loops, can often be easily predicted or speculated or even restructured away during manual parallelization. As we are trying to evaluate the upper-bound of parallelism in an algorithm, we avoid limiting parallelism by not imposing the branch-handling characteristics of serial machines. This is somewhat optimistic as true data-dependent branches would at least need be converted into speculation or predicated instructions.

Each trace is analyzed for true data dependencies. Register dependencies are recognized if an instruction consumes a value produced by an earlier instruction (read-after-write). Dependencies on values carried by the instruction pointer register are ignored, to avoid dependencies due to instruction-pointer-relative data addressing. Like earlier limit studies [9, 10], stack pointer register manipulations are ignored, to extract parallelism across function calls. Memory disambiguation is perfect: Dependencies are carried through memory only if an instruction loads a value from memory actually written by an earlier instruction.

It is also important to be able to extract loop-level parallelism and avoid serialization of loops through the loop induction variable. We implemented a generic solution to prevent this type of serialization. We identify instructions that produce result values that are statically known, which are instructions that have no input operands (e.g. load constant). We then repeatedly look for instructions dependent only on values that are statically known and mark the values they produce as statically known as well. We then remove dependen-

cies on all statically-known values. This is similar to repeatedly applying constant folding and constant propagation optimizations [24] to the instruction trace. The dead code that results is removed as described in Section 3.3.3.

A loop induction variable [24] is often initialized with a constant (e.g. 0). Incrementing the induction variable by a constant depends only on the initialization value of the induction variable, so the incremented value is also statically known. Each subsequent increment is likewise statically known. This removes serialization caused by the loop control variable, but preserves genuine data dependencies between loop iterations, including loop induction variable updates that depend on a variable computed value.

3.5 Results

In this section, we present analysis of our results. First, we look at the speedup that can be achieved when adding a parallel co-processor to a sequential machine and show that the speedup is highly dependent on the sequential performance of the parallel processor. We then look at the effect of communication latency and bandwidth, and see that the effect is significant, but small. We then derive area and power efficiency metrics and find parallel processor designs that maximize those metrics.

3.5.1 Why Heterogeneous?

Figures 3.3 and 3.4 give some intuition for the characteristics of the scheduling algorithm. Figure 3.4 shows the parallelism of the instructions that are scheduled to use the parallel processor when our workloads are scheduled for best performance. Figure 3.3(a) shows the proportion of instructions that are assigned to execute on the parallel processor. As the instruction latency increases, sections of the workload where the benefit of parallelism does not outweigh the cost of slower sequential performance become scheduled onto the sequential processor, raising the average parallelism of those portions that remain on the parallel processor, while reducing the proportion of instructions that are scheduled on the

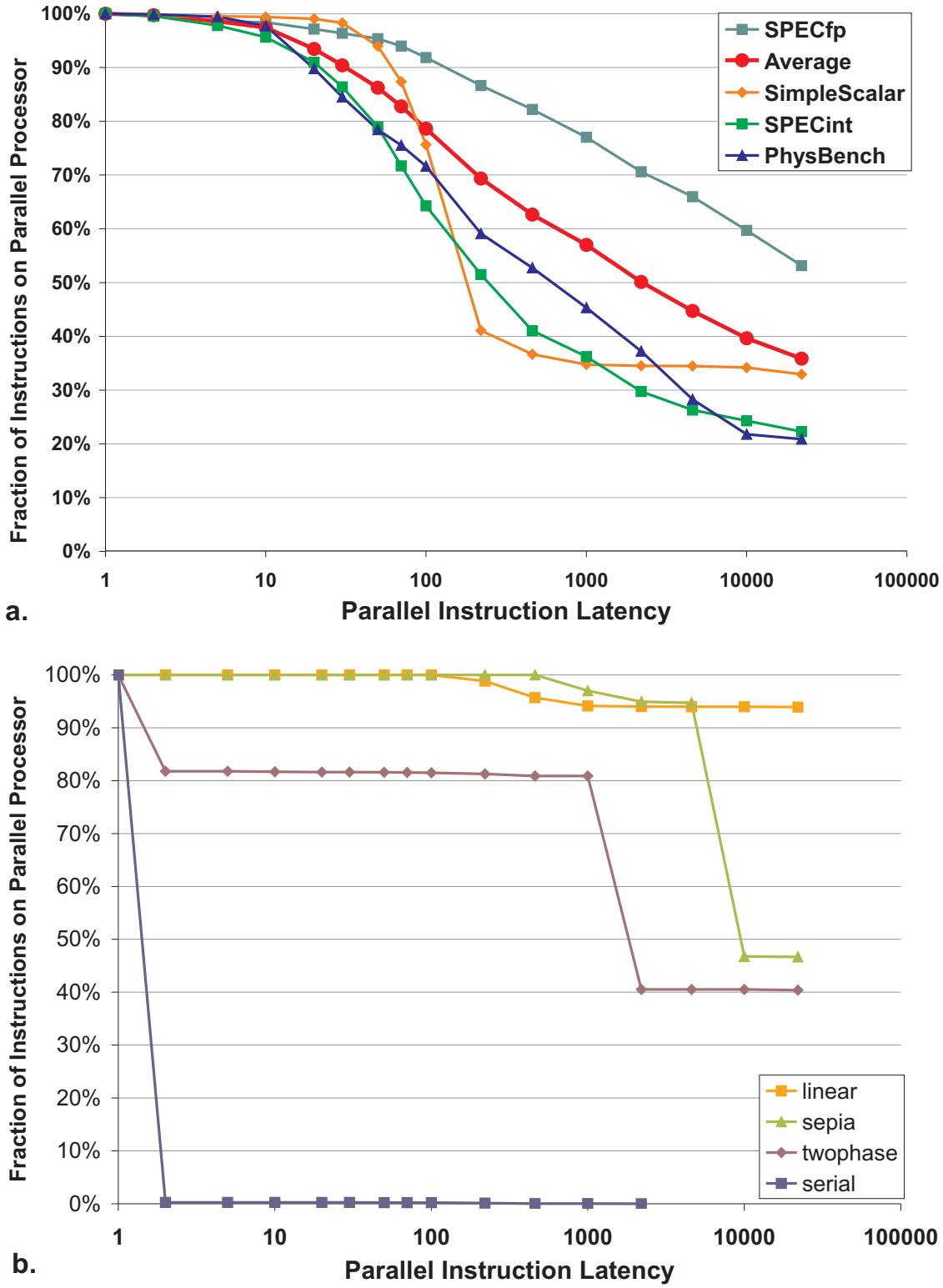

Figure 3.3: Proportion of Instructions Scheduled on Parallel Core. Real benchmarks (a), Microbenchmarks(b)

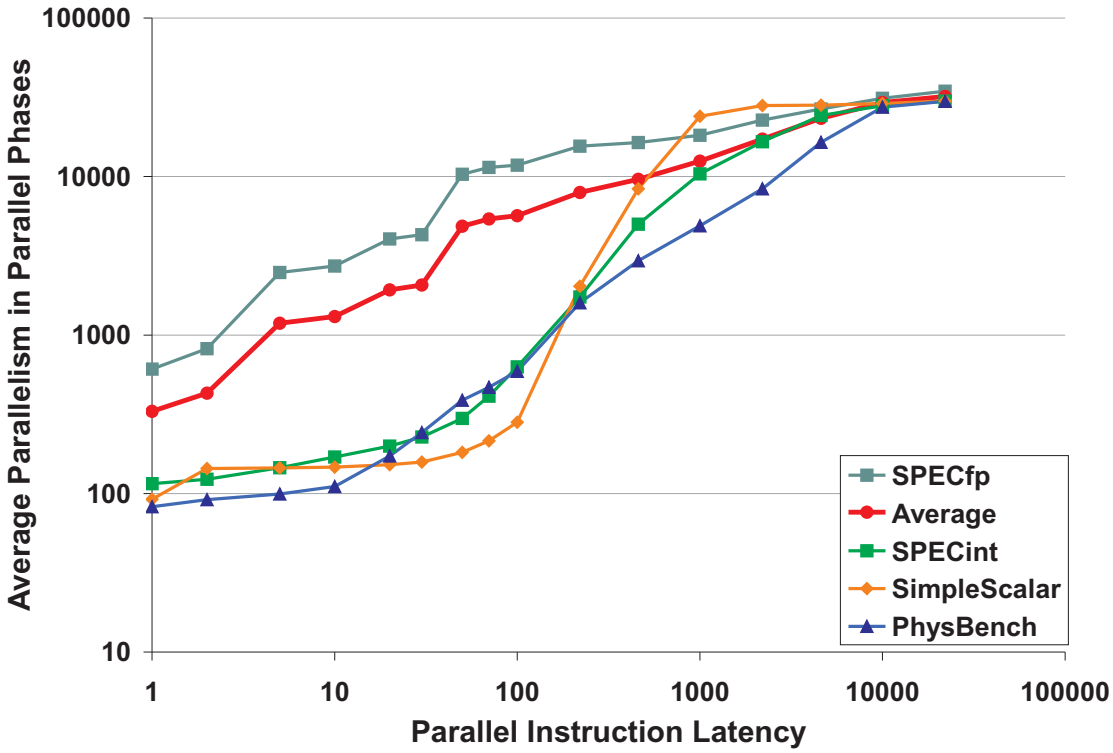

Figure 3.4: Average Parallelism on Parallel Processor

parallel processor. The instructions that are scheduled to run on the sequential processor receive no speedup, but scheduling more instructions on the parallel processor in an attempt to increase parallelism will only decrease speedup.

The microbenchmarks in Figure 3.3(b) show our scheduling algorithm works as expected. Serial has nearly no instructions scheduled for the parallel core. Twophase has about 18.5% of instructions in its serial component that are scheduled on the sequential processor leaving 81.5% on the parallel processor, while sepia and linear highly prefer the parallel processor.

We look at the potential speedup of adding a parallel processor to an existing sequential machine. Figures 3.5(a) and (b) show the speedup of our benchmarks for varying parallel instruction latency, as a speedup over a single sequential processor. Two plots for each benchmark group are shown: The solid plots show the speedup of a heterogeneous system where communication has no cost, while the dashed plot shows speedup when communication is very expensive. We focus on the solid plots in this section.

It can be observed from Figures 3.5(a) and (b) that as the instruction latency increases, there is a significant loss in the potential speedup provided by the extra parallel processor, becoming limited by the amount of parallelism available in the workload that can be extracted, as seen in Figure 3.3. Since our parallel processor model is somewhat optimistic, the speedups shown here should be regarded as an upper bound of what can be achieved.

With a parallel processor with GPU-like instruction latency of 100 cycles, SPECint would be limited to a speedup of 2.2x, SPECfp to 12.7x, PhysicsBench to 2.5x, with 64%, 92%, and 72% of instructions scheduled on the parallel processor, respectively. The speedup is much lower than the peak relative throughput of a GPU compared to a sequential CPU ($\approx$50x), which shows that if a GPU-like processor were used as the parallel processor in a heterogeneous system, the speedup on these workloads would be limited by the parallelism available in the workload, while still leaving much of the GPU hardware idle.

In contrast, for highly-parallel workloads, the speedups achieved at an instruction latency of 100 are similar to the peak throughput available in a GPU. The highly-parallel

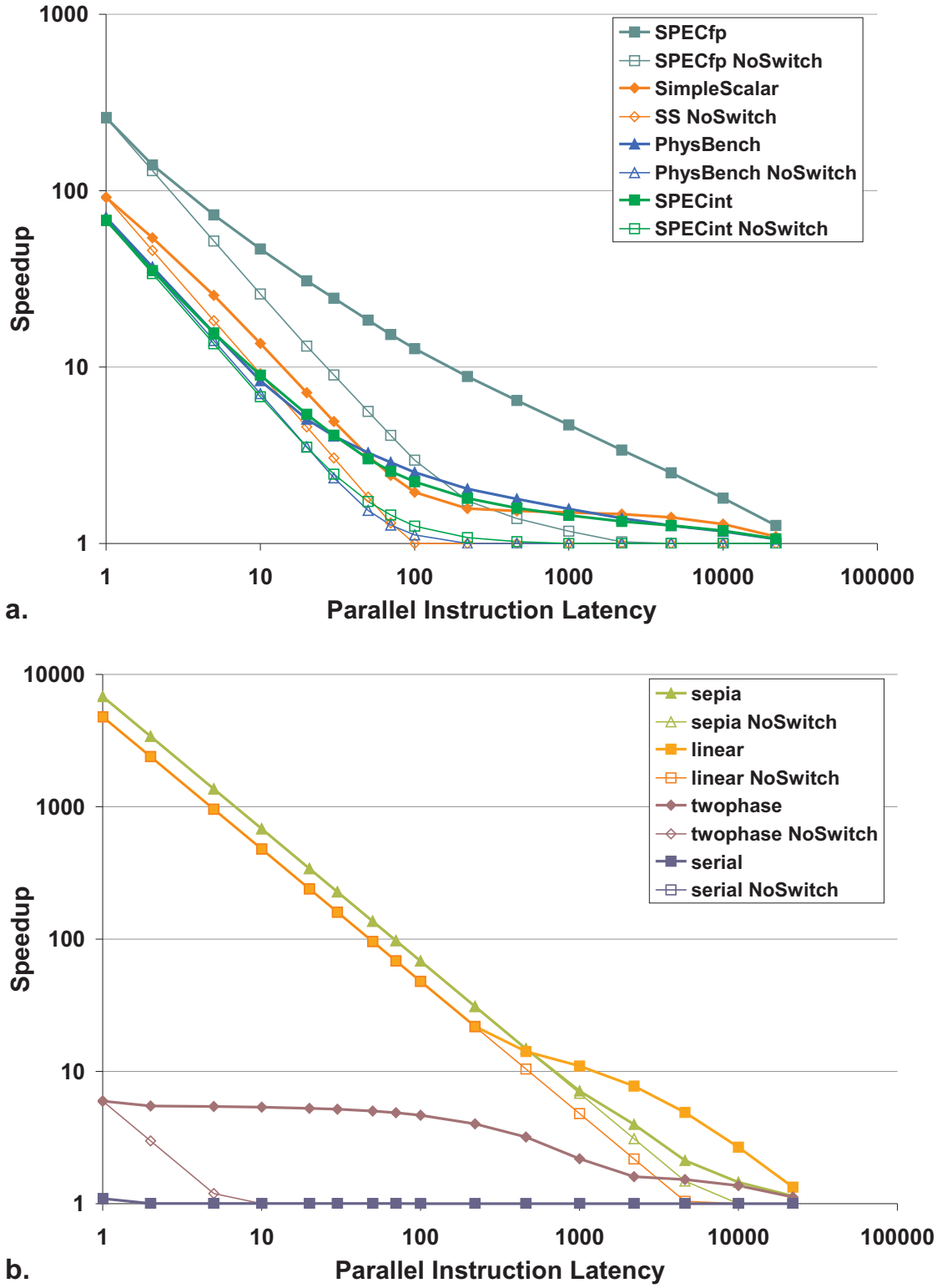

Figure 3.5: Speedup of Heterogeneous System. Traces show speedup for ideal communication (solid) and with communication forbidden (dashed, NoSwitch). Real benchmarks (a), Microbenchmarks (b).

linear filter and sepia tone filter (Figure 3.5(b)) kernels have enough parallelism to achieve 50-70x speedup at an instruction latency of 100. A highly-serial workload (serial) does not benefit from the parallel processor.

Although current GPU compute solutions built with efficient low-complexity multithreaded cores are sufficient to accelerate algorithms with large amounts of thread-level parallelism, general-purpose algorithms would be unable to utilize the large number of thread contexts provided by the GPU, while under-utilizing the arithmetic hardware available. In order to be useful for applications without copious amounts of parallelism, we believe that instruction latencies of GPUs will need to decrease and can no longer rely mostly on fine-grain multithreading to keep utilization high.

3.5.2 Communication

In this section, we evaluate the impact of communication latency and bandwidth on the potential speedup, comparing performance between the extreme cases where communication is unrestricted and communication is forbidden. The solid plots in Figure 3.5 show speedup when there are no limitations on communication, while the dashed plots (marked NoSwitch) has communication so expensive that the scheduler chooses to run the workload entirely on the sequential processor or parallel processor, never switching between them. Figures 3.6(a) and (b) show the ratio between the solid and dashed plots in Figures 3.5(a) and (b), respectively, to highlight the impact of communication. At both extremes of instruction latency, where the workload is mostly sequential or mostly parallel, communication has little impact. It is in the moderate range around 100-200 where communication potentially matters most.

The potential impact of expensive (latency and bandwidth) communication is significant. For example, at a GPU-like instruction latency of 100, SPECint achieves only 56%, SPECfp 23%, and PhysicsBench 44% of the performance of no communication, as can be seen in Figure 3.6(a). From our microbenchmark set (Figures 3.5(b) and 3.6(b)), twophase is particularly sensitive to communication costs, and gets no speedup for instruction la-

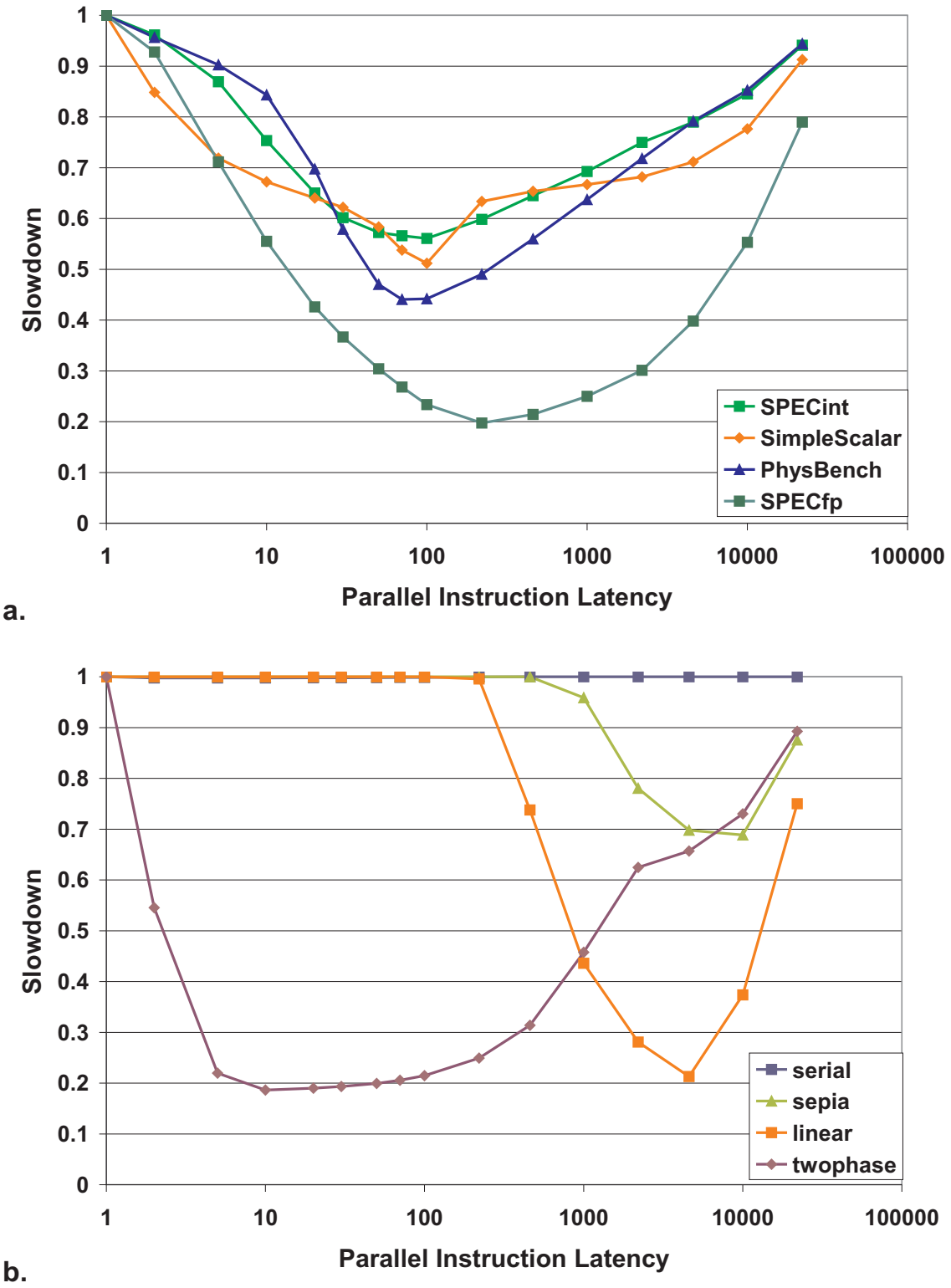

Figure 3.6: Slowdown of infinite communication cost (NoSwitch) compared to zero communication cost. Real benchmarks (a), Microbenchmarks (b).

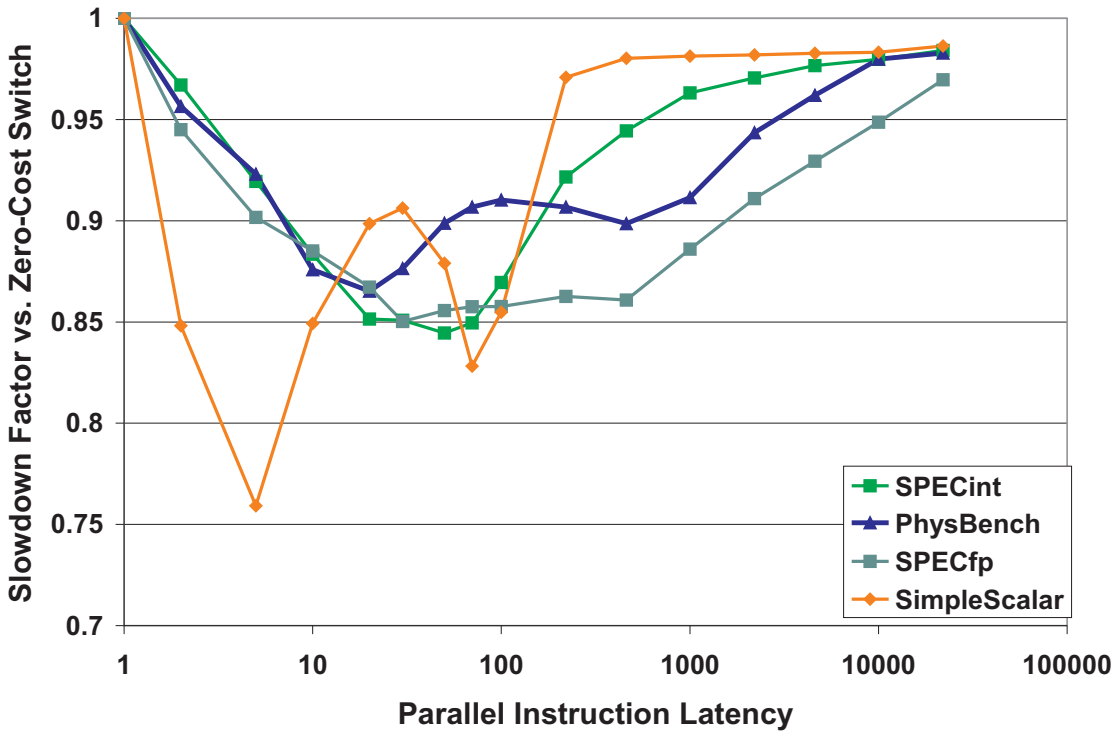

Figure 3.7: Slowdown due to 100,000 cycles of mode-switch latency. Real benchmarks.

tency above 10. We look at more realistic constraints on latency and bandwidth in the following sections.

3.5.3 Latency

Poor parallel performance is often attributed to high communication latency [11]. Heterogeneous processing adds a new communication requirement — the communication channel between sequential and parallel processors (Figure 3.1). In this section, we measure the impact of the latency of this communication channel.

We model this latency by requiring that switching modes between the two processor types causes a fixed amount of idle computation time. In this section, we do not consider the bandwidth of the data that needs to be transferred. This model represents a heterogeneous system with shared memory (Figure 3.1(a)), where migrating a task does not involve

data copying, but only involves a pipeline flush, notification to the other processor of work, and potentially flushing private caches if caches are not coherent.

Figure 3.7 shows the slowdown when we include 100,000 cycles of mode-switch latency in our performance model and scheduling, when compared to zero-latency mode switch.

The impact of imposing a delay for every mode switch has only a minor effect on runtime. Although Figure 3.6(a) suggested that the potential for performance loss due to latency is great, even when each mode switch costs 100,000 cycles (greater than 10us at current clock rates), most of the speedup remains. We can achieve ≈85% of the performance of a heterogeneous system with zero-cost communication.

For systems with private memory (e.g. discrete GPU), data copying is required when migrating a task between processors at mode switches. We consider bandwidth constraints in the next section.

3.5.4 Bandwidth

In the previous section, we saw that high communication latency had only a minor effect on achievable performance. Here, we place a bandwidth constraint on the communication between processors. Data that needs to be communicated between processors is restricted to a maximum rate, and the processors are forced to wait if data is not available in time for an instruction to use it, as described in Section 3.3.5. We also include 1,000 cycles of latency as part of the model.

We first construct a model to represent PCI Express, as discrete GPUs are often attached to the system this way. PCI Express x16 has a peak bandwidth of 4GB/s and latency around 250ns [25]. Assuming current processors perform about 4 billion instructions per second on 32-bit data values, we can model PCI Express using a latency of about 1,000 cycles and bandwidth of 4 cycles per 32-bit value. Being somewhat pessimistic to account for overheads, we use a bandwidth of 8 cycles per 32-bit value (about 2GB/s).

Figure 3.8 shows the performance impact of restricting bandwidth to one 32-bit value

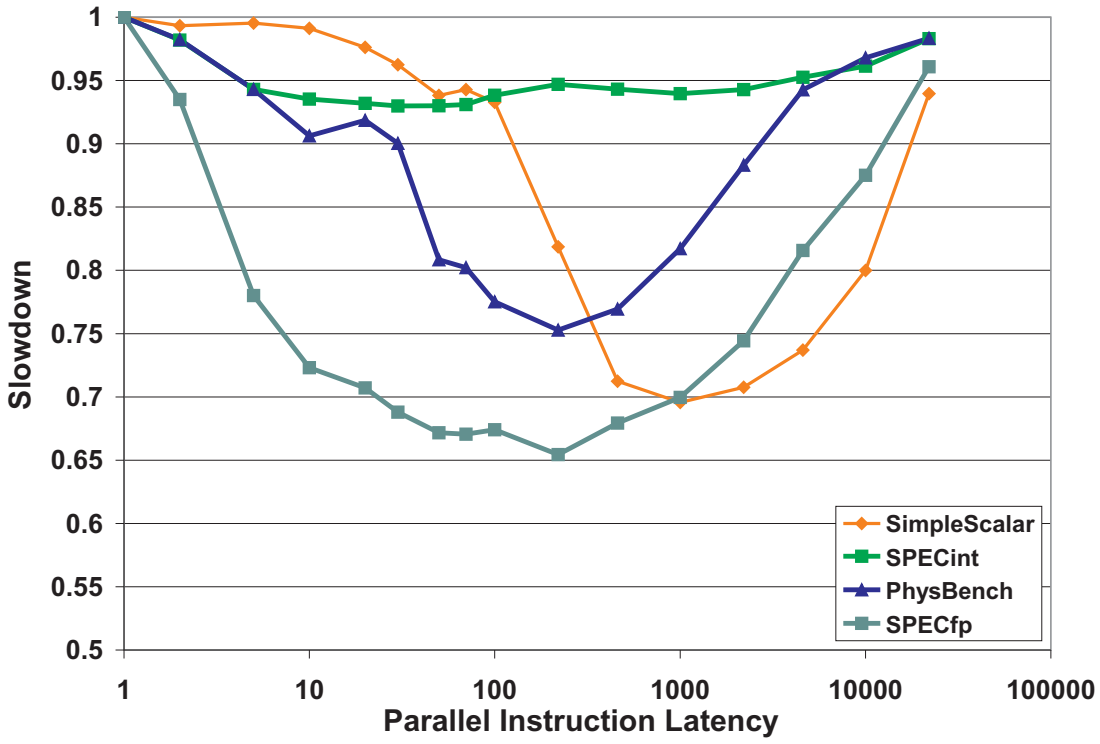

Figure 3.8: Slowdown due to a bandwidth constraint of 8 cycles per 32-bit value and 1,000 cycles latency, similar to PCI Express x16. Real benchmarks.

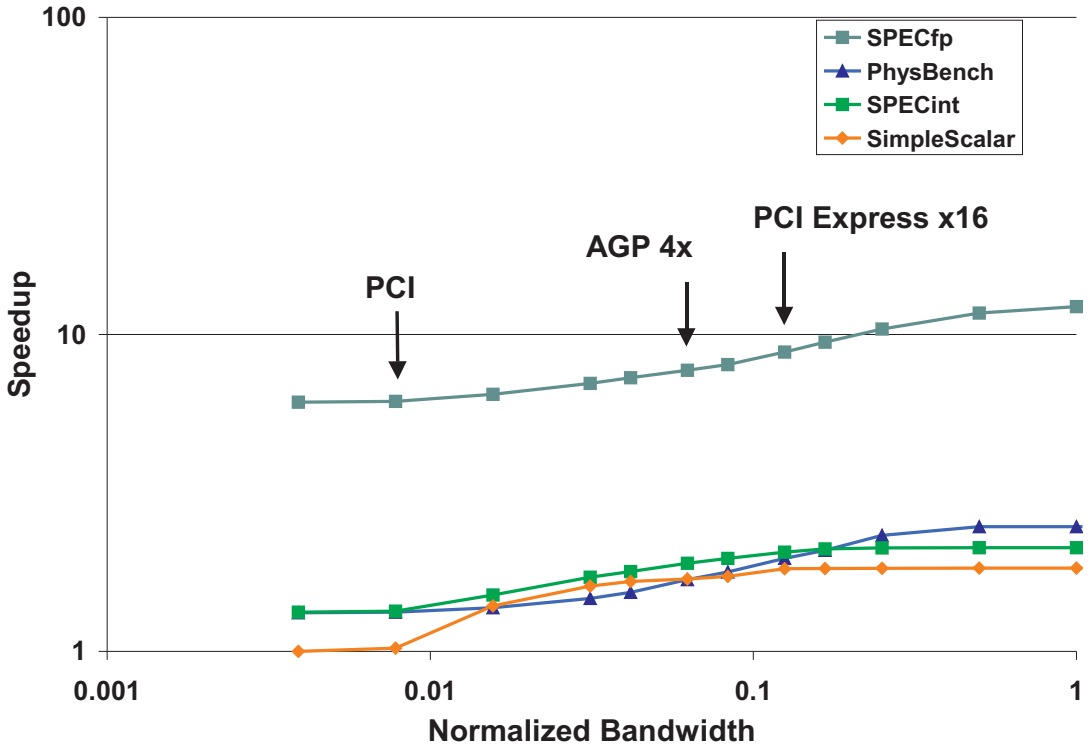

Figure 3.9: Speedup over sequential processor for varying bandwidth constraints. Real benchmarks.

every 8 clocks with 1,000 cycles of latency. Slowdown is worse than with 100,000 cycles of latency, but the worst benchmark set (SPECfp) can still achieve ≈65% of the ideal performance. Comparing latency (Figure 3.7) to bandwidth (Figure 3.8) constraints, SPECfp and PhysicsBench has more performance degradation than under a pure-latency constraint, but SPECint performs better, suggesting that SPECint is less sensitive to bandwidth.

The above plots suggest that a heterogeneous system attached without a potentially-expensive, low-latency, high-bandwidth communication channel can still achieve much of the potential speedup.

To further evaluate whether GPU-like systems could be usefully attached using even lower bandwidth interconnect, we measure the sensitivity of performance to bandwidth for instruction latency 100. Figure 3.9 shows the speedup for varying bandwidth. Bandwidth (x-axis) is normalized to 1 cycle per datum, equivalent to about 16GB/s in today's systems. Speedup (y-axis) is relative to the workload running on a sequential processor.

SPECfp and PhysicsBench have similar sensitivity to reduced bandwidth, while SPECint's speedup loss at low bandwidth is less significant (Figure 3.9). Although there is some loss of performance at PCI Express speeds (normalized bandwidth = 1/8), about half of the potential benefit of heterogeneity remains at PCI-like speeds (normalized bandwidth = 1/128). At PCI Express x16 speeds, SPECint can achieve 92%, SPECfp 69%, and PhysicsBench 78% of the speedup achievable without latency and bandwidth limitations.

As can be seen from the above data, heterogeneous systems can potentially provide significant performance improvements on a wide range of applications, even when system cost sensitivity demands high-latency, low-bandwidth interconnect. However, it also shows that applications are not entirely insensitive to latency and bandwidth, so high-performance systems will still need to worry about increasing bandwidth and lowering latency.

The lower sensitivity to latency than to bandwidth suggests that a shared-memory multi-core heterogeneous system would be of benefit, as sharing a single memory system avoids data copying when migrating tasks between processors, leaving only synchro-

nization latency. This could increase costs, as die size would increase, and the memory system would then need to support the needs of both sequential and parallel processors. A high-performance off-chip interconnect like PCI Express or HyperTransport may be a good compromise.

3.5.5 Efficiency

Earlier we have shown that heterogeneous systems provide significant performance benefits over a wide range of parameters. In this section, we examine the potential area and power efficiency gains.

Earlier work has shown examples where their particular designs of heterogeneous systems are capable of improved area and power efficiency [13, 14]. Our motivation for evaluating efficiency is to find the parameters of the heterogeneous design that maximizes efficiency.

To calculate efficiency, we need both the performance and cost of each design point. We use performance normalized to the runtime of a serial processor as our performance metric (Figure 3.5). As before, we schedule workloads among the two processors to minimize runtime, so our efficiency metrics measure the efficiencies of the highest-performing scheduling. We use the data presented in Figures 3.3 and 3.4.

To estimate cost, we assume that the area and power scales linearly with increasing parallelism, and that Pollack's Rule [1] regarding complexity holds when scaling the instruction latency of the core. Hill's analysis [16] of multi-core scaling using Amdahl's Law uses the same performance-complexity scaling rule.

Pollack's Rule states that the area and power complexity of a processor scales proportional to the square of its performance, thus power/performance (energy per instruction, EPI) scales linearly with performance. In this study, we use the inverse of instruction latency as performance in Pollack's Rule. One limitation of Pollack's Rule is that it may not generalize beyond cores already existing, in particular to very-low-cost cores. Pollack's Rule assumes one can continue to scale processor designs arbitrarily small, resulting in

arbitrarily high area and power efficiency.

We normalize the performance and cost against the sequential processor. For cost calculations, a parallel processor with latency=1 supporting parallelism=1 is equivalent to a sequential processor. The area and EPI cost functions are described below.

3.5.6 Area Cost

When computing the area cost, we assume the hardware needs to be built large enough to accommodate the parallelism of the entire benchmark set. Although it is possible to build hardware with less parallelism than what the workload requires, this is similar to increasing the instruction latency as the excess tasks need to share the same hardware, which causes the optimal workload balance to change. The optimal workload balance is used when scheduling for a higher-latency parallel processor, but building insufficient hardware and scaling the parallel processor runtime leads to a sub-optimal workload balance.

Average parallelism for each workload (Figure 3.4) as well as the standard deviation across the set of benchmarks (not shown) is measured. We do not use peak parallelism as the metric as we wish to avoid outliers. Our area cost assumes we support parallelism of two standard deviations above the average parallelism plotted in Figure 3.4. Using a value other than two standard deviations does not appreciably change the shape of the area efficiency plot in Figure 3.11. Since we are not building application-specific accelerators, we do not use an area cost specific to each benchmark.

Equation 3.1 illustrates our area cost function:

$$cost_{area} = \frac{1.0 + parallelism}{latency^2} \tag{3.1}$$

$$parallelism = \frac{\sum_{benchmarks} parallelism_{benchmark}}{n_{benchmarks}} + 2\sigma$$

The normalized area cost is the sum of two terms: the area of the sequential processor (1.0), and the area of the parallel processor, which scales linearly with parallelism sup-

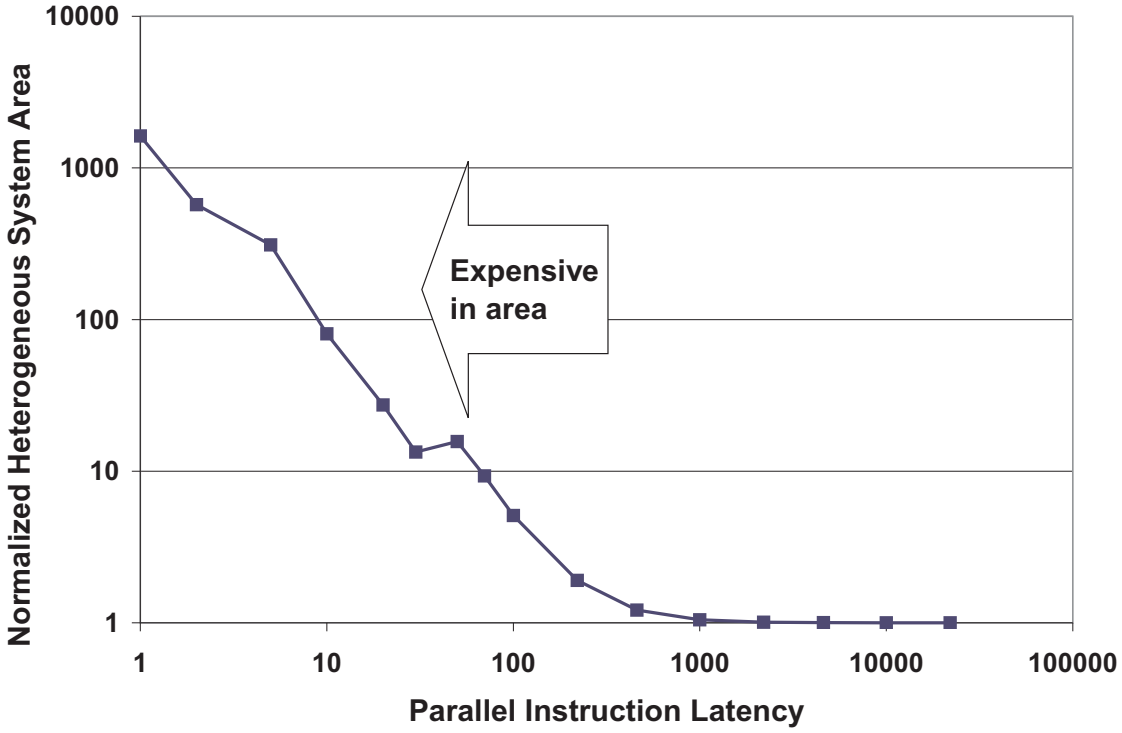

Figure 3.10: Heterogeneous system area, normalized to area of sequential processor.

ported and scales as the inverse square of instruction latency following Pollack's Rule. We support parallelism two standard deviations above the average parallelism in our benchmark set.

The area cost relative to the sequential processor is shown in Figure 3.10. The parallelism extracted from our benchmarks is an upper bound, so we expect the area cost as presented to also be an upper bound. Nevertheless, building a parallel processor capable of providing the parallelism demanded by our benchmark set with instruction latency below 10 likely uses too much area to be built.

Our area efficiency metric is performance per area, which should be maximized. This metric is plotted in Figure 3.11 and illustrated in Equation 3.2. We discuss in Section 3.5.8.

$$area\ efficiency = \frac{speedup}{cost_{area}} \tag{3.2}$$

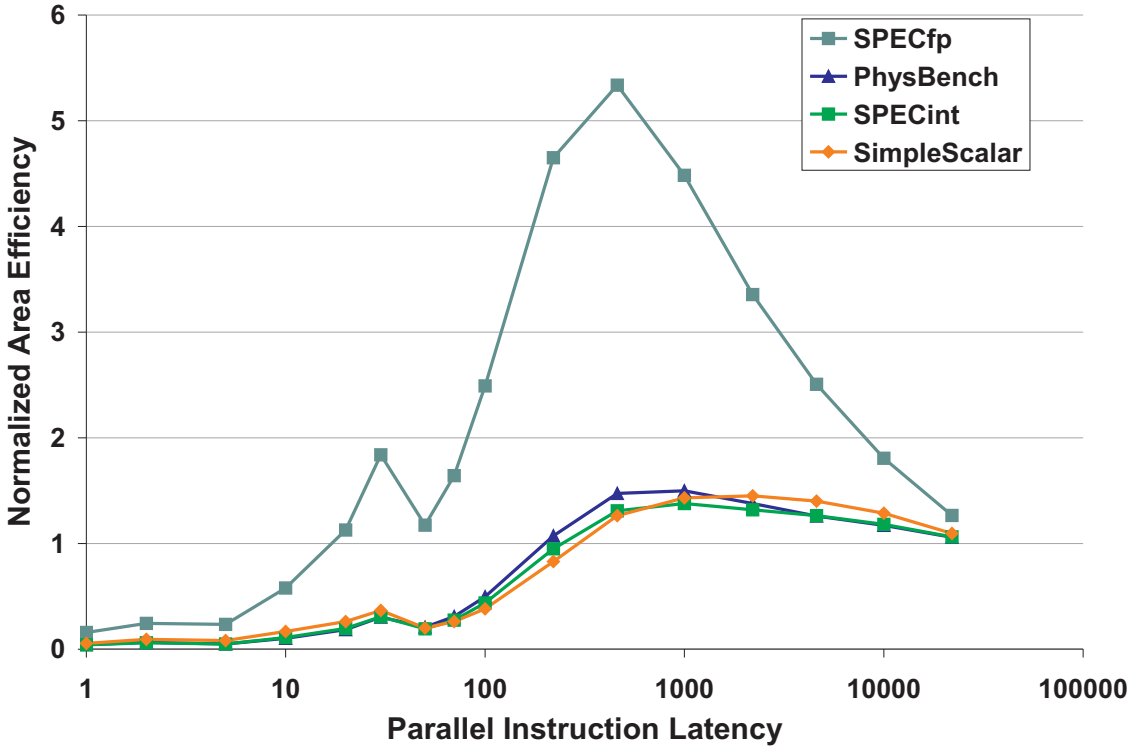

Figure 3.11: Normalized area efficiency. Real benchmarks.

3.5.7 Energy Per Instruction

Power consumption, unlike area, can be reduced on a per-workload basis by turning off unused portions of hardware. Thus, we can model energy usage by estimating the EPI of each core type, and weighting them by the number of instructions executed on each core type. It is assumed that it is possible to shut down part or all of the cores when it is not processing instructions, by clock gating or other means.

The following equations illustrate our EPI cost function.

$$EPI_{sequential} = 1$$

$$EPI_{parallel} = latency^{-1}$$

$$EPI_{system} = \frac{insns_{seq.} \times EPI_{seq.} + insns_{par.} \times EPI_{par.}}{insns_{total}} \tag{3.3}$$

We set the EPI for the sequential core to 1 and assume that EPI scales inversely as instruction latency, following Pollack's Rule. System EPI is the weighted average of sequential EPI and parallel EPI.

System EPI, plotted in Figure 3.12, is the cost function we wish to minimize.

3.5.8 Efficiency Results

As SPECfp has the highest performance improvement from a heterogeneous processor (Figure 3.5), it is not surprising that SPECfp also has the best area efficiency (Figure 3.11).

To maximize area efficiency, the parallel processor instruction latency should be near 500. At this design point, SPECfp can achieve 5x, SPECint about 1.47x, and PhysicsBench about 1.31x the performance per area compared to using only a sequential processor. For comparison, we estimate the latency of current GPUs to be 100 (See Section 3.3.4).

SPECfp shows the best (lowest) EPI, as SPECfp has more parallelism and is able to use the more efficient parallel core more often (Figure 3.3). At 100 cycles of instruction latency, SPECfp can achieve an EPI of 0.05, SPECint 0.30, and PhysicsBench 0.26, all significantly

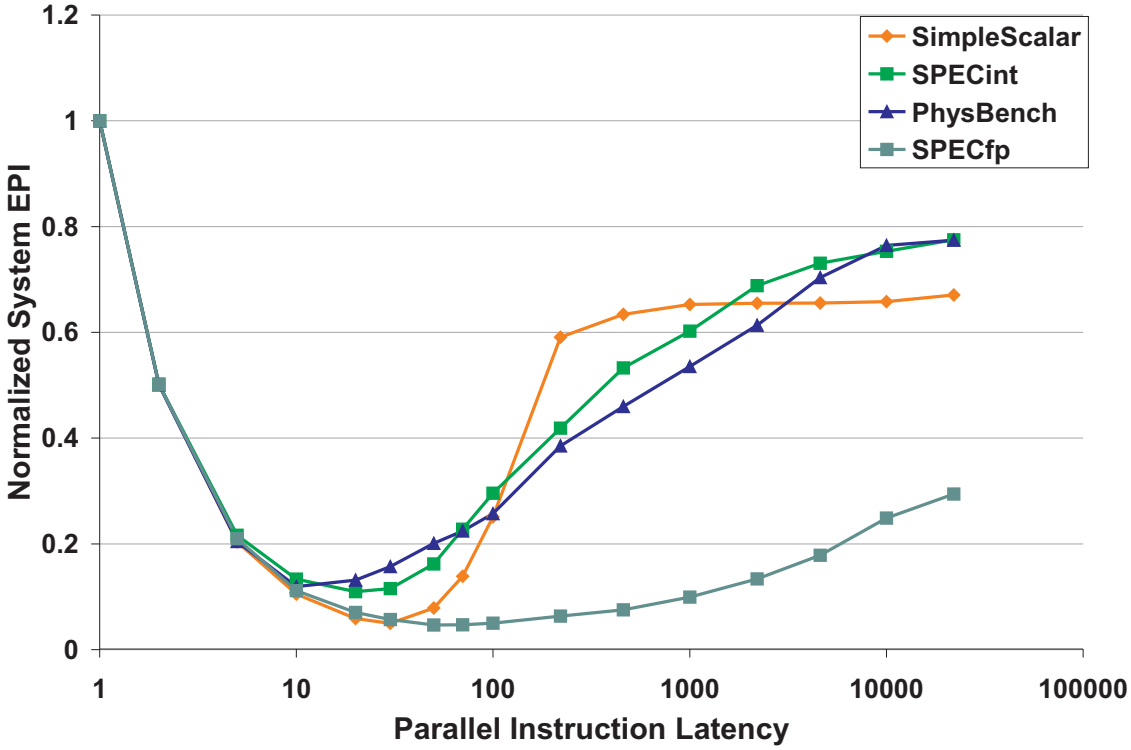

Figure 3.12: Normalized energy per instruction (EPI). Real benchmarks.

better than using only the sequential core (EPI = 1). The parallel processor design resulting in lowest system EPI has instruction latency around 50-100 for SPECfp, 20 for SPECint, and 10 for PhysicsBench. At the optimal CPI for each benchmark set, SPECfp can achieve EPI=0.047 at latency 70, SPECint EPI=0.11 at latency 20, and PhysicsBench EPI=0.12 at latency 10.

Especially for SPECint and PhysicsBench, the optimal instruction latency for EPI is significantly lower than current GPUs. This result is quite different from area efficiency, where underutilized high-cost cores reduces area efficiency at low instruction latency.

With workloads scheduled for performance, it appears that the optimal area and EPI design points are conflicting. If Moore's Law continues to hold, future designs will become increasingly power-limited rather than area-limited. As our data suggests that both performance and EPI improves with lower-latency parallel processors, we believe optimal designs in the future should have somewhat lower instruction latency and higher complexity than current GPUs.

3.6 Conclusion

We have modeled the abstract behavior of a heterogeneous system with a sequential processor and a parallel processor with higher instruction latency. We analyzed a set of benchmarks, and optimally scheduled them onto the two processor types.

We showed that there is a significant, but limited, amount of parallelism that can be extracted from our workloads, and that instruction latency of the parallel processor was a significant factor in performance. Latency and bandwidth, while significant factors, have comparatively minor effects on performance. Latency and bandwidth characteristics of PCI Express was sufficient to achieve most of the available performance, and about half of the available performance could be attained using a low-performance PCI-like interconnect.

We also showed that, assuming Pollack's Rule, area efficiency and EPI can be improved

with a heterogeneous system, and that the optimal area-efficient design differs from the optimal power-efficient design. The optimal area-efficient design has a relative instruction latency of around 500, while the optimal power-efficient design has latency around 20 to 70, depending on workload.

Note that since our results are normalized to the sequential processor, our results scale as processor designs improve. As sequential processor performance improves in the future, the absolute performance of the parallel processor will also need to improve to match.

Heterogeneous systems have the potential to improve performance and efficiency for single application workloads, and still do so with low-performance interconnect. We see little reason from a performance and efficiency perspective not to build heterogeneity in some form. However, today's restrictive hardware and programming models may further limit attainable speedups and increase software developer effort. Intel's upcoming Larrabee [26] claims to improve ease of development by being x86-compatible.

As systems become more power constrained and less area constrained, our data suggests that the parallel processor should have faster sequential performance than what is commonly found in today's GPUs, increasing performance and power efficiency at the expense of some area efficiency.

References

[1] S. Borkar. Thousand Core Chips — A Technology Perspective. In *Proc. 44th Annual Conference on Design Automation*, pages 746–749, 2007.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings vol. 30*, pages 483–485, 1967.

[3] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of ACM*, 31(5):532–533, 1988.

[4] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, K. Chang. The case for a single-chip multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, 1996.

[5] Nvidia. Compute Unified Device Architecture Programming Guide Version 2.0. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf.

[6] S. Che, J. Meng, J. W. Sheaffer, K. Skadron. A Performance Study of General Purpose Applications on Graphics Processors. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[7] L. Nyland, M. Harris, J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems 3*, 2007.

[8] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93/6, DEC WRL, 1993.

[9] M. S. Lam, R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th International Symposium on Computer Architecture*, pages 46–57, 1992.

[10] M. A. Postiff, D. A. Greene, G. S. Tyson, T. N. Mudge. The Limits of Instruction Level Parallelism in SPEC95 Applications. *ACM SIGARCH Computer Architecture News*, 27(1):31–34, 1999.

[11] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, D. Connors. Chip multi-processor scalability for single-threaded applications. *ACM SIGARCH Computer Architecture News*, 33(4):44–53, 2005.

[12] M. D. Hill, M. R. Marty. Amdahlś Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[13] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. 31st International Symposium on Computer Architecture*, page 64, 2004.

[14] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. 36th IEEE/ACM International Symposium on Microarchitecture*, page 81, 2003.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[16] J. Hensley. Close to the Metal. ACM SIGGRAPH 2007 courses, course 24 article 7, 2007.

[17] AMD. 2007 Financial Analyst Day. http://download.amd.com/Corporate/MarioRivasDec2007AMDAnalystDay.pdf, 2007.

[18] T. Y. Yeh, P. Faloutsos, S. J. Patel, G. Reinman. ParallAX: an architecture for real-time physics. In *Proc. 34th International Symposium on Computer Architecture*, pages 232–243, 2007.

[19] GeForce 8 Series. http://www.nvidia.com/page/geforce8.html.

[20] GeForce GTX 280. http://www.nvidia.com/object/geforce_gtx_280.html.

[21] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software ISPASS*, 2007.

[22] T. Austin, E. Larson, D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[23] T. Sherwood, E Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS*, 2002.

[24] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997.

[25] B. Holden. Latency Comparison Between HyperTransport and PCI-Express in Communications Systems. http://www.hypertransport.org/docs/wp/Latency_Comparison_HyperTransport_PCIe_in_Communications_Systems.pdf.

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *International Conference on Computer Graphics and Interactive Techniques*, 2008.

Chapter 4

Conclusion

In this section we summarize the preceding two chapters, comment on limitations, and discuss future work.

4.1 Summary

This thesis explores CPU-GPU heterogeneous architectures. Chapter 2 presented Pangaea, a tightly-integrated heterogeneous multicore design where the GPU and CPU share the memory hierarchy. The limit study in Chapter 3 explores the potential for GPU-CPU heterogeneous systems to accelerate general-purpose applications and some requirements of the architecture to do so.

4.1.1 Pangaea

In Chapter 2, we presented Pangaea, a heterogeneous GPU-CPU multicore architecture. Pangaea aims to support only computation with the GPU cores, tighter integration for lower communication latency, with a shared memory programming model.

As Pangaea is targeted for compute applications only and not graphics, the GPU's graphics-specific hardware can be removed. We find that in the Intel X4500 GPU, the graphics-specific hardware occupies an area equivalent to 9 processing cores (Execution Units, EU), and power consumption equivalent to 5 EUs. As the X4500 GPU has on the order of 10 EUs, there is a large area and power savings by removing legacy 3D graphics support. In our 2-EU prototype, the area and power savings are 81% and 70%, respectively. In addition, the addition of a dedicated thread spawner while removing much of

the legacy graphics pipeline results in a reduced thread spawn latency, from ~1500 clocks down to 26.

Pangaea also boasts tighter integration between CPU and GPU than current GPU compute systems. This is accomplished by locating the GPU cores on the same chip as the CPU, and providing fast signaling mechanisms to communicate. Fast communication is accomplished by IA32 ISA extensions implementing fast user-level interrupts on top of existing cache coherency mechanisms. The SIGNAL instruction is a modified store to spawn threads, the EMONITOR instruction sets up monitoring for writes to address ranges to trigger user-level interrupts, and the ERETURN instruction returns from a ULI routine.

Pangaea implements Exo's [1] *address translation remapping* and *collaborative exception handling* using similar mechanisms, allowing Pangaea's GPU and CPU to shared a single virtual memory address space. Pangaea also shares the last level cache between GPU and CPU, supporting efficient collaboration between CPU and GPU (e.g. producer-consumer).

A synthesized prototype implemented on an FPGA is also presented.

4.1.2 Limit Study

The limit study presented in Chapter 3 explores GPU compute-like acceleration for general-purpose applications. The limit study makes optimistic assumptions about the parallelizability of an instruction trace and attempts to model the upper-bound of performance under various constraints found in typical GPU compute systems. We then assume Pollack's Rule [2] concerning processor scaling and observe some general trends in GPU compute core design.

GPUs support large amounts of parallelism, but typically have high register read-after-write latency [3], such that the sequential performance of GPUs is lower than CPUs. GPUs are typically attached to the rest of the system via an off-chip interconnect like PCI Express. This limit study looks at the potential parallelism usable under the constraints of lower GPU sequential performance, and limited bandwidth and latency of the interconnect.

By analyzing instruction traces from general-purpose applications, the amount of parallelism available to be extracted by GPU-like systems can be measured. Like earlier limit studies on available parallelism [4, 5], we find that under optimistic assumptions there is a large amount of parallelism in general-purpose applications, but we find that the potential speedups are limited by the poor sequential performance of GPUs. We also showed that limitations on latency and bandwidth of the interconnect have a relatively minor effect on achievable performance, such that it is not critical that a GPU needs to be on the same die as the CPU for performance reasons. Anecdotally, this appears to be the case in application examples, where most applications spend a very small proportion of time transferring data between CPU and GPU. [6].

Using our performance model and assuming Pollack's Rule on area and power scaling holds, we can derive area and power efficiency metrics as the GPU core performance scales. We see that with the exception of the fully-serial microbenchmark, heterogeneity improves performance and efficiency. We find that the optimal area-efficient parallel core has register read-after-write latency around 500x greater than the CPU, while the optimal power-efficient design has a latency of 50-70. For comparison, we estimate the modern GPU's latency to be ∼100.

Future GPUs will need to rely less on extreme amounts of parallelism if it wishes to accelerate ordinary applications.

4.2 Relation Between Works

The contributions presented earlier were toward improving accelerating general-purpose computation on the GPU. Pangaea (Chapter 2) proposed an architecture, while the limit study in Chapter 3 explored the design and application space around it.

The limit study was conceived to address some of the limitations of Pangaea's narrow coverage of design space and applications. Chapter 2 explored only one design point for the CPU and GPU, and only a small set of highly-parallel workloads. To improve on that, the limit study explores a wider range of benchmarks as well as a larger span of varying

GPU core performance.

One interesting set of data from the limit study involves the two benchmarks that were used in evaluating Pangaea (*linear* and *sepia*). As can be seen in Figures 3.2 and 3.3, these two benchmarks are almost entirely parallelizable. Although they get good speedup on Pangaea, these two benchmarks are not representative of the typical program. However, there are several benchmarks in the SPECfp suite (Figure 3.2) which approach the parallelism in *linear* and *sepia* (*equake*, *galgel*, and *mgrid*). These may be good targets for GPU acceleration on today's architectures.

4.3 Potential Applications

Pangaea's proposed architecture can be used directly in designing processors. Our area and power analysis (Section 2.5) shows that the current approach to GPU compute is inefficient because a significant part of the GPU is unused in general-purpose compute applications, and should be removed. Our FPGA prototype shows that the architecture is functional as proposed.

The data from the limit study presented in Chapter 3 can be used to identify applications which may be amenable to GPU acceleration. Applications with large amounts of parallelism (Figure 3.2) may be suitable candidates for GPU acceleration on today's architectures. As GPU cores improve in sequential performance, applications with less parallelism will become suitable for acceleration.

The limit study also gives some suggestions on how future general-purpose accelerators should be designed. In particular, the limit study suggests that GPU architectures should be made to rely less on parallelism to improve utilization, but GPUs need to improve single-threaded performance in order to accelerate a wider assortment of workloads. Figure 3.5 shows that potential speedup is tied strongly to the parallel processor's register read-after-write latency. The same is true for minimizing energy per instruction (Figure 3.12), which will become increasingly important as designs become increasingly power limited.

4.4 Limitations and Future Work

There are some limitations with the work presented in the preceding chapters. In this section we point out some limitations and propose work to remedy them.

4.4.1 Pangaea

The most obvious drawback of the Pangaea design as proposed is that it does not support legacy 3D graphics acceleration. It was a design decision to evaluate GPU cores for compute purposes only. This design decision could be changed if one does not remove the legacy 3D hardware, but instead merely turns them off when unused.

Another limitation of the performance evaluation of Pangaea is that the benefits of the proposed communication mechanisms were not isolated. Future work to remedy this limitation would involve a more detailed prototype that accurately modeled the communication mechanisms. Some insight on whether a fast communication mechanism is necessary can be gained by the limit study we showed in Chapter 3, however.

Pangaea presents only a small set of benchmark kernels. Manually parallelizing a large benchmark set optimally is unfortunately infeasible. We partially address this limitation in the limit study, where we automatically extract parallelism from a large set of single-threaded benchmarks.

4.4.2 Limit Study

Typical of limit studies, a limitation of the limit study presented in Chapter 3 does not model the machine in detail. In our limit study, we modeled processors capable of executing one instruction in some fixed time, without considering effects such as memory latencies or varying instruction latencies depending on the type of instruction. We also do not model a thread-based architecture with inter-thread communication latency as is popular in today's hardware. Although it is infeasible to model in detail while still sampling a large design space, future work should take insights from the limit study and do

detailed modeling of interesting applications and design points. Likewise, our assumption of Pollack's Rule may not generalize to cores that do not yet exist.

Another limitation with the limit study lies in its optimistic treatment of branches. Currently, branch directions are assumed to be known *a priori*, essentially ignoring branches. Although many branches are predictable and optimal manual parallelization may remove a large number of branches from the code, mispredicted branches still limit parallelism. Future work should perform condition dependence analysis and model the effects of mispredicted branches on available parallelism.

One major issue with GPU computation that was not addressed by this thesis is the issue of ease of programming. Although there have been improvements from using programmable shaders to C-like high-level languages [3, 7] to shared virtual address spaces [1], GPUs still remain difficult to program. It would be useful to objectively evaluate the ease of use and performance of various programming models.

References

[1] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

[2] S. Borkar. Thousand Core Chips — A Technology Perspective. In *Proc. 44th Annual Conference on Design Automation*, pages 746–749, 2007.

[3] Nvidia. Compute Unified Device Architecture Programming Guide Version 2.0. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf.

[4] M. S. Lam, R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th International Symposium on Computer Architecture*, pages 46–57, 1992.

[5] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM.

[6] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.