GPU performance modeling and optimization

GPU Performance Modeling and Optimization

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 18 oktober 2016 om 14.00 uur

door

Ang Li

geboren te Shanxi, China

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. B. Smolders
promotor:	prof.dr. H. Corporaal
promoter:	prof.dr. A. Kumar (Technische Universität Dresden)
leden:	prof.dr. K. Goossens
	prof.dr.ir. P.H.N. de With
	prof.dr. Y. Ha (National University of Singapore)
	prof.dr. W.F. Wong (National University of Singapore)
	prof.dr. V. Bharadwaj (National University of Singapore)
	dr.ir. C. Nugteren

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

GPU PERFORMANCE MODELING AND OPTIMIZATION

ANG LI

A THESIS SUBMITTED FOR THE JOINT DEGREE OF DOCTOR OF PHILOSOPHY BETWEEN

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE

AND

DEPARTMENT OF ELECTRICAL ENGINEERING
EINDHOVEN UNIVERSITY OF TECHNOLOGY

2016

Doctorate committee:

prof.dr.ir. B. Smolders	Eindhoven University of Technology
prof.dr. H. Corporaal	Eindhoven University of Technology
prof.dr. A. Kumar	Technische Universität Dresden
prof.dr. K. Goossens	Eindhoven University of Technology
prof.dr.ir. P.H.N. de With	Eindhoven University of Technology
prof.dr. Y. Ha	National University of Singapore
prof.dr. W.F. Wong	National University of Singapore
prof.dr. V. Bharadwaj	National University of Singapore
dr.ir. C. Nugteren	Blippar Layar

This work is supported by the Research Scholarship from National University of Singapore.

Cover designed by Yanfei Li.

Printed by CPI-Koninklijke Wöhrmann B.V., The Netherlands.

A catologue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-4155-3

Acknowledgments

First I would be greatly thankful to my wife, my parents and my brother for their continuous support in my life. I would like to thank my supervisor Prof. Henk Corporaal who offered me the great opportunity to enroll in the joint-PhD program so that I had the chance to join the ES group and Eindhoven University of Technology (TU/e). His insightful comments and sharp questions always motivate me a lot and make me realize that there is always a better approach. I would thank my supervisor Prof. Akash Kumar. Without him, I could not have the chance to pursue the PhD degree in National University of Singapore (NUS). His smartness, patience and the extremely effectiveness in all research, teaching and life always remain the objective for me to follow. He has helped me resolve quite a lot of troubles I feel so frustrated and incapable to deal with. I would also like to thank my co-supervisor Prof. Yajun Ha, for his great support during my stay in NUS and offered me the high-quality journal articles for review.

I would like to thank Prof. Bart Smolders, Prof. Veeravalli Bharadwaj, Prof. Weng-Fai Wong, Prof. Peter de With, Prof. Kees Goossens, Prof. Ana Lucia Varbanescu and and Dr. Cedric Nugteren for serving on my dissertation committee for both sides. Their valuable feedback significantly improved the thesis.

I would like to thank Prof. Y.C. Tay for getting me in the module about performance modeling for compute systems, which was the most excellent class I ever took. The great theory I learnt in the module about closed queuing-network and transit analysis are the most initial sources for me to derive the idea about X-model presented in this thesis. I would like to thank Dr. Leon Shuaiwen Song for guiding and working with me on the latest several papers. Without him, I could not be so efficient in the past year and completed my PhD on time. I would like to thank to Dr. Weifeng Liu for his great help in the experiments of the SC-15 paper and the collaboration in the EuroPar-16 paper.

I would like to thank all the members in the PARsE group: Gert-Jan van den Braak, Maurice Peemen, Mark Wijtvliet, Luc Waeijen, Roel Jordans, Sohan Walimbe for their comments and suggestions on my presentations during the biweekly PARsE meetings. Special thanks to Gert-Jan for his assistance in the ICS-15 paper and SC-15 paper. Special thanks to Mark for his help in designing and setting up the environment to measure the power of the Jetson-TX1 board

for the ICS-16 paper.

Sincerely thank to my great friends Li Yonghui, Tang Qi, Chen Xin, Geng Tong, Li Zechuan, Shi Runbin, Jiao Hailong, Wang Wenjin, Chita, Wang Wenfeng, He Yifan, Wang Qing and She Dongrui. Thanks to Yonghui for the great help that I can never count up. Thanks to Tang Qi for sharing the wonderful wine with me, the sharp and interesting conversation about politics, education and life in Austria. Thanks to Chen Xin, Geng Tong and Li Zechuan for shopping together every Saturday morning, which are my most relaxing time for me in a week. Thanks to Runbin for the happy cooking and the comfortable conversation together in my early days in TU/e. Thanks to Hailong for being the host to organize our parties and driving me in the early morning to Schiphol Airport for the conferences in US. Thanks to Wenjin for his Philip member card. Thanks to Chita for being my neighbor and sharing me with the delicious Indian curry and Chinese tofu. Thanks to Wenfeng for passing me the form for applying to work in Flux during weekends. Thanks to Yifan for leaving me the userful hair-cutting tools. Thanks to Dongrui for transferring his seat/desk in the old Potential building and inviting me to his defense dinner. Thanks to everybody, without you all, my life in TU/e would fade and lost its color.

Thanks to my old friends during my first two years in Singapore: Zhao Wenfeng, Chen Yongzhen, Lin Longyang, Zhao Jian, Zhao Yang, Wang Xi, Wu Qiang, Anup Das, Amit Kumar Singh, Nam Khanh Pham, Mohammad Shihabul Haque, Tuan Nguyen, Hoo Chin Hau, Luo Shaobo, Wang Yi, Jin Jing, Li Weimin, Tang Liang, Wu Tong, Fang Fan, Hu Qikai, Zhong Guanwen, Tan Cheng, Wan Xuejun, Wang Zi, An Jianfeng and others.

Many thanks to everyone in the ES-group, who made working in the Lab a great and enjoyable experience. Special thanks to the head of the group Prof. Twan Basten for letting me attend the conferences to present my papers. Special thanks to our secretary Marja, for all the patience and substantial supporting in the involved paper-works.

Finally, thanks to my coming daughter, although you have not yet come to the world, you already bring me the great courage and the enthusiasm for the future life. Looking forward to see you after the PhD oral defense.

Abstract

The last decade has witnessed the blooming emergence of general-purpose Graphic-Processing-Unit computing (GPGPU). With the exponential growth of cores and threads in a modern GPU processor, how to analyze and optimize its performance becomes a grand challenge. In this thesis, as the modeling part, we propose an analytic model for throughput-oriented parallel processors. The model is visualizable, traceable and portable, while providing a good abstraction for both application designers and hardware architects to understand the performance and motivate potential optimization approaches. As the optimization part, we focus on each crucial component of a GPU streaming-multiprocessor, in particular registers-files, compute-units (SPU, DPU, SFU), caches (L1, L2, read-only, texture, constant) and scratchpad memory alternatively, clarify its underlying performance tradeoffs, and propose effective solutions to handle the tradeoffs in the design space. All the proposed optimization approaches are purely software-based. They are adaptive, transparent, traceable and portable, which leads to achievable and immediate performance gains for various existing GPU devices, especially for GPU integrated high-performance-computers (HPC).

Particularly, the first contribution in Chapter 3 is a novel visualizable analytic model called "X" that is specially for today's highly parallel machines. It comprehensively analyzes the interaction between the four types of parallelism (TLP, ILP, DLP and MLP) and two types of memory effects (local on-chip cache effect and remote off-chip memory effect), in terms of system throughput. The X-model acts as the theoretical basis of this thesis.

The second contribution in Chapter 4 is an effective auto-tuning framework to resolve the conflict between overall thread concurrency and per-thread register usage for GPUs. We discover that the performance impact from register usage is continuous, but from concurrency is discrete. Their joint-effects form a special relationship such that a series of critical-points can be pre-computed. These critical-points denote the best performance for each concurrency level. Therefore, the global optimum, which refers to the optimal number of registers per-thread, can be quickly and efficiently selected to deliver the best GPU performance.

The third contribution in Chapter 5 is an adaptive cache bypassing framework for GPUs. It uses a simple but effective approach to throttle the number of threads

that could access the three types of GPU caches –L1, L2 and read-only caches, thereby avoiding the fierce cache thrashing of GPUs, and significantly improving the performance for cache-sensitive applications.

In Chapter 6, we focus on a crucial GPU component that has long been ignored – the Special Function Units (SFUs) and show its outstanding role in performance acceleration and approximate computing for GPU applications. We exhaustively evaluate the numeric transcendental functions that are accelerated by SFUs and propose a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. It partitions the active threads into a PE-based slower but accurate path, and a SFU-based faster but approximated path, and tunes the relative partition ratio among two paths to control the tradeoffs between the performance and accuracy of the GPU kernels. In this way, a fine-grained and almost linear tuning space for the tradeoff between performance and accuracy can be created.

Finally, the last contribution in Chapter 7 is a novel approach for fine-grained inter-thread synchronizations on the shared memory of modern GPUs. By reassembling the low-level assembly-based micro-operations that comprise an atomic instruction, we develop a highly efficient, low cost lock approach that can be leveraged to set up a fine-grained producer-consumer synchronization channel between cooperative threads in a thread block. Additionally, we show how to implement a dataflow algorithm on GPUs using a real 2D-wavefront application.

Contents

Chapter 1

Introduction

High computing capability is always in high demand, especially for modern emerging applications, such as physical, chemical and biological simulations, data mining, computational financing, high-quality video processing, machine learning, big-data processing, virtual reality, etc. Traditionally, all applications are executed in Central-Processing Units (CPUs). However, the ever increasing compute demand substantially outstrips the scaling of CPU performance. Therefore, various compute accelerators are introduced, including Graphics Processing Units (GPUs) [1], Xeon Phi [2], Field-Programmable Gate Arrays (FPGAs) [3] and the recently shipped Micron Automata Processors [4]. Within all these accelerators, GPUs are most popular due to their easier accessibility, since a GPU, no matter integrated or independent, is the default component for displaying in a modern computer system.

Traditionally, GPUs are utilized for graphics purposes only. However, with the high demand of computing capability and the increased programmability of GPUs, people are seeking to apply GPUs also for (**G**)eneral-(**P**)urpose applications, known as **GPGPU** [1]. For some applications, GPUs are reported to achieve hundreds of times speedup over CPUs [5, 6, 7, 8, 9].

Although GPUs obtain great success and demonstrate much faster performance scaling [10], the ever-growing compute demand still enforces great pressure over the performance scaling of GPUs. On the other hand, with a completely divergent design principle, the throughput-oriented GPUs incorporate much larger volume of light-weighted cores and threads than the latency-oriented CPUs, which devote a large portion of their on-chip areas for caches. Therefore, conventional CPU-targeted optimizations strategies, especially for reducing latency, are no longer applicable for GPUs; the community requires new optimization approaches specially for GPUs. Even worse, when a GPGPU application shows certain performance on a GPU device, it is hard for the CPU developers to locate the GPU performance bottlenecks, since the latency bottlenecks are not necessarily the throughput bottlenecks, either in software or hardware.

This thesis attempts to answer the two fundamental questions about GPGPU performance: "how to explain and improve GPGPU performance", via *performance modeling* and *software-based optimization* approaches. We propose a high-level, visualizable analytic model for analyzing the performance of throughput-oriented parallel machines, with GPUs being the best representative. Meanwhile, we target various design tradeoffs for general GPGPU programs and present four primary software-based optimization strategies. The four strategies, focusing on GPU registers, caches, function units and scratchpad memory respectively, are validated on multiple GPU platforms in different generations to show their portability and great benefits.

The remaining part of this chapter is organized as follows. In Section 1.1 we briefly review GPU's history and the conventional graphic rendering pipeline. In Section 1.2, we summarize the development, the performance scaling and the research trends of GPGPU. In Section 1.3, we propose the research problems of this thesis. In Section 1.4, we list the contributions of this thesis. Finally, in Section 1.5, we draw an outline of the remaining chapters.

1.1 Traditional GPUs

According to *Wikipedia*, GPU is traditionally defined as *a specialized electronic circuit to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display*. In this section, we briefly describe the origin of GPU and the conventional design purpose of GPU — to process graphics via the graphics rendering pipeline.

1.1.1 GPU History

Each commodity hardware is designed with specific customer requirements from certain markets. GPU, as an indispensable component for modern computer systems, was born and grown with the demand of high-quality graphic display from video-game players. Early to 1970s, chips specialized for graphic utilizations had been implemented in the arcade system boards (Figure 1.1). The major reason is that the random-access memory (RAM) utilized as the frame buffers for the display of these video games were too expensive at that time. A good example for such specialized chips was *Fujitsu's MB14241 video shifter* (Figure 1.2), which was designed to accelerate the drawing of sprite graphics for various arcade games, e.g., *Gun Fight (1975), Sea Wolf (1976)* and *Space Invaders (1978)*. In 1982, the system boards for arcade games such as *"Robotron:2084", "Joust"* and *"Bubbles"* all included custom coprocessors for operating 16-color bitmaps [11]. In 1988, the *CPS-1* arcade system board developed by *Capcom* contained a graphics chipset that offered a 65,536 color palette and hardware support for sprites, scrolling and multiple playfields. From early 1990, CPU-assisted real-time 3D graphics became increasingly popular in arcade, computer and console games, which led to the high demand for hardware-accelerated 3D graphics, e.g., *Sega Model, Namco System-22* arcade system boards and *Saturn, PlayStation* video game consoles.

At the same time, *OpenGL* [12] appeared as a professional graphics API. Early implementations of OpenGL were based on software, but soon hardware implementation became the trend. Meanwhile, *DirectX* [13] appeared as the popular graphics API for Windows game developers. To be compatible with these fast developed graphics APIs, 3D accelerator cards started to add substantial hardware stages beyond the conventional 3D rendering pipeline, which led to the release of the world's first

Figure 1.1: Arcade Machine

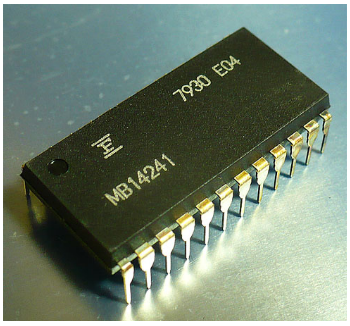

Figure 1.2: Fujitsu MB14241

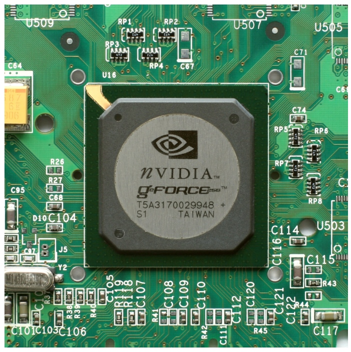

Figure 1.3: NVIDIA GeForce 256 GPU

genuine GPU product – the NVIDIA GeForce 256 [14] (Figure 1.3). By "genuine", NVIDIA's official website technically describes a GPU as "*A single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.*". Later in 2001, NVIDIA announced the first GPU that supported programmable shading[1], known as GeForce 3, which was adopted in the Microsoft Xbox console. In 2002, *ATI* introduced *Radeon 9700*, which was the world's first Direct3D 9.0 GPU, and in which pixel and vertex shaders were capable to implement floating-point operations and loops. With these features, GPUs became much more flexible and offered orders of magnitude performance speedup for operations upon image-like arrays than their CPU counterparts. The introduction of NVIDIA GeForce 8800 further improved the flexibility of GPUs by integrating generic streaming processing units. Such increased flexibility, together with the tremendous potential performance benefit, led to the tendency of GPGPUs.

1.1.2 GPU Graphics Pipeline

GPU was originally designed to process graphics via the so-called *graphics rendering pipeline*. Rendering refers to the process of generating image on the display (e.g., a monitor) from the model descriptions. Figure 1.4 shows a 3D graphics rendering pipeline, which reads in the descriptions of 3D objects in terms of vertices and primitives. Primitives here refer to the shapes or connected vertices, such as triangle, point, line and quad. The pipeline outputs the color values for all the pixels on the display. The graphics rendering pipeline is composed by the following stages:

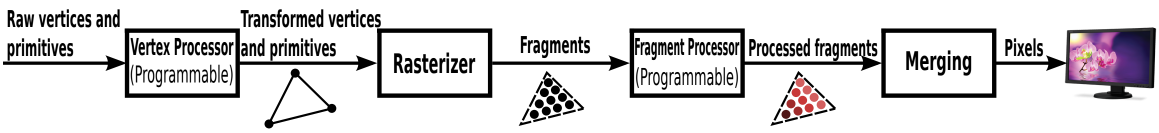

Figure 1.4: The 3D Graphics Rendering Pipeline

- **Vertex Processing.** It is performed by vertex processors, which transform individual vertices into a common coordinate system (e.g., via rotation, translation and scaling).

[1] *Shaders* are the short programs that describe the properties of a vertex or a pixel before being projected onto the screen.

- **Rasterization.** It is performed by rasterizers, which convert primitives into fragments[2].
- **Fragment Processing.** It is performed by fragment processors, which process individual fragments (e.g., binding texture).
- **Merging.** It is to combine all the processed fragments of primitives (in 3D space) into a 2D array of pixels for displaying.

For old GPUs, the four stages in the rendering pipeline were fixed. But soon (e.g., in GeForce 3), the vertex processing and fragment processing stages became programmable. People can write vertex shaders and fragment shaders to do custom transformations of vertices and fragments. The shader programs are in C-like style. Typical shading languages are *GLSL* (OpenGL Shading Language) [15], *HLSL* (High-Level Shading Language for Microsoft Direct3D) [16] and *Cg* (C for Graphics used by NVIDIA) [17].

1.2 GPGPU

With the enhanced programmability of GPUs (e.g., the vertex processors and fragment processors), GPGPU becomes possible. However, the real prosperity of GPGPUs could not appear without the generic programming models, such as *Compute-Unified-Device-Architecture* (CUDA) [10] and *Open-Computing-Language* (OpenCL) [18]. In this section, we introduce these models and the recent development of GPGPUs, attempting to answer the questions about *why GPGPUs become so popular? What are the utilizations of GPUs in different domains? What is the performance scaling of GPUs? What are the current popular GPGPU research topics?*

1.2.1 CUDA and OpenCL make GPGPU Popular

Prior to the introduction of CUDA and OpenCL, programming non-graphics applications on GPUs was extremely complicated and difficult, which required deep understanding on both the graphic rendering pipelines [19], the graphic programming interface (e.g., DirectX [13] and OpenGL [12]) and possibly the shader languages (e.g., Sh [20] and Brook [21]). Most of the GPGPU applications at that time were linear-algebra programs performing intensive mathematic operations on image-like arrays in a streaming fashion [22, 23, 19, 24, 25].

These programming difficulties had been greatly mitigated since CUDA was published in 2007. CUDA, as the world's first and probably the most widely accepted GPGPU programming framework, was designed to work with popular programming languages such as C, C++, Fortran, Matlab and Python. Under the persistent promotion by NVIDIA, both CUDA and GPGPU gained great success and had been utilized in various domains. As a direct response, the other major GPU vendor –

[2] *Fragments* are the pixels in 2D or 3D space that are aligned with the pixel grid, with attributes such as position and color.

Table 1.1: NVIDIA GPU Architecture Generations. Compute Capability (X.Y) is to describe the hardware version of a GPU: X is the major architecture generation (e.g., Kepler is 3, Maxwell is 5, etc.); Y is the minor architecture version in the same generation (therefore sharing the same ISA.

Arch.	Release Year	Compute Capability	Process	Most highlighted Features	Flagship GTX/Tesla/Jetson GPUs	Ref.
Tesla	2008	1.0, 1.1, 1.2, 1.3	65 nm	GPU baseline architecture	GTX8800, GTX9800, GTX280, Tesla1060	[45]
Fermi	2010	2.0, 2.1	40 nm	L1/L2 caches, dual scheduler	GTX480, GTX460, GTX580, Tesla2070	[46][47]
Kepler	2012	3.0, 3.2, 3.5, 3.7	40/28 nm	Floating-point performance	GTX680, GTX-TitanZ, Tesla-K10, Tesla-K20, Tesla-K40, Tesla-K80, Jetson-TK1	[48][49]
Maxwell	2014	5.0, 5.2, 5.3	28 nm	Power efficiency	GTX750Ti, GTX980, GTX-TitanX, Tesla-M40, Tesla-M60, Jetson-TX1	[50][51]
Pascal	2016	6.0	16 nm	3D Memory, numeric SMs	Tesla-GP100, GTX1080	[52]

AMD, together with Apple, IBM and Intel, published a unified programming standard, known as OpenCL [26] for heterogeneous platforms, including GPUs [18], CPUs [27] and FPGAs [3]. NVIDIA also announced the support of OpenCL thereafter [28].

Although OpenCL is more general and vendor-independent, CUDA is more widely-adopted for GPGPU developers. It offers much stronger lower-level controllability over the GPU hardware (e.g., cache prefetching and bypassing, register throttling, low-level synchronization, etc), which substantially facilitates the extraction of the remarkable computing power of modern GPGPUs. Moreover, the great portability of OpenCL comes at a cost — to migrate an OpenCL program written for GPUs to CPUs or FPGAs, significant efforts are always necessary to attain the expected performance. That is why in this thesis, CUDA, rather than OpenCL, is utilized as the GPU programming language. Besides, all the GPU platforms for evaluation in this thesis are NVIDIA GPUs. For that reason, we also use CUDA terminology in this thesis.

Thanks to CUDA and OpenCL, today GPGPUs are widely adopted for various application domains, including *Linear Algebra* [29], *Image & Video Processing* [30], *Searching* [31], *Physical & Biological Simulations* [32], *Data Mining* [33], *Bioinformatics* [34], *Machine Learning* [35], *Computational Finance* [36], etc. Most of the example applications for these domains can be found in the open-source GPGPU benchmarks, such as *Rodinia* [37], *Parboi* [38], *Shoc* [39], *Polybench* [40], *Mars* [33], *LonestarGPU* [41], *CUDA-SDK* [42] and *GPGPU-sim* [43]. Their characteristics are summarized in Chapter 2. In addition, the book *GPU Computing-Gems* [44] provides thorough descriptions about a broad domain of GPGPU applications.

1.2.2 GPGPU Performance Scaling

For NVIDIA GPUs, during the past decade, there are in total five major architecture generations: **Tesla, Fermi, Kepler, Maxwell** and **Pascal**. (see Table.1.1). The Tesla architecture [45] is the first CUDA-enabled GPU architecture and is already out of date now. It does not even appear in the recent official CUDA programming guide [53]. Fermi, as a direct response to the criticism from its competitor [54], introduced the two-level cache hierarchy and the functionality of multi-issuing. The Kepler GPUs are most high-lighted for their enormous compute capability, as they contained

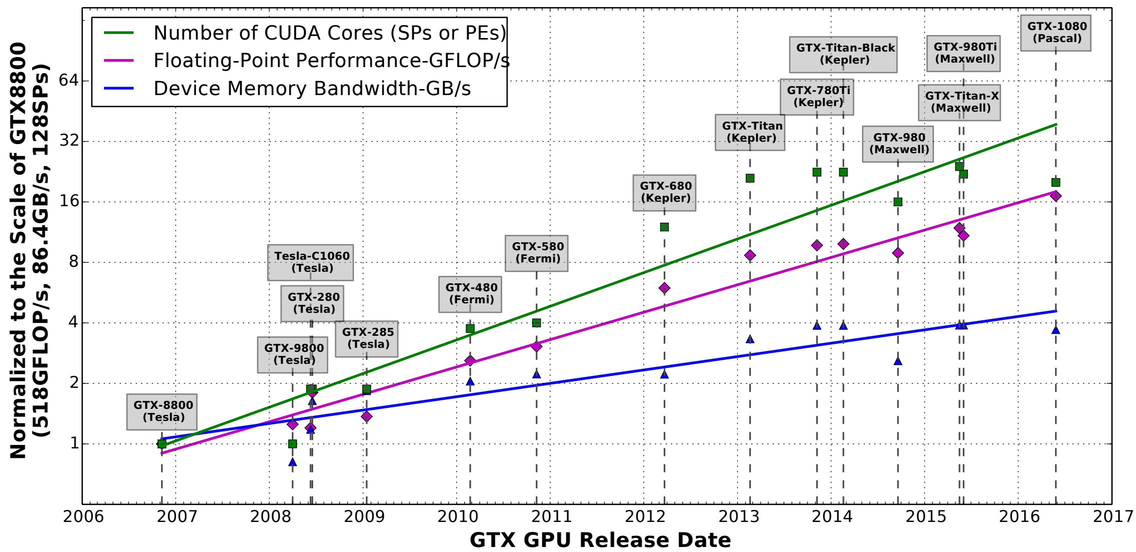

Figure 1.5: The Scaling of NVIDIA GTX Products for Desktop Utilizations

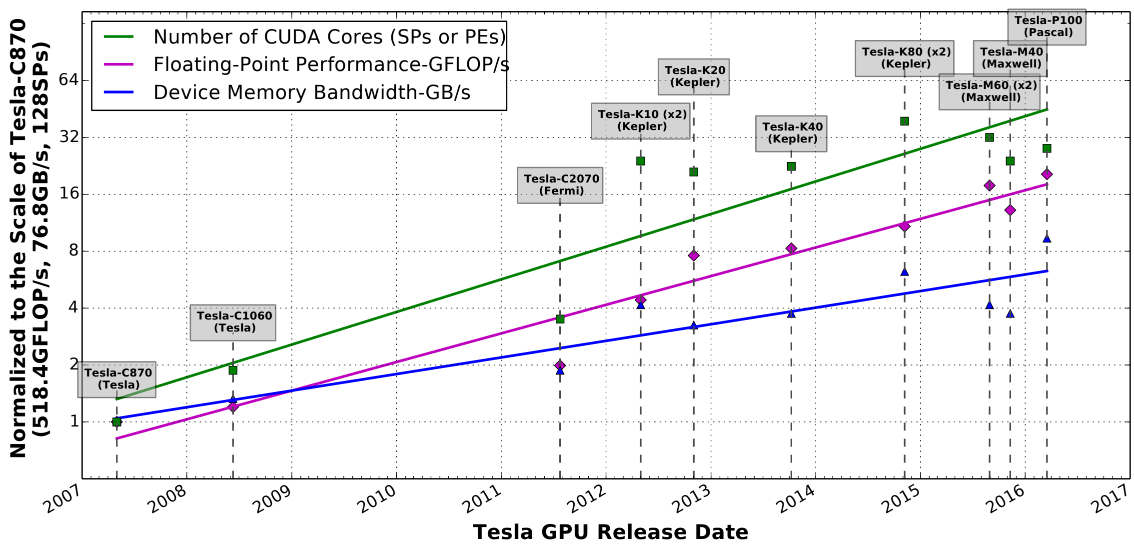

Figure 1.6: The Scaling of NVIDIA Tesla Products for Supercomputer Utilizations

the most number of CUDA cores per streaming-multiprocessor. From Maxwell, GPU started to put power-efficiency, in addition to performance, as its primary design principle. Finally, the latest Pascal architecture [52], which was announced early this year (2016), is known for introducing the 3D stacked memory and the ability to quickly process half-precision (16bits) calculations. Note, the *GTX product-line* is for desktop utilizations; the *Tesla product-line*[3] is for high-performance-computing (HPC) utilizations; the *Jetson TK1* and *TX1* are for embedded system (ES) utilizations.

Figure 1.5 illustrates the performance scaling of NVIDIA GTX and Tesla flagship GPU products in terms of CUDA cores, single-precision floating-point performance (GFLOP/s) and global memory

[3]The name "Tesla" is used by NVIDIA for both a GPU product line and a GPU architecture generation.

throughput (GB/s), normalized to the first CUDA-capable GPU — GTX8800 during the past decade. Specially, Figure 1.6 illustrates the performance scaling for NVIDIA Tesla Product GPUs, which represent the most advanced GPUs in each generation for HPC. The metrics are normalized to the first Tesla product – Tesla-C870.

As can be seen, the scaling of the three important performance metrics roughly comply with *Moore's Law* (i.e., performance doubles each two years, thus about 32 times in a decade). Additionally, we have the following observations:

- From GTX-Titan and Tesla-K10 onwards, the number of CUDA cores in a GPU does not increase much. This is due to the fall of CUDA cores per streaming-multiprocessors (SM) since Kepler — the number of CUDA cores per SM evolved from 32 in Tesla, to 48 in Fermi, to 192 in Kepler, to 128 in Maxwell and finally 64 in Pascal. Despite the stagnant core scaling, the deliverable floating-point performance has continuously increased in an exponential speed (red lines in Figure 1.5 and 1.6).
- The scaling of memory bandwidth remains far behind the scaling of cores or floating-point performance, which indicates that the memory-wall continuously remains the major challenge for harvesting GPU performance. In fact, also from Tesla-K10, the memory bandwidth scaling has slowed down significantly. However, such a big performance-scaling gap has substantially mitigated in the latest Pascal GPUs, which adopt the so-called High-Bandwidth-Memory 2 (HBM2) 3D-stacked memory technology [52]. This technology packs the memory dies in 3D and links them vertically via the through-silicon-vias (TSVs), which significantly reduces the wire length and the memory accessing latency while enhancing the accessing bandwidth.

1.2.3 GPGPU Research Trends

To further improve GPGPU performance and broaden the utilization of GPGPUs, contemporary GPGPU research mainly focuses on the following four topics:

Performance Scaling: As heterogeneous accelerators such as GPUs play a crucial role in the performance scaling towards exascale computing [55, 56, 57], continuously enhancing performance for these accelerators always remains a major research topic, from both software and hardware perspectives. This is also the focus of this thesis.

Energy Reduction: GPU is heavily criticized for its considerable power consumption. Therefore, efficiently reducing power while continuing the performance scaling is an important research topic for GPUs. Typical methods including clock-gating [58, 59], power-gating [60, 61, 62] and DVFS [63, 64]. Figure 1.7 summarizes the power consumption for the aforementioned GTX and Tesla GPUs with the evolving of CMOS manufacturing process. Figure 1.8 shows their energy efficiency (Gflop/joule or flop/s per watt). As can be seen, the energy efficiency of GPUs continuously scales with improved architectures and manufacturing processes.

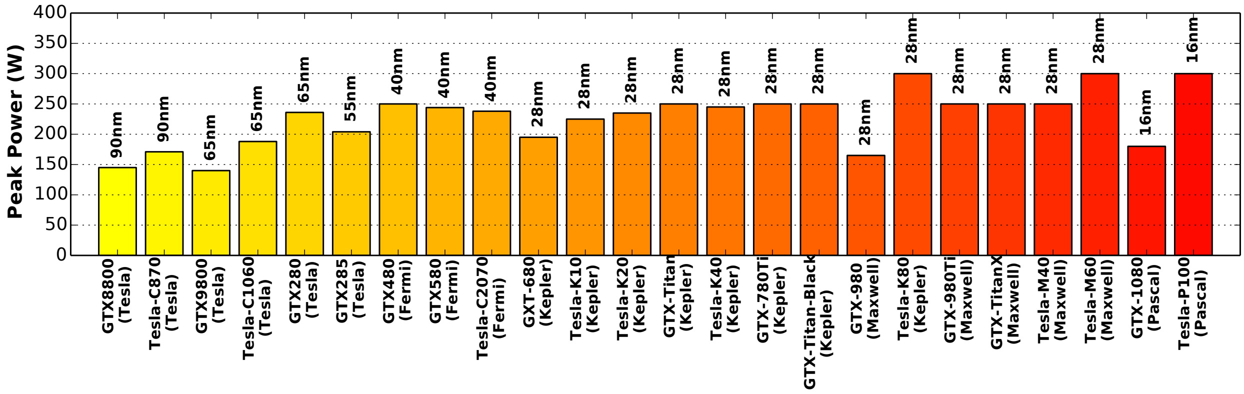

Figure 1.7: The Manufacturing Process and Peak Power Consumption for NVIDIA GTX and Tesla Flagship GPUs.

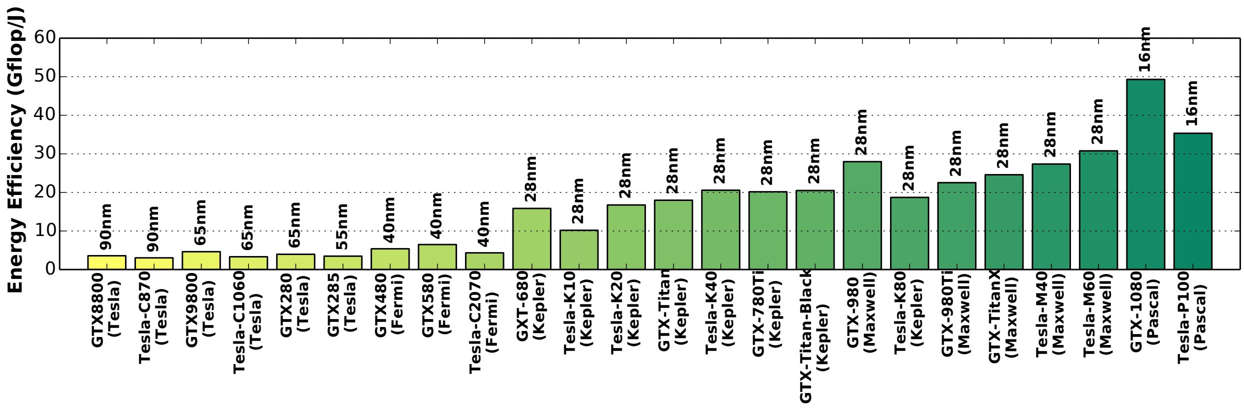

Figure 1.8: The Manufacturing Process and Energy Efficiency (with peak power consumption) for NVIDIA GTX and Tesla Flagship GPUs.

Emerging Applications: Today, the increased programmability of GPGPUs makes most contemporary applications relatively easy to migrate on GPUs. However, efficiently implementing irregular applications, especially the graph-related algorithms from big-data applications, still remains a difficult task. Therefore, the strategies to efficiently implement irregular algorithms on GPUs and the hardware designs to optimize GPU architectures for irregular routines/data-structures persistently remain hot research topics for GPGPU [65, 66, 67, 68]. Another type of emerging application domain is machine learning, especially the deep-learning [69, 70, 71]. In fact, the latest Pascal GPU P100 is specially designed for deep-learning utilizations (FP16 support, HBM2 memory, NVLink, large register file, large L2 cache as well as the specially-designed DGX-1 system for deep-learning [52]).

Resilience Related: There are three topics about resilience-related issues on GPUs: *approximate computing*, *fault-tolerance* [72, 73, 74] and *reliability* [75, 76, 77]. Specially, under the pressure of continuous performance scaling and power control, and given the inherent fault-tolerant properties of the emerging applications (e.g., big-data, multimedia and machine-learning), approximate computing quickly becomes an emerging and promising technique for GPGPU. This is one of the most rapidly developing areas for GPU research [78, 79, 80, 81, 82, 83].

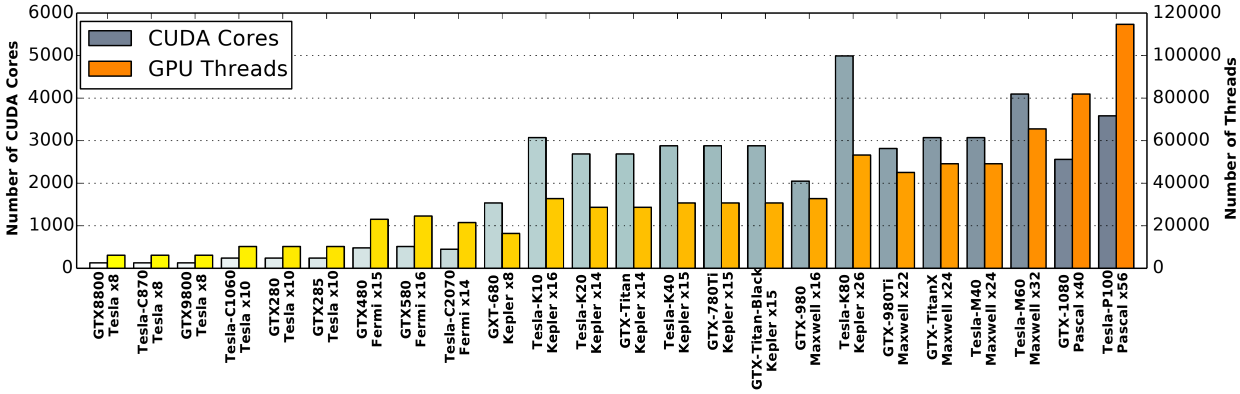

Figure 1.9: The Number of Cores and Maximum Number of Active Threads for NVIDIA GTX and Tesla Flagship GPU Products in the past decade. Note, the core number uses the left Y-axis while the thread number uses the right Y-axis. As can be seen, after ten years exponential scaling, today the number of cores has reached as many as 5,000 while the volume of resident threads is nearly 120,000 in a single GPU card! Such a "*thousands-of-cores while hundreds of thousands of threads in a card*" situation is never imaginable in any conventional CPU contexts.

1.3 Research Problems

Although GPGPU performance is scaling very fast, the ever-growing performance demand from emerging GPGPU applications, such as large-scale machine learning, oil-and-gas exploration, scientific simulations, big data, 3D reconstruction and computational financing, etc. still enforces great pressure on the scaling of GPGPU performance. This thesis focuses on the performance issues and attempts to answer the following two fundamental questions:

- ***How to understand GPU performance?***
- ***How to optimize GPU performance?***

In particular, we are concentrating on the following research problems:

New Performance Analytic Model: Traditional analytic models are predominately used for theoretical performance prediction and architecture design evaluations. However, both software designers and hardware architects can easily lose themselves in the ever-growing and over-detailed design space for modern highly parallel computer platforms, such as GPUs. With a completely divergent hardware design principle (e.g., throughput-oriented), the community urgently requires a novel high-level analytic tool to understand the performance shown on these platforms and answer questions, such as: How to understand the parallelism in GPU hardware and software? Given an application, what are the possible performance bottlenecks? Why such a bottleneck appears there? What kind of optimizations can be applied to mitigate or even eliminate the bottlenecks? How much performance gain can be anticipated if a specific hardware/software optimization is applied?

New Performance Optimization Approaches: As GPU is a highly parallel platform with massive numbers of cores and threads (see Figure 1.9); in addition, given the completely different performance evaluation metrics (e.g., processor occupancy), the GPU architecture and the way to program a GPU are significantly divergent from CPU. Therefore, conventional CPU-targeted optimization strategies (i.e., latency-oriented such as cache prefetching & bypassing, out-of-order execution, etc), either software or hardware, are mostly out-of-date and no longer applicable. Meanwhile, the GPU has its own performance considerations (e.g., memory coalescing, bank conflicts, divergence, etc). The community requires new optimization approaches for continuously performance improvement for GPUs, especially those that are immediately deployable and transparent to the user and the hardware.

1.4 Thesis Contributions

This thesis makes the following contributions to address the problems proposed in Section 1.3. We first discuss contributions of each chapter and then clarify their intercorrelations.

1.4.1 Chapter Contributions

For each chapter, this thesis makes the following contributions:

- In Chapter 3, we propose a novel **visualizable analytic model called "X"** specially for today's highly parallel machines. It comprehensively analyzes the interaction between the four types of parallelism (i.e., thread-level-parallelism, instruction-level-parallelism, data-level-parallelism and memory-level-parallelism) and two types of memory effects (local on-chip cache effect and remote off-chip memory effect), in terms of system throughput. The X-model acts as the theoretical basis of this thesis [84, 85].

- In Chapter 4, we propose an effective autotuning approach to resolve the conflict between overall thread concurrency and per-thread register usage for GPUs. We discover that the performance impact from register usage is almost continuous, but from concurrency is discrete. Their joint-effects form a special relationship such that a series of critical-points can be precomputed. These critical-points denote the best performance for each concurrency level. Therefore, the global optimum, which refers to the optimal number of registers per-thread, can be quickly and efficiently selected to deliver the best GPU performance [86].

- In Chapter 5, we propose an adaptive cache bypassing framework for GPUs. It uses a simple but effective approach to throttle the number of threads that could access the three types of GPU caches – L1, L2 and read-only caches, thereby avoiding the fierce cache thrashing of GPUs, and significantly improving the performance for cache-sensitive applications [87].

- In Chapter 6, we focus on a crucial GPU component that has long been ignored — the Special Function Units (SFUs) and show its outstanding role in performance acceleration and approximate computing for GPU applications. We exhaustively evaluate the numeric transcendental functions that are accelerated by SFUs and propose a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. It partitions the active threads into a PE-based slower but accurate path, and a SFU-based faster but approximated path, and tunes the relative partition ratio among two paths to control the tradeoffs between the performance and accuracy of the GPU kernels. In this way, a fine-grained and almost linear tuning space for the tradeoff between performance and accuracy can be created [82].

- In Chapter 7, we propose a novel approach for fine-grained inter-thread synchronizations on the shared memory of modern GPUs. By reassembling the low-level assembly-based micro-operations that comprise an atomic instruction, we develop a highly efficient, low cost lock approach that can be leveraged to set up a fine-grained producer-consumer synchronization channel between cooperative threads in a thread block. Additionally, we show how to implement a dataflow algorithm on GPUs using a real 2D-wavefront application [88].

1.4.2 Chapter Intercorrelation

The main-context chapters (Chapter 3 to 7) are interconnected and unified in the following way:

- **Theoretical Basis:** The X-model acts as the theoretical basis of this thesis. It is used to analyze the underlying tradeoffs between concurrency and registers in Chapter 4 and between memory-level-parallelism and cache-performance in Chapter 5.

- **Bico-scheduling Design Paradigm:** We generate a new design paradigm specially for GPGPUs. It is based on the unique SIMT execution model of GPUs. The SIMT has two typical features: single instruction-stream (SI) and multiple-threads (MT), which essentially enables a novel design paradigm for GPU architecture design and performance tuning, labeled as **"bico-scheduling"** (short for *binary co-scheduling*). Such a paradigm is motivated from the observation that when a new function module is integrated into GPUs for acceleration purposes (e.g., an on-chip cache, a special-function-unit), the excessive parallel GPU threads often flood the module and lead to fierce resource contention, which limits the performance. The bico-scheduling here introduces a fine-grained performance tuning space so that the large amount of GPU threads are separated into dual groups targeting two paths: one for the accelerator module as a fast path, one for the original path as a slow path (e.g., one thread-group buffers in the on-chip cache, the other thread-group bypasses the cache). In addition, a runtime-tunable threshold is introduced to control the partition degree for the two groups, so as to reach a good balance between parallelism and the utilization of the accelerator module (i.e., bico-scheduling among fast path and slow path). Such a design paradigm is only for GPU as the feature of SI creates a monotonic tuning space (threads are identical) while the MT feature enables a very fine-grained, incrementally changed tuning space, both are non-existent in conventional processors. It thus leads to many novel optimization opportunities for GPUs, such as the one for caches in Chapter 5 and the one for SFUs in Chapter 6. It is also possible to apply this paradigm upon other on-chip modules, such as NoC, lock-bit, registers, etc.

- **Purely Software Design:** All the proposed optimization techniques are transparent, tractable and portable. Here *transparent* means no modification effort is required from either application-developers or hardware-architects — all designs are purely software-based so that performance can be demonstrated immediately on commodity GPUs rather than GPU simulators. *Tractable* means the designs are intuitive to understand while straightforward to implement, it should bring immediate benefit. *Portable* means the approach works for various GPU platforms.

- **Unification:** The proposed optimization approaches are unified by focusing on alternative function modules inside a GPU streaming processor: register-files (Chapter 4), caches (Chapter 5), compute units (Chapter 6) and shared memory (Chapter 7). They differentiate each other by targeting different design tradeoffs: per-thread performance vs. parallelism for register files in Chapter 4, per-thread cache performance vs. overall cache performance for caches in Chapter 5, compute performance vs. compute accuracy for compute units in Chapter 6 and shared memory performance vs. programmability for shared memory in Chapter 7.

- **Thread Interaction:** Finally, we resolve the two types of performance degradation due to thread interaction for parallel execution: one is thread sharing/thrashing in the shared-cache; the other is thread cooperation/contention in the critical section. For the sharing/thrashing effect in shared caches of GPUs, we propose cache prefetching/bypassing to attain the best cache performance in Chapter 5. For the thread cooperation/contention problem in the critical section of GPU program, we propose lock-bit based fine-grained synchronization method to speed up thread cooperation in GPU shared memory in Chapter 7.

1.5 Thesis Structure

The remaining of the thesis is organized as follows: Chapter 2 introduces the background knowledge about GPUs: the machine model, the execution model, the programming model and the evaluation model. Chapter 3 presents the X-Model for parallel machines. Chapter 4 discusses the register file optimization technique for GPUs. Chapter 5 talks about the cache optimization technique for GPUs. Chapter 6 describes the compute units optimization technique for GPUs. Chapter 7 shows the shared memory optimization technique for GPUs. Finally, Chapter 8 summarizes the thesis and discusses potential future works.

CHAPTER 2

Background

To make an easier description of the GPGPU analytic model and optimization techniques in the next several chapters, we describe some background knowledge about modern GPGPUs in this chapter. To show readers a complete and comprehensive view about GPGPU, we describe it from four aspects: *GPU Machine Model* (i.e, architecture), *GPU Execution Model* (i.e., thread hierarchy and mapping to hardware), *GPU Programming Model* (i.e., kernel configuration and compilation) and *GPU Evaluation Model* (i.e., simulators, benchmarks and profiling tools).

2.1 GPU Machine Model – The SM-Centric Architecture

A GPU is composed of multiple *streaming-multiprocessors* (SMs), sharing an L2 cache and DRAM controllers via a crossbar interconnection network (NoC). The SMs are the central parts of the GPU architecture, which perform all the vertex/geometry/pixel-fragment shader-programs and GPGPU-programs. As shown in Figure 2.1, an SM features a number of *scalar processor* cores (SPs) and two other types of function-units — the *Double-Precision Units* (DPUs) for double-precision (DP) floating-point calculations and the *Special-Function Units* (SFUs) for processing transcendental functions and texture-fetching interpolations. Other components, such as the *register files* (RFs), *load-store units* (LSUs), *scratchpad memory* (i.e., shared memory), and various caches (i.e., *instruction cache*, *constant cache*, *texture/read-only cache*, *L1 cache*) for on-chip data caching also reside in the SMs.

2.1.1 Function-Units

This subsection introduces the four function-units inside an SM: *SP, SFU, DPU* and *LSU*.

Scalar-Processor (SP): The scalar-processors, known as the CUDA cores, are the primary basic processors in an SM, performing the fundamental integer, floating-point, comparison and type-conversion operations. Each SP contains a single-precision floating-point unit (FPU) and an integer arithmetic/logic unit (ALU) — both units are fully pipelined.

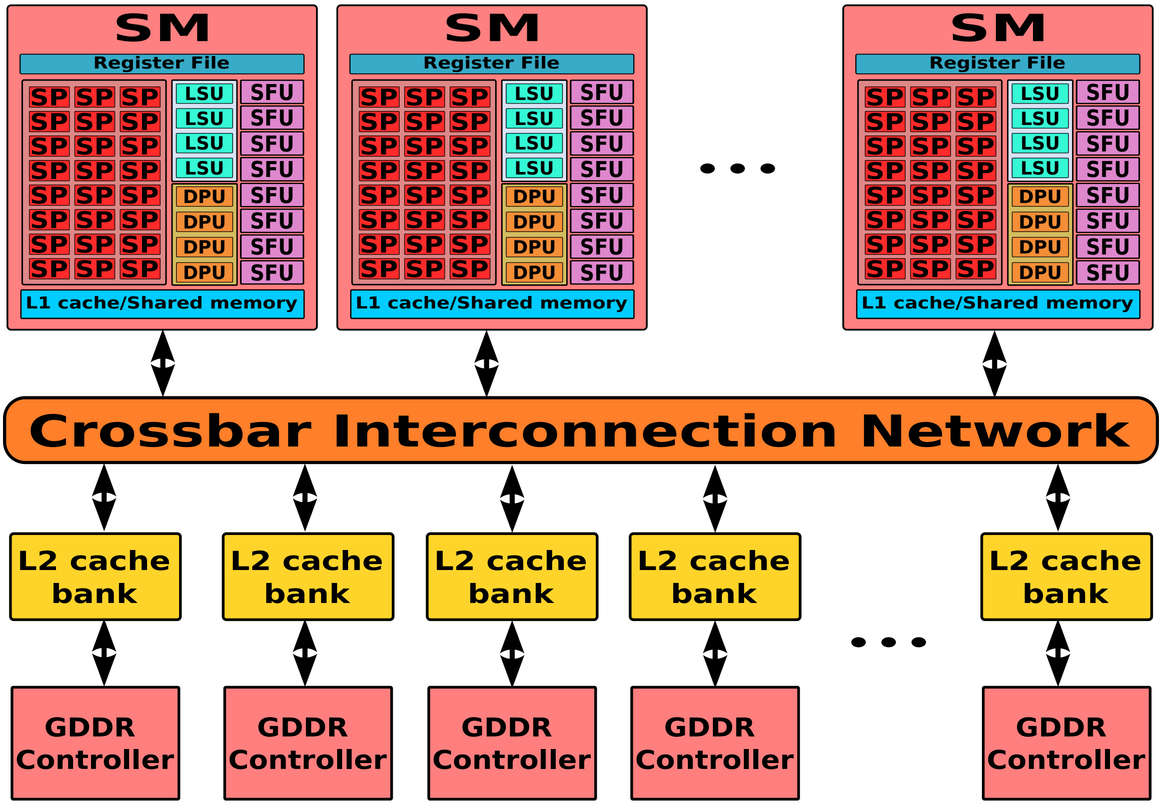

Figure 2.1: General Architecture for Modern GPUs.

Special-Function-Unit (SFU): The SFUs are integrated for fast transcendental function calculations (e.g., sine, cosine, reciprocal, square-root, etc.) and planar attribute interpolations. Each SFU also features four floating-point multipliers that can offer extra FP throughput in addition to SPs. The SFU pipelines are independent from the SP pipelines. We thoroughly evaluate the characteristics of SFUs in Chapter 6.

Double-Precision-Unit (DPU): The DPUs are the units specially for double-precision (DP) computations. They perform fused multiply-add (FMA) DP operations in highly efficient deep pipelines. The number of DPUs in an SM dictates the DP performance of a GPU device, e.g., the Maxwell GPUs have only 4 DPUs in the SMs, delivering only 1/32 DP performance compared to their SP performance. We exploit DPUs in Chapter 6.

Load-Store-Unit (LSU): As indicated by the name, the load-store units are used to fetch and save data to memory. They contain dedicated computing units to rapidly calculate the source and destination addresses for the initiated memory requests.

2.1.2 Device Memories

We discuss the various types of memories in a GPU, including *register files, shared-memory, local-memory, global-memory, constant memory* and *texture memory* in this subsection. Their basic features are summarized in Table 2.1.

Table 2.1: GPU Device Memory Features

Memory	On/Off Chip	Cached	Access	Scope	Lifetime
Register Files	On	N/A	Read/Write	Per-thread	Thread
Local Memory	Off	L1/L2	Read/Write	Per-thread	Thread
Shared Memory	On	N/A	Read/Write	Thread Block (CTA)	Thread Block (CTA)
Global Memory	Off	L1/L2	Read/Write	GPU+CPU	Host Allocation
Constant Memory	Off	Constant cache	Read Only	GPU+CPU	Host Allocation
Texture Memory	Off	Texture cache	Read Only	GPU+CPU	Host Allocation

Register Files (RF): GPUs overall have very large volume of registers. Due to the large size, GPU registers are implemented by SRAMs, which are partitioned into banks for throughput concern. Therefore, compared to CPU registers, the GPU registers experience long access latency and may suffer from potential bank conflicts [89]. We discuss how to exploit GPU registers in Chapter 4.

Local Memory (LM): The local memory is not a physical memory space but rather a portion of the global memory (see below). Its scope is thread-private, the same as RFs (see Table 2.1). It is generally used for temporal spilling when there are insufficient registers to hold all the required variables (i.e., register spilling), or when the arrays are declared in the kernel but the compiler cannot decide the exact indexing to reference them. The local memory is cached by L1 and L2 in Fermi and Kepler, but is only cached by L2 in Maxwell and Pascal. Register spilling in local memory hurts the performance as it introduces extra instructions and memory traffic, especially when there is a cache miss (so the register value has to be fetched from off-chip global memory). We evaluate the impact of local cache in Chapter 4.

Shared Memory (HM): The shared memory or *scratchpad memory* is an on-chip storage shared among all units inside an SM. It serves as a communication interface for fast data exchanging between different threads of a thread block (i.e., Cooperative-Thread-Array or CTA, see Section 2.2.1). Being on-chip, the shared memory has much higher bandwidth and shorter accessing latency compared to the local memory or global memory. Therefore, optimizations which can shift global/local memory access to shared memory are highly recommended by the CUDA programming guide [53]. To achieve higher bandwidth, the shared memory is partitioned into banks, thus can be accessed in parallel (similar to register files and L2 cache). However, in case two addresses from the same memory request fall in the same bank, a *bank conflict* occurs and the accesses have to be serialized, which seriously degrades the performance of the shared memory. We discuss optimization techniques regarding shared memory in Chapter 7.

Global Memory (GM): The global memory is the *device memory*, also known as GPU off-chip memory or GPU main-memory. It is the most frequently-used memory for GPUs such that its throughput in many conditions (i.e., memory-bound applications) determines the final achievable performance of GPUs. The attainable global memory throughput, or sustainable throughput [90], is mainly constrained by two factors: *raw memory bandwidth* and *coalescing degree*. (1) The raw memory bandwidth is limited by the pin number, wire length and the physical property of DRAM; therefore it is increasing slowly since Kepler (see Figure 1.5 in Chapter 1). However, such a stagnant situation is completely changed by the 3D-stack memory technique recently applied in Pascal [52]. (2)

To gain from transmitting large data blocks at a time, a technique known as *memory access coalescing* is applied. The LSUs initially calculate the target addresses of each warp lane individually. Before memory fetching, a special Address-Coalescing hardware [91] will check whether the addresses from the same warp are continuously distributed (which is the common case for global memory access). It then notifies the Memory-Interface-Units for one or multiple aggregated block transfers from the cache or global memory [91]. The CUDA programming guide provides detailed discussion about the identification of memory coalescing [10].

Constant Memory (CM) / Constant Cache: The constant memory is used to store data that does not change during the kernel execution. It is 64KB for all GPU generations and is off-chip. Similar to local memory, it is a special part of the global memory. However, the constant memory is not cached by L1/L2 but an individual cache known as constant cache. The 8KB/10KB constant cache in each SM is specially designed so that the data of a single memory address can be broadcast to all threads across the warp at a time. However, when different addresses are requested from a warp, the accessing request has to be split into as many requests as the number of different addresses.

Texture Memory (TM) / Texture Cache: The texture memory or *surface memory* also resides in the global memory. It is buffered by a texture cache so that texture fetches or surface read are performed only when there is a cache miss. The texture cache is specially optimized for 2D spatial locality. Therefore, threads from a warp can gain extra performance when they access nearby addresses in 2D space [92]. Besides, the addresses of texture memory are calculated by dedicated units outside the kernel [93], thus gaining extra compute capacity. In addition, the packed (image) data (if applicable) can be unpacked and broadcast to multiple variables in a single operation [93]. As the texture cache is designed for streaming fetches with fixed latency, a cache hit reduces off-chip memory throughput demand but not the fetching latency [53].

Prior to Maxwell, the texture cache was only utilized for texture memory. However, from Maxwell onwards, the previous L1 cache, which shared the same physical storage with the on-chip shared memory in an SM, has been discarded. On the other hand, the texture cache was firstly marked as *read-only* (or non-coherent) cache [49] and later labeled as the L1 cache in the CUDA official documents [50, 51, 52]. It is claimed that the texture cache (i.e., read-only cache) has higher tag bandwidth thus supporting full speed unaligned memory access patterns [49].

2.1.3 Device Caches

We have already discussed constant cache and texture cache. Now we introduce the *L1 Instruction Cache*, *L1 data cache* and *L2 data cache*.

L1 Instruction-Cache: There are very few documents or literature available discussing about GPU instruction cache, specially for new GPU architectures. One may refer to [89] for analysis on the old Tesla architecture GT200 GPUs. In addition, [94] discussed instruction cache thrashing when implementing warp-based synchronization schemes on Fermi GPUs.

L1 Data-Cache: The L1 data cache [1] for GPU was firstly introduced in Fermi. The SM-private L1 cache shares the same on-chip storage with the shared memory of an SM. Their relative sizes are reconfigurable (16/48 or 48/16 KB in Fermi and 16/48, 32/32 or 48/16 KB in Kepler). The L1 cache-line is 128B. It caches both global memory read and local memory access (read and write) and is non-coherent [46]. The local memory is generally utilized for register spilling, function calls and automatic variables [53]. The L1 cache is read-only when caching access to global memory, but is writable when caching access to local memory. As discussed, from Maxwell, the traditional L1 cache is unified with texture cache.

L2 Cache: The unified L2 cache is also firstly introduced in Fermi. It services all types of memory access (i.e., global, local, constant and texture) and is coherent with the host CPU memory. The L2 cache is read/writable and adopts write-back replacement policy [46]. It is the primary point for data unification [49] and is a good place for data sharing across SMs. The L2 cache is generally partitioned into banks, each of them acting as a buffer for a way of off-chip memory channel (GDDR or HBM2-DRAM), so as to significantly reduce the ultimate memory bandwidth demand.

2.1.4 NoC and ROP

We briefly discuss the NoC and ROP in this subsection to make the description complete, although they are not relevant to the main topics of this thesis.

Interconnection Network (NoC): The interconnection network among SMs and L2 banks is a crossbar network. It allows simultaneous communication between multiple SMs and L2 banks, thus offering considerable NoC throughput. As introduced in [95], a typical crossbar NoC encapsulates an address bus and two data buses. The address bus is unidirectional from SMs to L2 banks; whereas the two data buses form a bidirectional channel between SMs and L2 banks. Here, the communication are point-to-point [96]. A memory-request queue (MRQ) and a a bank load queue (BLQ) is attached to each SM and L2 bank, respectively. When a load request is generated from the LSUs inside an SM, it will first cache in the local MRQ and then be delivered to the destination BLQ through the crossbar NoC. After some waiting time in BLQ, the request will be processed by the L2 banks. It is already known that the crossbar network comes at a high switching cost for the simultaneously connections. Particularly, when the accessing requests are random and messy, interference will appear, which leads to the reduction of effective bandwidth [97].

Raster Operation Processor (ROP): The fixed-function ROP is to perform color and depth frame buffer operations directly on memory. It also services the external memory load, store and atomic accesses.

Finally, we have summarized the architecture configurations for each generation of NVIDIA GPUs, as shown in Table 2.2. This is done for the ease of future references.

[1] In this thesis, without special indication, instruction cache specially refers to L1 instruction cache while L1 means L1 data cache.

Table 2.2: GPU SM Architecture. "CC." stands for Compute Capability [53].

Arch.	CC.	Representative GPU	SMs	RF	SP	SFU	DPU	LSU	Shared Mem	Const	Texture	L1	L2
Tesla	1.0	Tesla-C870	8	8KB	8	2	N/A	N/A	16KB	8KB	12KB	N/A	N/A
Tesla	1.3	Tesla-C1060	10	16KB	8	2	N/A	N/A	16KB	8KB	12KB	N/A	N/A
Fermi	2.0	Tesla-C2070	16	32KB	32	4	16	16	16/48KB	8KB	12KB	16/48KB	768KB
Fermi	2.1	GTX-460	16	32KB	48	8	4	16	16/48KB	8KB	12KB	16/48KB	512KB
Kepler	3.0	Tesla-K10	8	64KB	192	32	8	32	16/32/48KB	8KB	48KB	16/32/48KB	512KB
Kepler	3.5	Tesla-K40	15	64KB	192	32	64	32	16/32/48KB	8KB	48KB	16/32/48KB	1536KB
Kepler	3.7	Tesla-K80(x2)	13x2	128KB	192	32	64	32	112KB	8KB	48KB	16KB	1536KB
Maxwell	5.0	GTX-750Ti	5	64KB	128	32	4	32	64KB	10KB	24KB	N/A	2048KB
Maxwell	5.2	Tesla-M40	24	64KB	128	32	4	32	96KB	10KB	48KB	N/A	2048KB
Pascal	6.0	Tesla-P100	60	64KB	64	16	32	16	64KB	10KB	48KB	N/A	4096KB

2.2 GPU Execution Model – Massive SIMT and Thread Mapping

We introduce the SIMT execution model and the thread hierarchy mapping of GPUs in this subsection. These are basements for further discussions of this thesis.

2.2.1 SIMT Execution Model

Evolved from SIMD, the execution model of GPUs is known as *single-instruction-multiple-threads* or **SIMT** [45, 53]. A kernel, which is a function that runs on the GPU part of the processing system (CPU+GPU), includes thousands of simultaneous lightweighted GPU threads that are primarily grouped into multiple *thread blocks* or *Cooperative-Thread-Arrays* (**CTAs**). When a kernel is launched, its CTAs are dispatched to the SMs. It is possible that several CTAs are dispatched to the same SM, depending on the available SM on-chip resources, such as the registers and shared memory. These resources are evenly divided among the concurrent CTAs of an SM.

Threads inside a CTA are further organized as a number of execution groups that perform the same operations on different data in a lockstep manner. Such execution groups are called **warps**. In an SM, a warp is the basic unit in terms of scheduling, executing and accessing cache/memory. If threads in a warp diverge at a point (e.g., upon *if-else*), all the branches will be executed alternatively and sequentially, with threads not belonging to the present branch being masked off, until divergent threads consolidate at a convergent point and continue the lockstep execution. Such a divergence (called *warp divergence*) incurs enormous overhead [98]. We deeply discuss such overhead and warp divergence issue in Chapter 7. Meanwhile, if a warp is obstructed by a long latency operation, e.g., off-chip global memory read, the warp scheduler will fetch-in another ready warp instantly with little cost [53]. How to establish an orchestrated warp scheduling for good execution overlapping or latency hiding, especially considering the positive/negative impact on the memory system, has recently become a hot research topic [99, 100, 101, 102].

GPU supports multi-issuing and multi-dispatching. During execution, the dual- or quad-warp schedulers select two or four ready warps (with up-to two independent instructions per warp [49]) to dispatch onto the different function units (e.g., SPs, SFUs). Although most instructions are accomplished by SPs, the DPUs and SFUs offer extra processing bandwidth when processing special

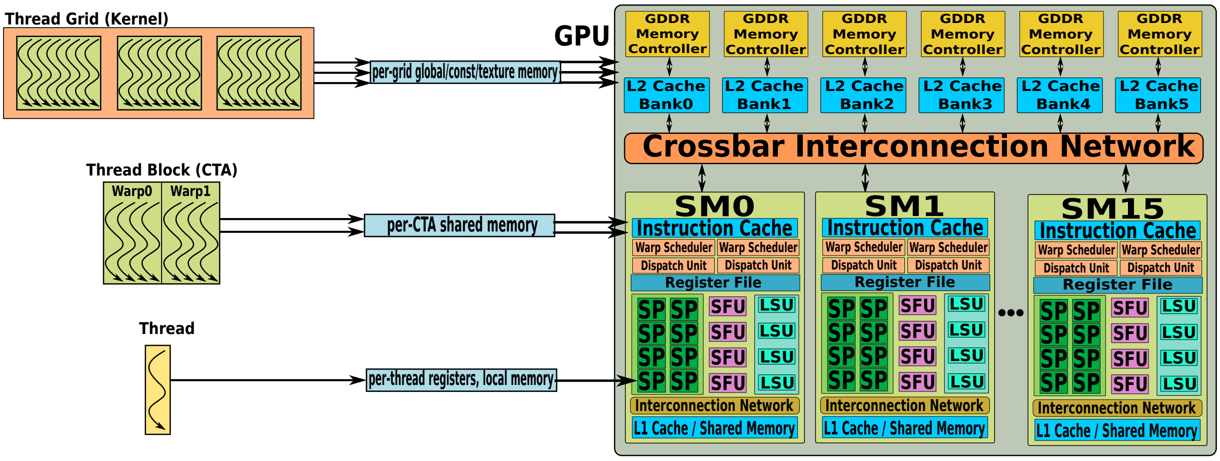

Figure 2.2: GPU Thread Hierarchy Mapping to Architecture

functions (e.g., transcendental functions) or double precision data. These special units are useful, but it is often challenging to leverage them in a balanced way. This is the reason why multi-issuing/dispatching mixed instructions to these function units remains critical for GPU performance delivery [103, 104].

2.2.2 Thread Hierarchy Mapping

Figure 2.2 summarizes the mapping from CUDA thread hierarchy to GPU architecture discussed in Section 2.1. As can be seen, (1) the thread instruction is mapped to a SP or SFU or DPU (in a unit of warp); (2) the thread blocks or CTAs are mapped to the SMs; (3) the thread grid is mapped to the GPU device. We also show the scope of memory introduced in Section 2.1 in the figure. The global memory, constant memory and texture memory are shared among all threads in a grid, while accessing the shared memory is only possible for threads within the same CTA. The register files and local memory are private to a thread.

2.3 GPU Programming Model: Configuration and Compilation

We introduce the GPU programming model, particularly how to configure a kernel function and how it is compiled in this subsection.

2.3.1 Kernel Configuration

CUDA extends C/C++ by allowing programmers to define *kernel functions*. As already discussed, the kernel is the function that runs on the GPU side by massive parallel GPU threads. The way to specify the number of threads to execute the kernel is via the <<<...>>> configuration syntax. As shown in Listing 2.2 which is a simple element-to-element multiplication for 2D matrices, <<<*Grid_config, CTA_config* >>> implies that a kernel has a grid configuration defined by *Grid_config* and a CTA

```
//2D vector multiplication
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      C[i][j]+= A[i][j]*B[i][j];
```

Listing 2.1: CPU Loop Nest

```
__global__ void VM2D(A,B,C){
  int x=blockIdx.x*blockDim.x+threadIdx.x;
  int y=blockIdx.y*blockDim.y+threadIdx.y;
  C[x][y]+=A[x][y]*B[x][y];
}
  VM2D <<<Grid_config,CTA_config>>>(A,B,C);
```

Listing 2.2: GPU Kernel and CTA

Table 2.3: GPU Thread Limit

Arch.	CC.	Grids/GPU	CTAs/Grid	Thds/CTA	CTAs/SM	Thds/SM	Thds/Warp	Warps/CTA	Warps/SM
Tesla	1.0	1	(512,512,64)	512	8	768	32	16	24
Tesla	1.1	1	(512,512,64)	512	8	768	32	16	24
Tesla	1.2	1	(512,512,64)	512	8	1024	32	16	32
Tesla	1.3	1	(512,512,64)	512	8	1024	32	16	32
Fermi	2.0	16	$(2^{16},2^{16},2^{16})$	1024	8	1536	32	32	48
Fermi	2.1	16	$(2^{16},2^{16},2^{16})$	1024	8	1536	32	32	48
Kepler	3.0	16	$(2^{31}-1,2^{16},2^{16})$	1024	16	2048	32	32	64
Kepler	3.2	4	$(2^{31}-1,2^{16},2^{16})$	1024	16	2048	32	32	64
Kepler	3.5	32	$(2^{31}-1,2^{16},2^{16})$	1024	16	2048	32	32	64
Kepler	3.7	32	$(2^{31}-1,2^{16},2^{16})$	1024	16	2048	32	32	64
Maxwell	5.0	32	$(2^{31}-1,2^{16},2^{16})$	1024	32	2048	32	32	64
Maxwell	5.2	32	$(2^{31}-1,2^{16},2^{16})$	1024	32	2048	32	32	64
Maxwell	5.3	16	$(2^{31}-1,2^{16},2^{16})$	1024	32	2048	32	32	64
Pascal	6.0	32	$(2^{31}-1,2^{16},2^{16})$	1024	32	2048	32	32	64

configuration defined by *CTA_config*. Both *Grid_config* and *CTA_config* can be 1D, 2D or 3D. For example, if *Grid_config* is $(1,2,3)$, it means the thread grid is defined as a 3D CTA grid consisting of $1\times2\times3=6$ CTAs. Analogously, if *CTA_config* is $(4,5,6)$, it means the CTA is defined as a 3D CTA consisting $4\times5\times6=120$ threads. Both *Grid_config* and *CTA_config* have constraints, as listed in Table 2.3. The kernel configuration is not only vital for implementing application algorithms, but is also crucial for GPU performance since the GPU execution resources are usually limited — there is a constant tradeoff between thread volume and per-thread resource share.

Meanwhile, each thread involved in the execution of the kernel is assigned with a unique thread ID, which can be acquired during execution by fetching the built-in register *threadIdx*. Similarly, each CTA is given a unique CTA ID, which can be acquired by fetching *blockIdx* (see Listing 2.2). Threads in a CTA can communicate with each other via the shared memory or synchronize their execution using the CTA-scope synchronization primitive "*__syncthreads()*". However, the execution of CTAs must be independent. They can be scheduled or executed in any order, in parallel or in series, without affecting the final correctness. The detailed discussion about the kernel configuration can be found in the CUDA programming guide [53].

2.3.2 Compilation Trajectory

There are two workflows to compile the CUDA kernels: *offline compilation* and *just-in-time compilation*.

Offline Compilation: As shown in Figure 2.3-(A), the source file is compiled into PTX assembly code first. *PTX* stands for *Parallel-Thread-Execution*, which is an intermediate-level thread execution

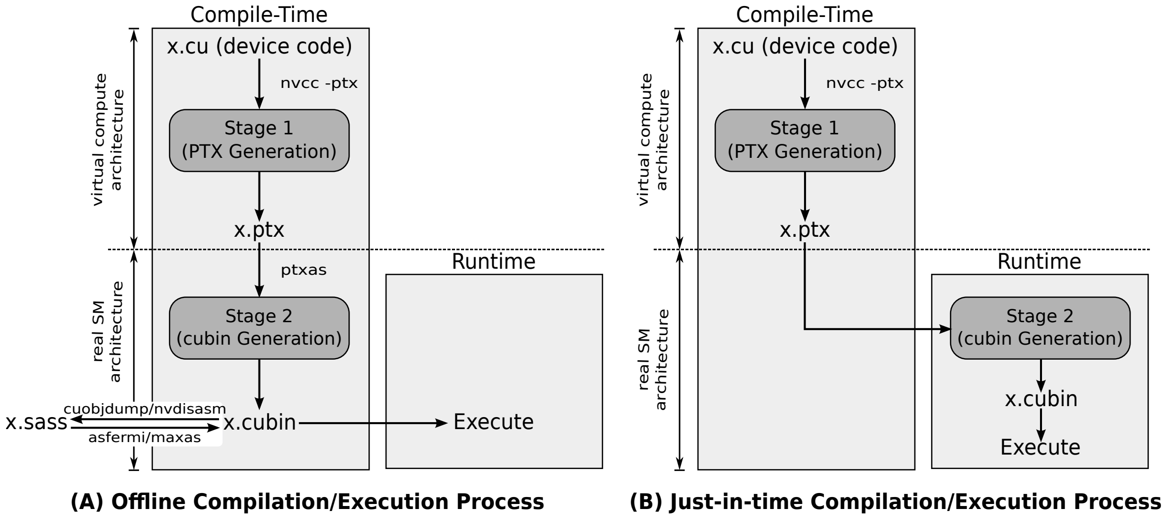

Figure 2.3: GPU Kernel Code Compile and Execution Trajectory

virtual machine and instruction set architecture (ISA) that offers inter-GPU program portability. The PTX instructions are compatible with all later GPUs or CUDA Runtime. Then, a PTX program is assembled into the cubin binary, which is an object file that can be linked by the host compiler (e.g., gcc, icc). The cubin object is architecture specific. It is only compatible with later GPUs of the same architecture generation. For example, a cubin object generated by CC-3.0 compiler can be executed on Tesla-K10 (CC-3.0), K40 (CC-3.5) or K80 (CC-3.7), but is not executable on Fermi (CC-2.x), Maxwell (CC-5.x) or Pascal (CC-6.x) [53].

Just-in-Time (JIT) Compilation: As shown in Figure 2.3-(B), instead of assembling to cubin at compile-time, the GPU device driver can assemble the PTX to cubin on-the-fly at runtime, known as *just-in-time compilation*. JIT introduces extra loading overhead, but offers the assembler/runtime/hardware portability. A GPU program compiled by an old version compiler can thus benefit from the improvements of *ptxas* and the updated GPU hardware [53, 105].

Shader-Assembly (SASS): In fact, the cubin binary can be dumped by *cuobjdump* [106] to another format of assembly code, called *Shader-Assembly* (SASS), which is a machine-dependent, human-readable low-level assembly. Modifying SASS code requires deep knowledge about the hardware implementation details that are often concealed by NVIDIA, thus is very difficult. Meanwhile, migrating SASS programs to another GPU is also very difficult as it is hardware dependent. More importantly, although a dumping tool (from cubin to SASS) is offered, there is no official SASS assembler (from SASS to cubin) available, as *ptxas* is not open-sourced. For the Fermi architecture, there is a homemade SASS assembler called *asfermi* [107]. For Maxwell, a similar one is called *maxas*. However, one has to handle the instruction scheduling issues manually by using the *maxas* assembler on Maxwell, which is quite complicated and difficult. SASS programming is further discussed in Chapter 7.

2.4 GPU Evaluation Model: Simulators, Benchmarks and Profiling

Finally, we give a brief introduction about the simulator and the benchmarks that are commonly used for GPGPU validations. We also introduce the profiling tools for evaluating real GPU hardware in this subsection.

2.4.1 Simulators

The most well-known and widely accepted GPGPU simulator is *GPGPU-Sim* [43]. Today, almost all proposed hardware designs for GPUs in academia are validated in GPGPU-Sim. However, before the dominance of GPGPU-Sim, there were other alternatives, such as *Barra* [108] and *Ocelot* [109]. *Barra* is an SASS-level functional simulator designed for NVIDIA G80 GPUs. *Ocelot* in its backend integrates a PTX emulator. *Ocelot* later evolved into a dynamic JIT compilation framework for GPUs. People also use it for instrumenting [110] and memory trace dumping [111]. However, both *Barra* and *Ocelot* are not actively maintained right now. Besides, the very old version CUDA Runtime (CC-1.x) once included an "official" simulator for machines without a CUDA-capable GPU to run CUDA program, which was soon discarded before Fermi.

Although GPGPU-Sim still remains widely adopted, it is now a bit out-of-date as it only supports the very old Fermi architecture; GPGPU is a fast developing domain, since Fermi, three GPU architecture generations have been published: Kepler, Maxwell and Pascal. However, the development of simulators is far lagging behind the development of the hardware, as few technical details have ever been published by the vendors while in the meantime GPUs have become increasingly complicated. Recently, an open-source, RTL-level GPU SM implementation has been announced, known as *MIAOW* [112, 113]. However, few utilizations have been reported based on *MIAOW* up till now.

2.4.2 Benchmarks

The benchmarks frequently used for evaluating software/hardware GPU designs are: *Rodinia*, *Parboil*, *Shoc*, *Polybench*, *Mars*, *LonestarGPU*, *CUDA-SDK* and *GPGPU-sim*. All the applications evaluated in this thesis are taken from these benchmark suites.

Rodinia [37] is the most widely-used GPU benchmark that contains applications from various domains. Their basic features are summarized in Table 2.4. A detailed characterization about Rodinia can be found in [114].

Parboil [38] is a GPU benchmark suite emphasizing on throughput-oriented streaming-applications. For each application included in Parboil, there is a naive CUDA implementation and an optimized implementation. Information about Parboil benchmark are summarized in Table 2.5.

Shoc [39] is developed for measuring performance and stability of coprocessor based systems, such as GPUs, Xeon-Phi, etc. The information is summarized in Table 2.6.

Table 2.4: Rodinia Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	OpenMP
`backprop`	*Perceptron back propagation*	Neural Network	Yes	Yes	Yes
`bfs`	*Breadth first search*	Graph Algorithm	Yes	Yes	Yes
`b+tree`	*B+tree Operation*	Searching	Yes	Yes	Yes
`leukocyte`	*Detect leukocytes in blood vessel video*	Medical Imaging	Yes	Yes	Yes
`heartwall`	*Tracks the mouse heart movement by stimulus*	Medical Imaging	Yes	No	Yes
`cfd`	*Finite volume solver for 3D Euler equations for flow*	Fluid Dynamics	Yes	Yes	Yes
`lud`	*Calculate the solutions of a set of linear equations*	Linear Algebra	Yes	Yes	Yes
`hotspot`	*Estimate processor temperature*	Physical Simulation	Yes	Yes	Yes
`nw`	*Optimization method for DNA sequence alignments*	Bioinformatics	Yes	Yes	Yes
`kmeans`	*Clustering algorithm*	Data Mining	Yes	Yes	Yes
`srad`	*Speckle reducing anisotropic diffusion*	Image Processing	Yes	Yes	Yes
`streamcluster`	*Finds medians to assign points to nearest centers*	Data Mining	Yes	Yes	Yes
`particlefilter`	*Locate object location based on noise and path*	Medical Imaging	Yes	Yes	Yes
`pathfinder`	*Dynamic programming to find a path on a 2D grid*	Grid Traversal	Yes	Yes	Yes
`gaussian`	*Solving variables in a linear system*	Linear Algebra	Yes	Yes	No
`nn`	*Find k-nearest neighbors from an unstructured data set*	Data Mining	Yes	Yes	Yes
`lavaMD`	*Calculate particle potential and relocation in 3D*	Molecular Dynamics	Yes	Yes	Yes
`myocyte`	*Simulate the behavior of cardiac hear muscle cell*	Biological Simulation	Yes	Yes	Yes

Table 2.5: Parboil Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
`bfs`	*Breadth-first-search*	Graph Algorithm	Yes	Yes	Yes
`cutcp`	*Compute Coulombic potential for a 3D grid*	Molecular Dynamics	Yes	Yes	Yes
`histogram`	*Compute 2D saturating histogram with maximum 256 bins*	Data Mining	Yes	Yes	Yes
`lbm`	*Fluid dynamics simulation using Lattice-Bolzmann Method*	Fluid Dynamics	Yes	Yes	Yes
`mm`	*Dense matrix-matrix multiply*	Linear Algebra	Yes	Yes	Yes
`mri-gridding`	*Compute regular data grid via weighted interpolation*	Medical Imaging	Yes	Yes	Yes
`mir-q`	*Compute scanner configuration for calibration in 3D MRI*	Medical Imaging	Yes	Yes	Yes
`sad`	*Sum of absolute differences kernel in MPEG video encoders*	Image Processing	Yes	Yes	Yes
`spmv`	*Compute the product of a sparse matrix with a dense vector*	Linear Algebra	Yes	Yes	Yes
`stencil`	*An iterative Jacobi stencil operation on a regular 3D grid*	Cellular Automation	Yes	Yes	Yes
`tpacf`	*Analyze the spatial distribution of astronomical bodies*	Data Mining	Yes	Yes	Yes

Table 2.6: SHOC Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
`qtclustering`	*Group genes into high quality clusters*	Bioinformatics	Yes	No	No
`s3d`	*Compute chemical reaction rate across a 3D grid*	Simulation	Yes	Yes	No
`scan`	*Parallel prefix sum of floating point numbers*	Data Mining	Yes	Yes	No
`reduction`	*Sum reduction operation of floating point numbers*	Data Mining	Yes	Yes	No
`md`	*Lennard-Jones potential computations*	Molecular Dynamics	Yes	Yes	No
`fft`	*Fast Fourier transform*	Signal Processing	Yes	Yes	No
`sgemm`	*Single precision general matrix multiply*	Linear Algebra	Yes	Yes	No
`sort`	*Fast radix sort program*	Data Mining	Yes	Yes	No
`stencil2d`	*Standard 2d 9 points stencil calculation*	Cellular Automation	Yes	Yes	No
`bfs`	*Breadth-first-search*	Graph Algorithm	Yes	Yes	No
`spmv`	*Sparse matrix vector multiplication*	Linear Algebra	Yes	Yes	Yes

Polybench [40] is a benchmark containing kernels that are converted from structural/nonstructural loop-nests. These loops are previously utilized for evaluating Polyhedron Model based optimization tools. The features about Polybench are summarized in Table 2.7.

Mars [33] includes several data-mining applications implemented on GPU using the famous Map-Reduce framework [115]. The six applications are summarized in Table 2.8. They share a common kernel library that implements the Map-Reduce operation primitives – the *MarsLib*.

Longstar Benchmark [41] focuses on applications that are irregular. Most of the computations in these applications are data-dependent or topology-dependent. Their characteristics are summarized

Table 2.7: Polybench Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
2dconv	*2D convolution*	Linear Algebra	Yes	Yes	Yes
2mm	*2 matrix multiply*	Linear Algebra	Yes	Yes	Yes
3dconv	*3D convolution*	Linear Algebra	Yes	Yes	Yes
3mm	*3 matrix multiply*	Linear Algebra	Yes	Yes	Yes
atax	*Matrix transpose and vector multiplication*	Linear Algebra	Yes	Yes	Yes
bicg	*Bicg kernel for BiCGStab linear solver*	Linear Algebra	Yes	Yes	Yes
corr	*Correlation computation*	Linear Algebra	Yes	Yes	Yes
covar	*Covariance computation*	Linear Algebra	Yes	Yes	Yes
fdtd2d	*2D finite difference time domain kernel*	Simulation	Yes	Yes	Yes
gemm	*matrix multiply*	Linear Algebra	Yes	Yes	Yes
gesummv	*Scalar vector and matrix multiplication*	Linear Algebra	Yes	Yes	Yes
gramschm	*Gram-schmidt process*	Linear Algebra	Yes	Yes	Yes
mvt	*Matrix vector product and transpose*	Linear Algebra	Yes	Yes	Yes
syr2k	*Symmetric rank-2k operations*	Linear Algebra	Yes	Yes	Yes
syrk	*Symmetric rank-k operations*	Linear Algebra	Yes	Yes	Yes

Table 2.8: Mars Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
sm	*Find the position of a string in a file*	Data Mining	Yes	No	No
ii	*Build inverted index for links in HTML files*	Data Mining	Yes	No	No
ss	*Compute pair-wise similarity score for docs*	Data Mining	Yes	No	No
mm	*Multiply two matrices*	Linear Algebra	Yes	No	No
pvc	*Count distinct page views from web logs*	Data Mining	Yes	No	No
pvr	*Find the top ten hottest pages in the web log*	Data Mining	Yes	No	No

Table 2.9: Longstar Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
bfs	*Breadth first search*	Graph Algorithm	Yes	No	No
bh	*Simulate the gravitational forces in Barnes-Hut algorithm*	Simulation	Yes	No	No
dc	*Lossless compression upon double-precision FP data*	Signal Processing	Yes	No	No
dmr	*Meshrefinement algorithm from computational geometry*	Image Processing	Yes	No	No
pta	*Andersen's flow/context-insensitive points-to analysis*	Graph Algorithm	Yes	No	No
sp	*Heuristic SAT-solver based on Bayesian inference*	Graph Algorithm	Yes	No	No
sssp	*Shortest path in a directed graph with weighted edges*	Graph Algorithm	Yes	No	No
tsp	*Traveling salesman problem*	Graph Algorithm	Yes	No	No

in Table 2.9. Other characteristics about irregular programs on GPUs can be found in [116, 117].

CUDA SDK [42] is the official GPU benchmark collecting a number of applications from a variety of domains to demonstrate the superior performance of GPU computing as well as to introduce how to exploit the various features of CUDA/OpenCL in a professional way. The commonly-used applications for evaluation in SDK are summarized in Table 2.10.

GPGPU-Sim [43] Besides, the GPGPU-Sim simulator itself contains some evaluation applications in its distribution. These applications are later used for validating GPU-related designs, especially on GPGPU-Sim. Their characteristics are summarized in Table 2.11.

Finally, there are plenty of other characterization works about GPGPU applications, such as [114, 118, 119, 120, 121], etc. Interesting readers can refer to them for more deeply characterization of the existing GPGPU applications.

Table 2.10: Commonly-used CUDA-SDK Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
bilateralFilter	*Edge-preserving non-linear smoothing filter*	Image Processing	Yes	Yes	Yes
binomialOption	*Evaluate option call price using binomial model*	Computational Finance	Yes	Yes	Yes
BlackScholes	*Evaluate option call price using Black-Scholes model*	Computational Finance	Yes	Yes	Yes
convolutionFFT2D	*2D convolutions using FFT*	Image Processing	Yes	Yes	Yes
dct8x8	*Discrete cosine transform for blocks of 8 by 8 pixels*	Image Processing	Yes	Yes	Yes
dxtc	*High quality DXT compression*	Image Processing	Yes	Yes	Yes
dwtHaar1D	*1D discrete Haar wavelet decomposition*	Image Processing	Yes	Yes	Yes
eigenvalues	*Eigenvalues of a tridiagonal symmetric matrix*	Linear Algebra	Yes	Yes	Yes
fastWalshTransform	*Hadamard-ordered Fast Walsh transform*	Linear Algebra	Yes	Yes	Yes
FDTD3d	*Finite differences time domain progression stencil*	Cellular Automation	Yes	Yes	Yes
grabcutNPP	*GrabCut approach using the 8 neighborhood*	Graph Algorithm	Yes	Yes	Yes
histogram	*64/256 bin histogram*	Data Mining	Yes	Yes	Yes
imageDenoising	*Using KNN and NLM for image denoising*	Image Processing	Yes	Yes	Yes
lineOfSight	*A simple line-of-sight algorithm*	Graphic Application	Yes	Yes	Yes
Mandelbrot	*Mandelbrot or Julia sets interactively*	Graphic Application	Yes	Yes	Yes
matrixMul	*Matrix multiplication*	Linear Algebra	Yes	Yes	Yes
mergeSortv	*Merge Sort algorithm*	Data Mining	Yes	Yes	No
MersenneTwister	*The Mersenne Twister random number generator*	Signal Processing	Yes	Yes	Yes
MonteCarlo	*Evaluate option call price using Monte Carlo approach*	Computational Finance	Yes	Yes	Yes
nbody	*All-pairs gravitational n-body simulation*	Simulation	Yes	Yes	Yes
oceanFFT	*Simulate an Ocean height field*	Simulation	Yes	Yes	Yes
reduction	*Compute the sum of a large arrays of values*	Data Mining	Yes	Yes	No
scalarProd	*Calculate scalar products of input vector pairs*	Linear Algebra	Yes	Yes	Yes
scan	*Parallel prefix sum*	Data Mining	Yes	Yes	Yes
SobelFilter	*Sobel edge detection filter for 8-bit monochrome images*	Image Processing	Yes	Yes	Yes
SobolQRNG	*Sobol Quasirandom Sequence Generator*	Computational Finance	Yes	Yes	Yes
transpose	*Matrix transpose*	Linear Algebra	Yes	Yes	Yes

Table 2.11: GPGPU-Sim Benchmark Characteristics

Application	Description	Domain	CUDA	OpenCL	C
aes	*AES algorithm in CUDA to encrypt and decrypt files*	Cryptography	Yes	No	No
dc	*A discontinuous Galerkin time-domain solver*	Simulation	Yes	No	No
lps	*3D Laplace Solver*	Computational Finance	Yes	No	No
lib	*Monte Carlo simulation in London-interbank-offered-rate Model*	Computational Finance	Yes	No	No
mum	*Pairwise local sequence alignment for DNA string*	Bioinformatics	Yes	No	No
nn	*Convolutional neural network to recognize handwritten digits*	Machine Learning	Yes	No	No
nqu	*The N-Queen solver*	Simulation	Yes	No	No
ray	*Ray-tracing (rendering graphics with near photo-realism)*	Graphic Application	Yes	No	No
sto	*Sliding-window implementation of the MD5 algorithm*	Data Mining	Yes	No	No
wp	*Accelerate part of the Weather Research and Forecast Model (WRF)*	Simulation	Yes	No	No

2.4.3 Profiling-Tools

The most frequently used profiling tools for GPGPU programs on NVIDIA products are: *Visual Profiler*, *Command-line Profiler* and *nvprof*. In this thesis, the *command-line profiler* and *nvprof* are intensively utilized for measuring different runtime events and performance metrics, such as kernel execution time, L1 hit rate, etc. Please refer to [122] for the details about the profiler tools.

2.5 Conclusion

In this chapter, we gave a brief introduction about GPGPU. Combining the machine model, the execution model, the programming model and the evaluation model, it can be seen that GPGPU has already evolved to be a practical, concrete and complete programming & execution environment.

Since NVIDIA does not reveal sufficient details about GPU architectures, as well as CUDA runtime and low-level drivers, we could not have sufficient reliable materials to give a thorough description about GPGPU. The information provided in this chapter has been derived from the whitepapers of different GPU architectures, the official CUDA programming tutorials and various research articles (e.g., [45, 46, 47, 48, 49, 50, 51, 52, 10, 93, 123, 124, 96, 95], etc). In the next chapter, we discuss our analytic model X for parallel machines such as GPU.

CHAPTER 3

The X-Model for Parallel Machines

To continuously comply with Moore's Law, modern parallel machines become increasingly complex. Effectively tuning application performance for these machines therefore becomes a daunting task. Moreover, identifying performance bottlenecks at application and architecture level, as well as evaluating various optimization strategies, are becoming extremely difficult when the entanglement of numerous correlated factors is being presented.

To tackle these challenges, in this chapter we present a visual analytical model named "X". It is intuitive and sufficiently flexible to track all the typical features of a parallel machine. Different from the conventional analytic models that focus on the temporal state of a representative core or thread, our proposed X-model concentrates on the spatial state of the parallel machines – the distribution of concurrent threads among different subsystems of these machines, while predicting the overall throughput based on such state. One major highlight of our model is its tractability as it only requires a small number of essential parameters from the application and architecture. Meanwhile, it is able to effectively help users investigate the combined-effects of different types of parallelism: the instruction-level-parallelism (ILP), the thread-level-parallelism (TLP), the memory-level-parallelism (MLP) and the data-level-parallelism (DLP). Through the X-model, developers and architects can quickly draw an intuitive figure called X-graph to identify performance bottlenecks and play "what-if" scenarios to evaluate the effectiveness of the proposed optimization techniques by investigating their individual and combined effects. The basic version of the X-model called Transit model has been presented at the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC-15) [84]. The complete X-model has been presented at the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS-16) [85].

3.1 Introduction

Despite the fact that Moore's Law has continued to be promising, the mainstream computing has been leveraging multiprocessors and parallel applications extensively for superior performance, due to the end of frequency scaling for uniprocessors. However, decades of practical experience demonstrated that analyzing and optimizing performance for the complex modern parallel architectures still remains a challenging task, especially concerning the huge design space with divergent types of parallelism to exploit. Therefore, developers often found themselves lost when exploring a large number of design options and their combined effects. For instance, as one of the most popular throughput-oriented many-core architectures, GPU is well-known for its ability to initiate thousands or even millions of concurrent threads. A performance metric called "occupancy" is then proposed to measure the ability of a workload to utilize the available thread slots on a GPU for peak performance. However, programmers who attempt to pursue high occupancy for better performance then become confused, as more recent research indicates that maximizing occupancy may lead to register spilling and inferior cache performance [125]. They become even more hesitant when other research demonstrates that if there is plenty of instruction-level-parallelism, better performance can be achieved with lower occupancy [103].

These challenges emerge because developers often constrain themselves to address a very specific performance issue for a machine component (e.g., registers, caches, main memory, etc.) without much indication for better understanding of the global systematic effects. In other words, as modern parallel architectures become increasingly complicated, most performance factors are not independent with each other but are often intercorrelated or even mutual conflicted. Therefore, a high-level and easy-to-use performance analysis tool, that can provide comprehensive information for identifying performance bottlenecks and demonstrate the performance variation characteristics when a particular factor is altered, is highly desired.

In this chapter, we present such a performance analysis tool called "X-model", which is a high-level and visualized analytic model for general parallel machines. It can help developers understand the observed phenomena and derive new optimization strategies. Based on the spatial state of the parallel machine, the model is able to comprehensively investigate the combined effects of various types of parallelism: the instruction-level-parallelism (ILP), the thread-level-parallelism (TLP), the memory-level-parallelism (MLP) and the data-level-parallelism (DLP); and it only requires very few essential parameters from application and architecture for the model construction. With our X-model, developers and architects can easily draw an intuitive figure called "X-graph" to identify performance bottlenecks and discern potential optimizations. More significantly, by drawing an X-graph, designers and researchers can easily find out, in a visualized and conceptual way, whether a proposed technique by a manuscript is effective for resolving the problem it targets and why, as well as what else can be done subsequently. This chapter thus makes the following contributions:

- We propose a high-level visualizable analytic model for parallel machines that can comprehensively analyze the joint-effects of numerous factors such as MLP, ILP, TLP and DLP (Section 3.3.1).
- We propose an approach to integrate a shared cache into the X-model (Section 3.3.2) to form X-graphs that can reflect complex cache effects (Section 3.3.3). Based on these X-graphs, interesting performance insights are derived (Section 3.3.4).
- We provide a thorough case study on how to leverage the X-model for evaluating different optimization options for real applications. We demonstrate that our model can identify the limiting

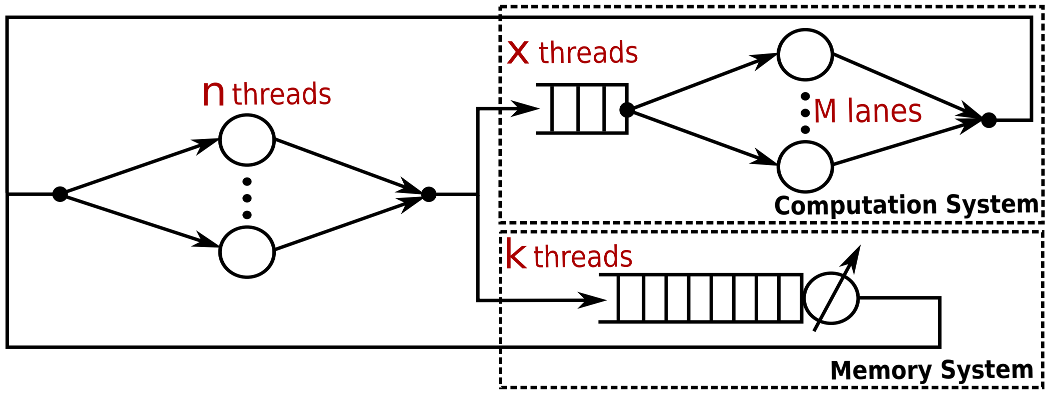

Figure 3.1: Baseline Multithreaded Machine Model.

factors, suggesting potential optimization techniques, reasoning and bounding the effectiveness of a technique, and explore new opportunities for further optimizations (Section 3.3.6).

3.2 The Basic Transit Model

Before describing the *X-model* in detail, we first introduce the *Transit Model* that we proposed in [84] for visualizing simple performance analysis for a multithreaded machine. Although the X-model is built upon the Transit model, we significantly extend it to include important features such as analyzing various types of parallelisms and expressing sophisticated cache effects on modern architectures. These features are essential, and can significantly affect the overall performance of modern parallel machines.

In the Transit model, a multithreaded machine is partitioned into a computation system (CS) and a memory system (MS). Their boundary is flexible (i.e., can move along the memory hierarchy) depending on the requirements. The CS throughput is viewed as the primary performance metric while the MS throughput is also of interest. As shown in Figure 3.1, the multithreaded machine is modeled as an interactive queuing network. There are totally n threads in the machine, in which x of them are in CS and $n-x=k$ in MS. The CS is a single-queue-multiple-server system. Each server denotes an in-order computation lane that can perform one compute operation per cycle. The MS is viewed as an aggregated queuing system. During execution, a typical thread executes in one of the M lanes of CS for Z cycles on average, and fetches a memory request. It then enters MS for L cycles to do the data fetching. After being fetched, the thread enters CS again, starting a new turnaround. The related notations used in this chapter (for application and architectural input, intermediate variables, output) are listed in Table 3.1.

CS: As shown in Figure 3.1, with x threads occupying x lanes in CS, the utilization of CS would be x/M. As one computation lane generates one operation per cycle, the CS throughput function $g(x)$ can be expressed as $g(x) = min(x,M)$, which is a roofline-like figure shown in Figure 3.2-B. Since there is one memory request per Z cycles on average, in total there are $g(x)/Z$ memory requests per cycle. This is the demand throughput from CS to MS. Note that we reverse X-axis' direction for

Table 3.1: Notations Used In This Paper

Symbol	Meaning	Unit	Parameter Type
n	Threads in the parallel machine	thds	App Input
k	Threads in the memory system (MS)	thds	Intermediate
x	Threads in the computation system (CS)	thds	Intermediate
$f(k)$	MS supply throughput to CS	B/s	Output
$g(x)$	MS demand throughput from CS	B/s	Output
Z	Compute intensity (ops/bytes ratio)	ops/B	App Input
E	Instruction-level-parallelism degree	-	App Input
R	Maximum sustainable MS throughput	B/s	Arch Input
M	Computation lanes	ops	Arch Input
π	CS transition point (when CS saturated)	(thds, B/s)	Intermediate
δ	MS transition point (when MS saturated)	(thds, B/s)	Intermediate
L	Average MS access latency	s	Arch Input
h	Shared cache hit rate	-	Intermediate
ψ	Position of cache peak	thds	Intermediate

further integration and utilization. π in Figure 3.2-B represents the CS transition point, at which CS begins to saturate.

MS: With k threads filling up k pipeline stages in MS shown in Figure 3.1, if the MS pipeline delay is L, the utilization of MS can be described as k/L if we assume each thread occupies a pipeline stage and each pipeline stage delay is 1. Consequently, the MS throughput function $f(k) = min(R, (kR)/L)$. This is also a roofline-like figure shown in Figure 3.2-A. It illustrates the supply throughput from MS to CS. δ represents the MS transition point, at which MS starts to saturate.

Based on the *flow balance property* [126, 127], for a steady state of the system, $f(k) = g(x)$. Therefore, if we combine Figure 3.2-A and Figure 3.2-B, a **cross-roofline figure** can be obtained, shown in Figure 3.3. This is called a *transit figure* [84]. The intersection point of $f(k)$ and $g(x)$ is the equilibrium between the demand throughput and supply throughput of MS, which is exactly the current MS throughput, or $f(k_0)$ when k_0 is used to describe the k value at the intersection. Consequently, the CS throughput is $Z * f(k_0)$.

The **inputs** of the transit model are three architecture-related parameters R, L, M and two application-related parameters Z and n (described in Table 3.1). In the transit model, since the raw memory latency L is very difficult to change in practice, it is postulated to be constant; the other four are changeable. The **output** of the model is the machine performance, or the delivered throughput of CS and MS. Three principles are proposed to evaluate the CS and MS throughput in the transit figure:

- ***Principle 1:*** *If the intersection of $f(k)$ and $g(x)$ goes up, then MS throughput increases.*
- ***Principle 2:*** *If the intersection goes up and Z is unchanged, then CS throughput increases.*
- ***Principle 3:*** *If compute intensity Z is increasing and the intersection is on the right side of CS transition point π, then CS throughput increases.*

The other focus of the transit model is on illustrating various state transitions of the multithreaded machine based on different types of performance bounds, including *thread-bound, computation-*

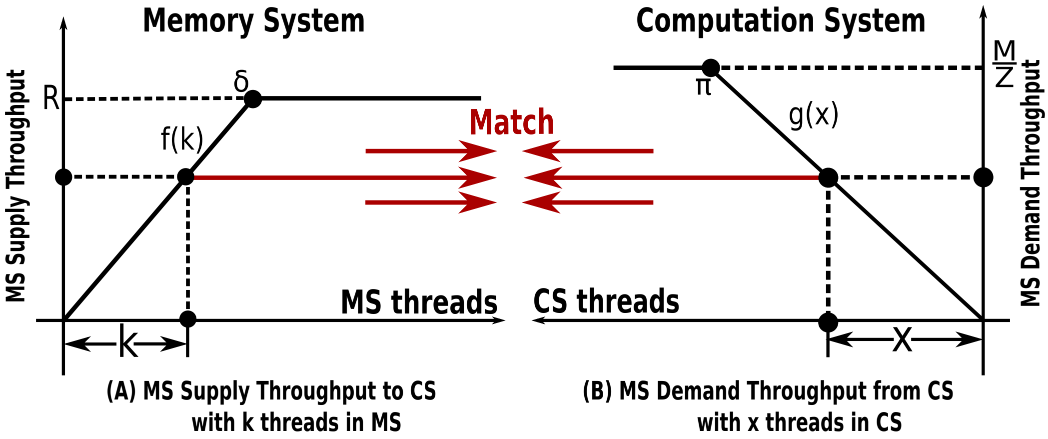

Figure 3.2: (A): MS supply throughput function $f(k)$ and (B): CS throughput demand function $g(x)/Z$ to MS.

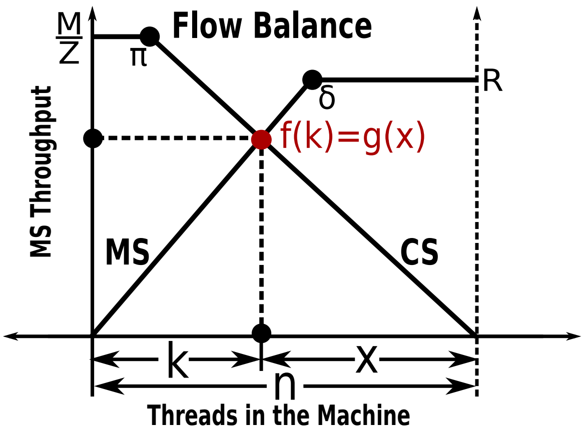

Figure 3.3: Transit Figure: the intersection of $f(k)$ and $g(x)$ represents the equilibrium between service demand and supply of MS. It indicates the spatial machine state: within the total n threads, k of them are in MS and x in CS.

bound, memory-bound and *capacity-bound*. Please refer to [84] for detailed descriptions.

3.3 The X-Model

In this section, we present the X-model. We use the letter "**X**" to label the model because it illustrates the general shape of the model — a cross-roofline. Unlike the original roofline model which is built generally for sequential machines, the X-model is a dynamic, high-level and visualized analytic model for parallel machines. Moreover, with only six parameters from application and architecture, and based on the present spatial state of a parallel machine, X-model can help users comprehensively explore the combined effects of various types of parallelism, including TLP, ILP, MLP, and DLP. This is very different than the transit model, in which only simple performance analysis (i.e., computation/memory/thread/capacity bound analysis) can be conducted. Furthermore, the X-model integrates the shared cache effects into the parallel machine to form a more complete model for matching the complex modern multi- and many-core architectures, in which cache effects directly impact the delivered performance. Next, we demonstrate how to operate the X-model for performance analysis and evaluation. Then, we discuss how to model and integrate the cache effects

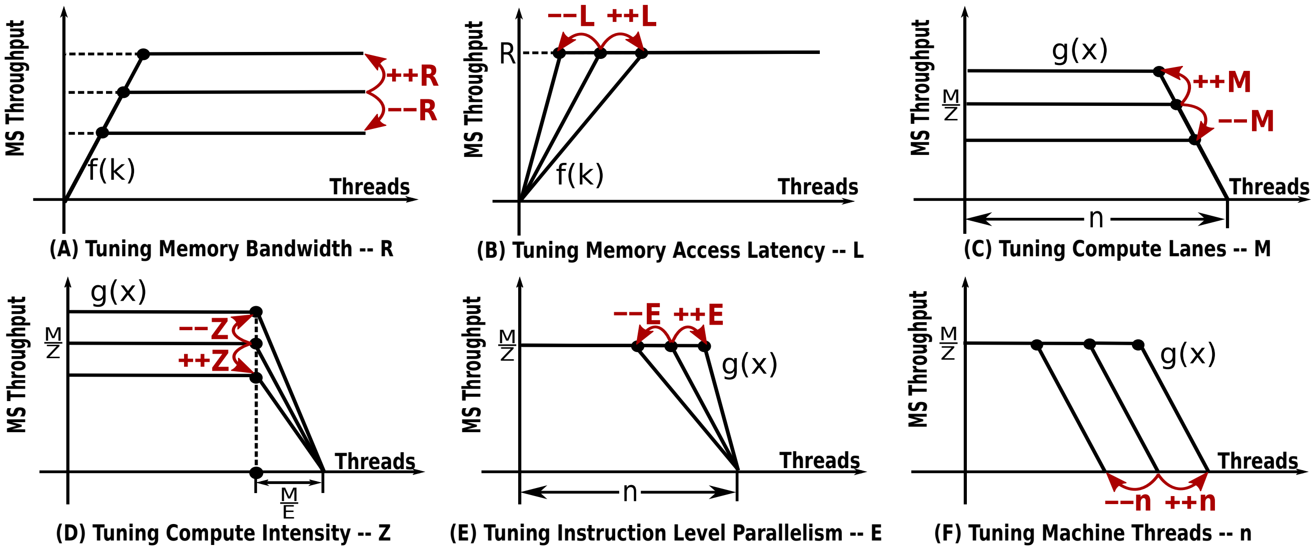

Figure 3.4: Operating X-Model.

in the X-model. The parameters used in the discussion are listed in Table 3.1.

3.3.1 Operating X-Model For Analysis and Evaluation

Memory-Level-Parallelism (MLP)

As shown in Figure 3.4-B, L is the average memory access latency. In the transit model, L is viewed as a constant parameter. Therefore, the reciprocal of L is just the average per-thread memory throughput. Before MS throughput function $f(k)$ hits its upper bound R (or reaches the MS transition point δ), $1/L$ is the slope of $f(k)$. Since L is constant, the sloping part of the curve is a straight line. Beyond the MS transition point δ ($k >= \delta$), $f(k)$ becomes flat as MS is already overloaded with the increasing number of k threads in MS.

In the X-model, as $1/L$ is the average per-thread throughput and R is the overall throughput, then $R/(1/L) = RL$ essentially implies the number of threads required to saturate the MS, or the **MLP of the machine**. Usually, with R being fixed, the larger latency L, the more threads (a larger k) are needed to fill the pipeline slots and hide the latency (Figure 3.4-B). Alternatively, with L being fixed, the larger throughput R also implies that more threads are necessary to reach R (Figure 3.4-A), which is just the MLP. On the other hand, the exploited MLP, or the **MLP of the workload**, is proportional to k, which is the number of threads in MS.

Instruction-Level-Parallelism (ILP)

The effect of **ILP of the machine**, which is also the ILP of CS since MS does not have the ILP concept, is difficult to be illustrated in the X-graph because of its entangled relationship with the TLP in CS. Their combined effect is the number of computation lanes (i.e., M) in CS. Since most of

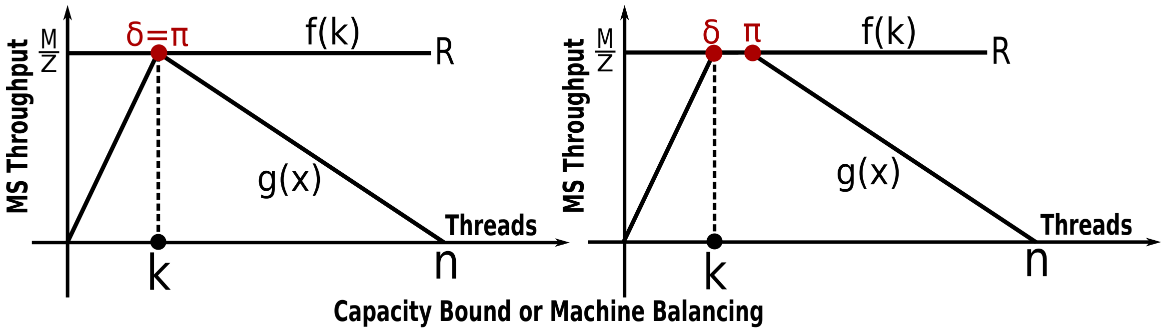

Figure 3.5: Capacity Bound or **Machine Balance**: both CS and MS attain its best performance. π is the CS transition point and δ is the MS transition point. n in the left figure indicates the TLP of the machine while for the right figure, due to the shortage of machine capacity, some threads are idle.

the modern parallel machines adopt dynamic scheduling, both ILP and TLP of the workload can be exploited via these lanes simultaneously. Note that for a real machine, the ability to exploit ILP and TLP heavily relies on the underlying hardware design (see Section 3.4).

ILP of the workload is more important. It indicates the parallelism inside the scope of a single thread, or how many computation lanes a thread can leverage at the same time. In the transit model, ILP of the machine is assumed as one, meaning that a thread only occupies a single lane. In the X-model, a variable E is employed to describe the ILP degree of the workload. As shown in Figure 3.4-E, we modify the CS curve $g(x)$ to address ILP. With a larger E, relatively fewer threads are required in CS (a smaller x) to fill up the available lanes and saturate CS. Note that compared to Z (the compute intensity in Figure 3.4-D), E defines the slope of $g(x)$ while Z acts as a scaling factor when integrating CS and MS curves (see Section 3.2 and Figure 3.2) for the X-graph.

Thread-Level-Parallelism (TLP)

Regarding the **TLP of the workload**, the X-model is similar to the transit model; it is simply n, the total number of threads (Figure 3.4-F). However, the **TLP of the machine** in the X-model is quite different. It is defined as the minimum number of threads to hit the *capacity bound* or *machine balance*. As shown in Figure 3.5, two different scenarios of the machine balance are illustrated, at which both CS and MS attain its best performance. The capacity bound or machine balance describes the optimal state for software-hardware co-design since both CS and MS bandwidth are fully leveraged ($f(k) = R$, $g(x) = M/Z$) [84]. Unlike the right figure in Figure 3.5, the left one does not have any idle threads in either CS or MS. Therefore, its current n value is the **TLP of the resident parallel machine**.

Data-Level-Parallelism (DLP)

For the **DLP of the workload**, it is defined as a metric that measures the number of computation operations performed per data element, which is the ratio between computation operations and memory operations of the workload, or Z (**compute intensity**) shown in Figure 3.4-D. Meanwhile, the

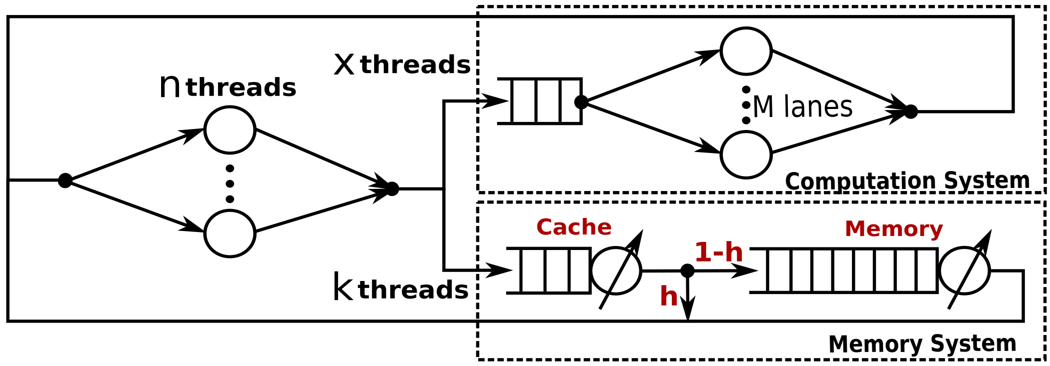

Figure 3.6: The new parallel machine model equipped with Shared Cache.

DLP of the machine indicates the intrinsic characteristic of the machine, which can be represented as M/R. Essentially, the relative relationship between DLP of the workload and DLP of the machine can be summarized as: *if DLP of the workload is less than DLP of the machine ($Z < M/R$), the system is memory bound; otherwise ($Z \geq M/R$), it is computation bound.* Note, DLP of the machine (M/R) is just the ridge point of the roofline model [128]. To some extent, it indicates the level of difficulty for programmers to achieve the peak computation performance for the underlying architecture.

3.3.2 The X-Model with Cache Effects

In this subsection, we model a MS with shared cache integrated. Based on the obtained new MS throughput curve, we then show the complete X-model in the next subsection. After that, we describe two novel observations revealed by the X-model.

In the transit model, a basic assumption is that threads in a multithreaded machine are independent of each other and there is no cache interference among threads. Moreover, the average memory access latency L is fixed. With these assumptions, a roofline-like figure for the MS throughput function $f(k)$ is generated (Figure 3.2-A). In the X-model, we relax these restrictions and replace the roofline-like $f(k)$ with a more practical throughput curve that can better address the cache effects.

As shown in Figure 3.6, on top of the transit model, an intermediate cache system (\$) is placed ahead of the main memory m in MS. If the hit rate of the shared cache is h, a memory request would have a probability of h to be quickly returned from the cache while a probability of $(1-h)$ to be slowly returned from the main memory. Therefore, if we use $L_\$$ to denote cache latency and L_m to denote off-chip memory latency, the average MS latency L with k threads in MS (L_k) would be:

$$L_k = h * L_\$ + (1-h) * L_m \tag{3.1}$$

and the new MS throughput function $f(k)$ with k threads is

$$f(k) = k/L_k \tag{3.2}$$

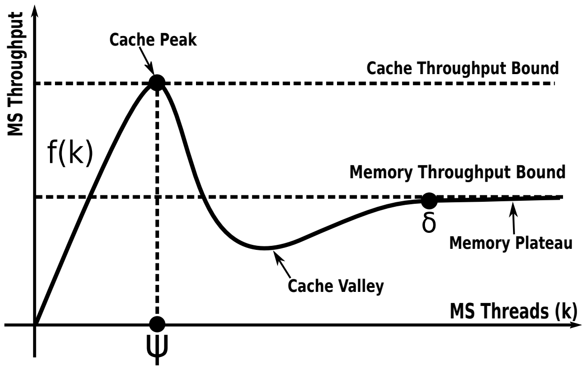

Figure 3.7: Throughput functions $f(k)$ for a MS with cache integrated.

The remaining question is to find a proper cache model that supports multithreading. We adopt the one proposed by Jacob et. al. [129] to accomplish this. If the cache size is $S_\$$ and there are k threads accessing the cache, each thread shares on average $S_\$/k$ of the cache storage. The hit rate seen by a thread hence can be represented as:

$$h(\frac{S_\$}{k}) = 1 - (\frac{S_\$}{\beta k} + 1)^{-(\alpha-1)} \tag{3.3}$$

where α and β are two parameters describing the locality of the workload – *the better locality, the larger α and smaller β*. Meanwhile, the main-memory throughput is still bounded by R. Therefore,

$$L_m = max\{L, k/R\} \tag{3.4}$$

where L is the constant memory latency before MS is saturated, as discussed before. Combining Eq. 3.1, 3.2, 3.3 and 3.4, we remodel the MS throughput function $f(k)$ as

$$f(k) = k/[L_\$ + (max\{L, \frac{k}{R}\} - L_\$)(\frac{S_\$}{\beta k} + 1)^{1-\alpha}] \tag{3.5}$$

A sample figure for the new $f(k)$ is shown in Figure 3.7. At the beginning, with the efficient utilization of the cache, the MS throughput increases almost linearly with the expanded MS threads, and eventually reaches a peak. We label this peak as **cache peak** where $k = \psi$. However, once the aggregated working set for the increased k threads exceeds the cache capacity, thrashing occurs and performance starts to degrade ($k > \psi$). Note that with a hit rate h, there are on average $h * k$ threads in the cache and $(1-h)*k$ threads in the main memory. At this time, the $(1-h)*k$ threads are not sufficient to saturate the main-memory system. In other words, the MLP of MS cannot be fully exploited by $(1-h)*k$ threads (see Section 3.3.1-(1)). This explains why there is a performance valley after the cache peak: *the cache throughput drops so quickly without the memory throughput being increased fast enough to compensate*. We label this valley as **cache valley**. Beyond the ridge point of the cache valley, the main-memory starts to play the major role for performance as the cache impact diminishes. With the further expanded threads, $f(k)$ increases again as effective MS bandwidth continuously being exploited. Once the thread number reaches the MS transition point δ,

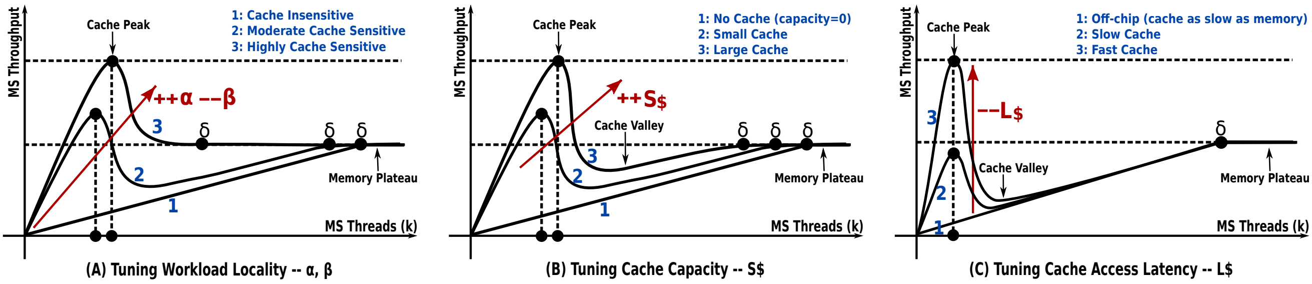

Figure 3.8: The three operations to tune the cache-integrated MS throughput function $f(k)$: (A) tuning working load locality; (B) tuning cache capacity; (C) tuning cache access latency.

$f(k)$ remains stable afterwards. We label this stable throughput as the **memory plateau**.

In Figure 3.8, we summarize three operations to tune the cache-integrated MS throughput function $f(k)$ (i.e., Eq. 3.5). The first operation is workload-locality related. As shown in Figure 3.8-A, by tuning α and β, we can obtain three representative shapes of $f(k)$ corresponding to three different cache-sensitivity conditions: *cache insensitive (CI)*, *moderately cache sensitive (MCS)* and *highly cache sensitive (HCS)*. The CI applications present the same curve (Curve 1 in Figure 3.8-A) as the $f(k)$ function of MS without cache. For both MCS and HCS applications, there is a cache peak. However, the cache peak of MCS applications (Curve 2) is lower and flatter than that of HCS (Curve 3). In addition, the MCS cache peak can be reached with fewer threads. Beyond the cache peak, there is a cache valley for MCS applications and possibly for HCS applications, depending on the hit rate and MLP of the MS. However, the valley of HCS, if exists, is not as deep as that in MCS due to the less significant cache effects towards performance in MCS.

The other two operations are architecture related. Figure 3.8-B shows the condition of tuning cache capacity ($S_\$$ in Eq. 3.5). Three curves correspond to *no cache*, *a small cache* and *a big cache*. Although the variations of the shapes are very similar to Figure 3.8-A, they are not exactly the same: the change of $S_\$$ is more like a scaling transform of the cache peak and valley, and the displacement of the curves is quite uniform. Finally, Figure 3.8-C illustrates the scenarios of tuning raw cache access latency ($L_\$$ in Eq. 3.5). Although this cannot be easily done theoretically, it can significantly affect the height of the cache peak and the depth of the cache valley. Comparing Curve 2 (*a slow cache*) with Curve 3 (*a fast cache*), it is clear that a fast cache is always beneficial, as it strengthens the positive cache effects by increasing the cache peak, while mitigates the negative effects through raising or smoothing the cache valley. Also note that the positions of the cache peak and valley do not change on x-axis when tuning $L_\$$.

3.3.3 X-graphs Reflecting Cache Effects

With the new $f(k)$, we are able to draw a complete X-graph. As shown in Figure 3.9-A, the X-graph is more comprehensive and accurate than the transit graph shown in Figure 3.3. It also highlights one of the major advantages of the X-model over the Roofline model [128]: *it compartmentalizes the machine into MS and CS*. Therefore, when the cache effects or other effects (e.g., scratchpad memory,

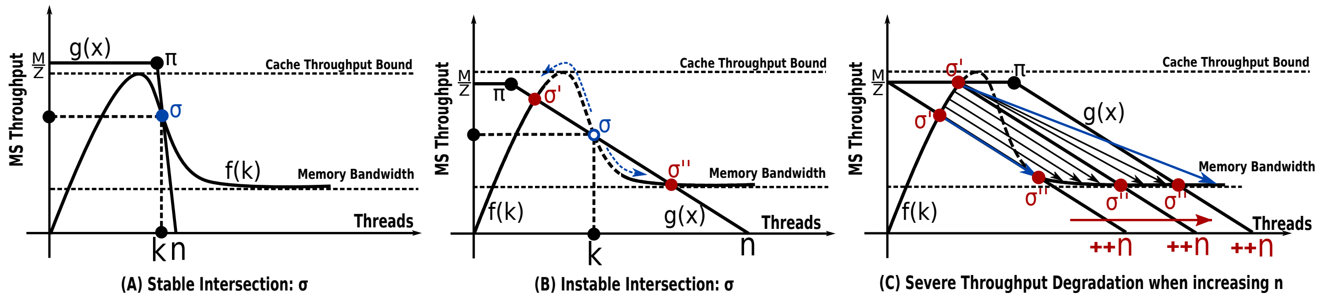

Figure 3.9: A complete X-graph reflecting cache effects. It illustrates three scenarios: (A) stable intersection; (B) unstable intersection; and (C) severe performance degradation when increasing n. The dashed part indicates the unstable region.

MSHRs, etc.) are needed to be reflected in the model, a new $f(k)$ based on a specific condition can be supplied without the interference from CS.

Note that we use the MS throughput as the y-axis in our X-graph instead of the CS throughput, albeit CS throughput seems more convenient for performance lookup. This is because, unlike $f(k)$, $g(x)$ is generally a regular roofline. If converting a complex cache-effect integrated $f(k)$ (Figure 3.8) into the CS space by multiplying Z, the process can be complicated. Therefore, the current approach simplifies the model.

3.3.4 Interesting Insights Gained From the X-graph

In this subsection, we will demonstrate two interesting insights on performance observed from the X-graph:

- An unstable intersection point exists in the X-graph but cannot be actually observed in practice;
- If Z is small and E is large, the workload may suffer from sharp performance degradation at certain point.

Unstable Intersection

Slightly different from Figure 3.9-A, $f(k)$ and $g(x)$ intersect at three points in Figure 3.9-B: σ, σ' and σ''. The key observation gained from this X-graph, is that the intersection σ is essentially unstable and cannot be observed on real parallel machines, because any perturbation will cause the equilibrium (Figure 3.3) to move away:

Consider the scenario that the current intersection is σ. At this time, k will be increased by one if a thread happens to leave the computation system and issues a memory request. Consequently, the MS throughput reduces as $f(k)$ decreases with a larger k (the descending dash-line part of $f(k)$). Meanwhile, since $x+k=n$ is fixed, x decreases by one. Although this decrease also causes $g(x)$ to reduce a bit (at the sloping part of $g(x)$), the reduced magnitude of $g(x)$ is smaller than that of $f(k)$ because the dropping slope of $f(k)$ is steeper than that of $g(x)$. Therefore, there is more throughput loss of MS than CS. Starting from the equilibrium σ, $f(k)$ becomes smaller than $g(x)$ after this process, causing more threads to leave CS than entering, since MS is the bottleneck currently (i.e., $f(k) < g(x)$). This leads to a further increase of k and triggers the same process again. Such process repeats until $f(k) = g(x)$, reaching a stable interaction σ''.

From the same initial state σ, the other possibility is that a thread happens to obtain the fetched data and aborts MS. This decreases k by one, which leads to the throughput increase for both MS and CS. However, as the slope of $g(x)$ is steeper than $f(k)$, after the process, $f(k) > g(x)$, making CS to be the performance bottleneck and more threads are likely to leave MS and being blocked in CS. Consequently, k decreases further, which will trigger the same process again. Such a process repeats until $f(k) = g(x)$. Under this condition, however, the machine state shifts leftwards and eventually settles at σ'.

To summarize, *any perturbation to k will cause the machine state to diverge from σ*. However, the intersection in Figure 3.9-A can be converged as the slope of $g(x)$ is steeper than that of $f(k)$. A perturbation is then revised under this condition, making this intersection stable. To explain this using a mathematical form, the stable scenarios in Figure 3.9-B need to meet the following **derivative** relationship:

$$|\partial g(x)/\partial x| > |\partial f(k)/\partial k| \tag{3.6}$$

which implies that the benefit from adding threads in CS should be greater than the benefit from reducing threads in MS (due to diminished cache conflict).

The remaining question for Figure 3.9-B is: *at which point (σ' or σ'') will the machine eventually converge to?* It is hard to say from the model itself. Mostly it depends on the thread distribution: if there are more threads in CS (x is large), CS is likely to have a higher throughput, which matches the good performance of MS with comparatively fewer threads in MS (k is smaller with a larger x under $x+k=n$). Under this scenario, the machine stabilizes at σ'. However, if there are fewer threads in CS, the lower throughput of CS also matches the poor performance of MS since excessive threads congest the cache, causing severe thrashing and resource shortage (e.g., MSHRs). The machine then stabilizes at σ''. Clearly σ'' is undesirable as the performance is poorer.

Severe Performance Degradation

We further explore the two stable intersections in Figure 3.9-B. As the machine state may be settled at either σ' or σ'', from σ' to σ'' the performance drops quite significantly. If we add more threads to the machine (i.e., increase n, or Figure 3.4-F), as shown in Figure 3.9-C, the positions of σ' and σ'' also move accordingly. However, when σ' coincides with the CS transition point δ, σ' starts to be constant. At this moment, the parallel machine is already computation bound although the cache can deliver higher throughput. The arrows in Figure 3.9-C indicate the magnitude of performance degradation that the machine might suffer from when increasing n: the *minimum* is from σ' to σ'', which occurs when $g(x)$ is tangent to $f(k)$; the *maximum* is $\frac{M}{Z} - R$, which is attained when there are infinite threads in the machine.

Table 3.2: Experiment Platforms. "LDS" is the number of load/store units per SM. "Scher" indicates the number of warp-schedulers per SM. "Disp" is the number of warp-dispatch-units per SM. Mwarps is the maximum number of warps per SM. δ(SP) is the transition point for the MS throughput with single-precision floating-point like data size (4 bytes) and fully coalescing access. The unit is warps – GB/s, e.g., 48/147 means MS throughput function saturates at 147 GB/s with 48 warps. δ(DP) is for 8 bytes data size with coalescing. There are at most 32 warps per thread block, so the X-axis stops at 32.

GPU	Arch	SM×SP	LDS	Freq	Mem Band	Dri/Rtm	Mwarps	Schr	Disp	δ(SP)	δ(DP)
GTX570	Fermi-2.0	15x32	16	1464 MHz	152 GB/s	6.5/4.0	48	2	2	48/147	24/152
Tesla K40	Kepler-3.5	15x192	32	876 MHz	288 GB/s	6.0/6.0	64	4	8	64/180	48/200
GTX750Ti	Maxwell-5.0	5x128	32	1137 MHz	86.4 GB/s	6.5/6.5	64	2	4	56/82	28/83

In summary, there are two forms of the X-model: *the regular one with cache* and *the simpler one without*. Generally, if users do not need to consider the cache effects, the basic X-model is more straightforward and simple. However, for the majority of the complex modern architectures, dealing with cache-level effects and optimizations is more common. In Section 3.6, we will show a case study using the regular X-model with cache effects.

3.4 Guidelines For Plotting X-Graph

In this section, we provide guidelines on how to draw an X-graph that represents the integration of features from workload and architecture. Our X-model provides a good abstraction for both the understudied architecture and the application. From the perspective of an architecture, it extracts three **machine-related parameters** M, R, L, based on which an **architectural X-graph** can be drawn first and it only requires to profile the hardware once. In this thesis, to showcase the ability of our model to address complex architectures, we choose to use one of the most popular throughput-oriented many-core architecture—GPU, for the purpose of evaluation and illustration. However, the same methodology can be applied to other parallel machines. Figure 3.10 shows the samples of architectural X-graphs based on the three major GPU generations (i.e., Fermi, Kepler and Maxwell) under single (SP) and double precisions (DP). To profile $f(k)$ (i.e., L and R) for the architectural X-graph, we use a modified CUDA version of the Stream Benchmark [130]. To profile the $g(x)$ (i.e., M), we developed a microbenchmark based on the method described in [103].

From the perspective of an application, the X-model extracts **three application-dependent parameters** Z, E, n, based on which the architectural X-graph shown in Figure 3.10 can be tuned to specifically address an application. The X-model provides a convenient way to *enable independent evaluation on architecture using a series of different application features*. It also provides a way to *predict application performance on an unreachable or nonexistent platform if those hardware features can be provided (R,M,L, and the ability to exploit ILP, TLP DLP and MLP)*. To draw an application X-graph, we first parse the application code/instructions via compiler/assembler. Once the ILP (i.e., E) is obtained, we then tune g(x) according to Figure 3.4-E, which corresponds to choose a curve from the $g(x)$ series with different Es, shown in Figure 3.10. Depending on the value of n, we can change the relative distance between $f(k)$ and $g(x)$, refer to Figure 3.4-F. Finally, when Z is available, we can divide CS throughput by Z to convert the $f(k)$ and $g(x)$ curves into the same

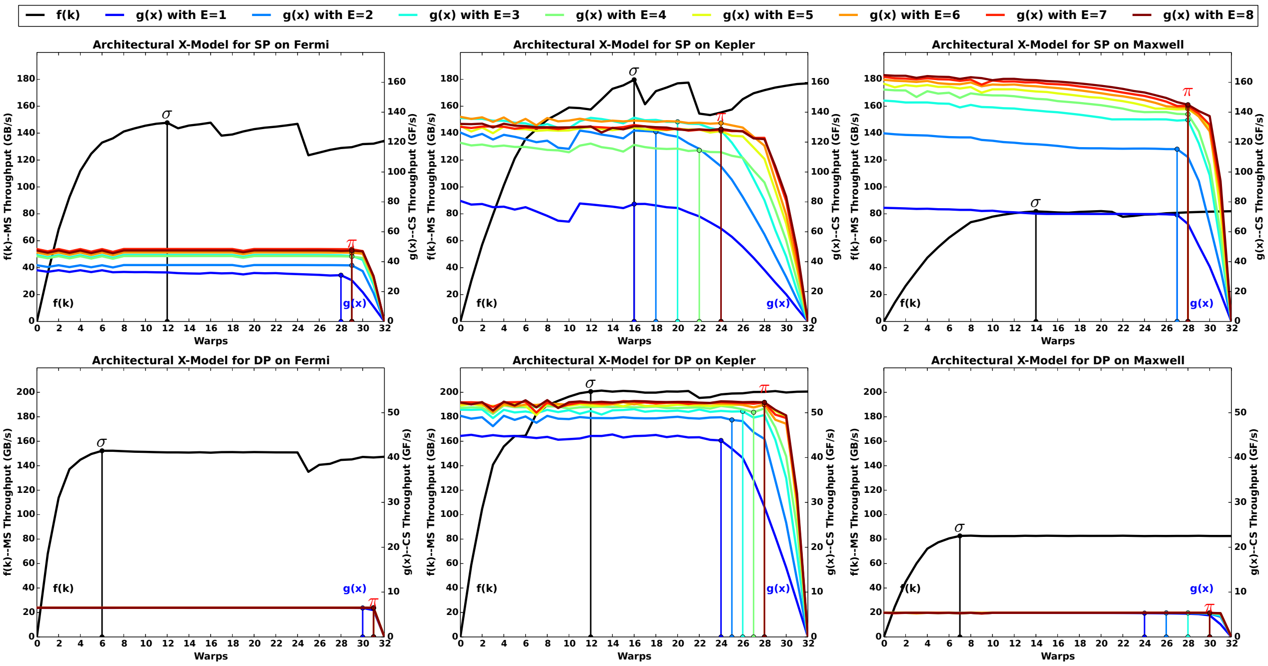

Figure 3.10: X-graphs for three different GPU architectures under single and double precisions.

MS throughput space (as can be seen, the left y-axis is MS throughput and the right y-axis is CS throughput, they are not in the same space). Thus, their intersection is just the current machine state, or present MS throughput. Following these three steps above, we can obtain the X-graph for a application running on a specific architecture. We will show some examples of applications' X-graphs in the next section.

3.5 Validation

In this section, we validate the X-model on the Kepler platform (listed Table 3.2). We use 12 practical applications `bfs`, `backprop`, `stencil`, `gesummv`, `hpccg`, `heartwall`, `leukocyte`, `nw`, `nn`, `spmv`, `atax`, `lud` from commonly-used benchmarks including Rodinia [37], Parboil [38], Polybench [131] and [132]. Based on the guideline introduced in the previous section, we take the Kepler architectural X-graph (Figure 3.10-B) as the starting-point and tune the $g(x)$ curve according to the application features, which are E, n and Z. To obtain these software-related parameters, we parse the SASS assembly code of the application. Regarding ILP or E, we use a new approach that is different from the existing one based on CFG analysis for a general machine [133]. Since Kepler, GPUs start to embed scheduling information in the SASS assembly code to simplify the hardware scheduler's task and reduce energy. We thus developed a tool to read this scheduling information from the cubin file and count the average number of instructions that are issued simultaneously, which is the ILP. Note the ILP obtained here is always less than or equal to two because the scheduling information is within the scope of a single warp and does not tell how many warp schedulers (4 for Kepler shown in Table 3.2) will select instructions from the single warp at runtime. In order to be accurate, we weight the ILP values for each code-block by the number of iterations for that block. Similarly, we also count the ratio between the number of total instructions and off-chip memory instructions for all the basic code-blocks, and weight by the number of loop iterations to calculate the value of computation intensity (Z). The loop iterations, in case branching, can be profiled using the user-managed profiler counters [122]. Finally, we calculate how many warps can be allocated simultaneously on a SM (i.e., the occupancy), which is the just the value of n. We developed a script to draw the X-graph and compared the predicted computation and memory throughput (i.e., the MS and CS throughput at the intersection of $f(k)$ and $g(x)$) with the values measured by the CUDA profiler. The results are shown in Figure 3.11.

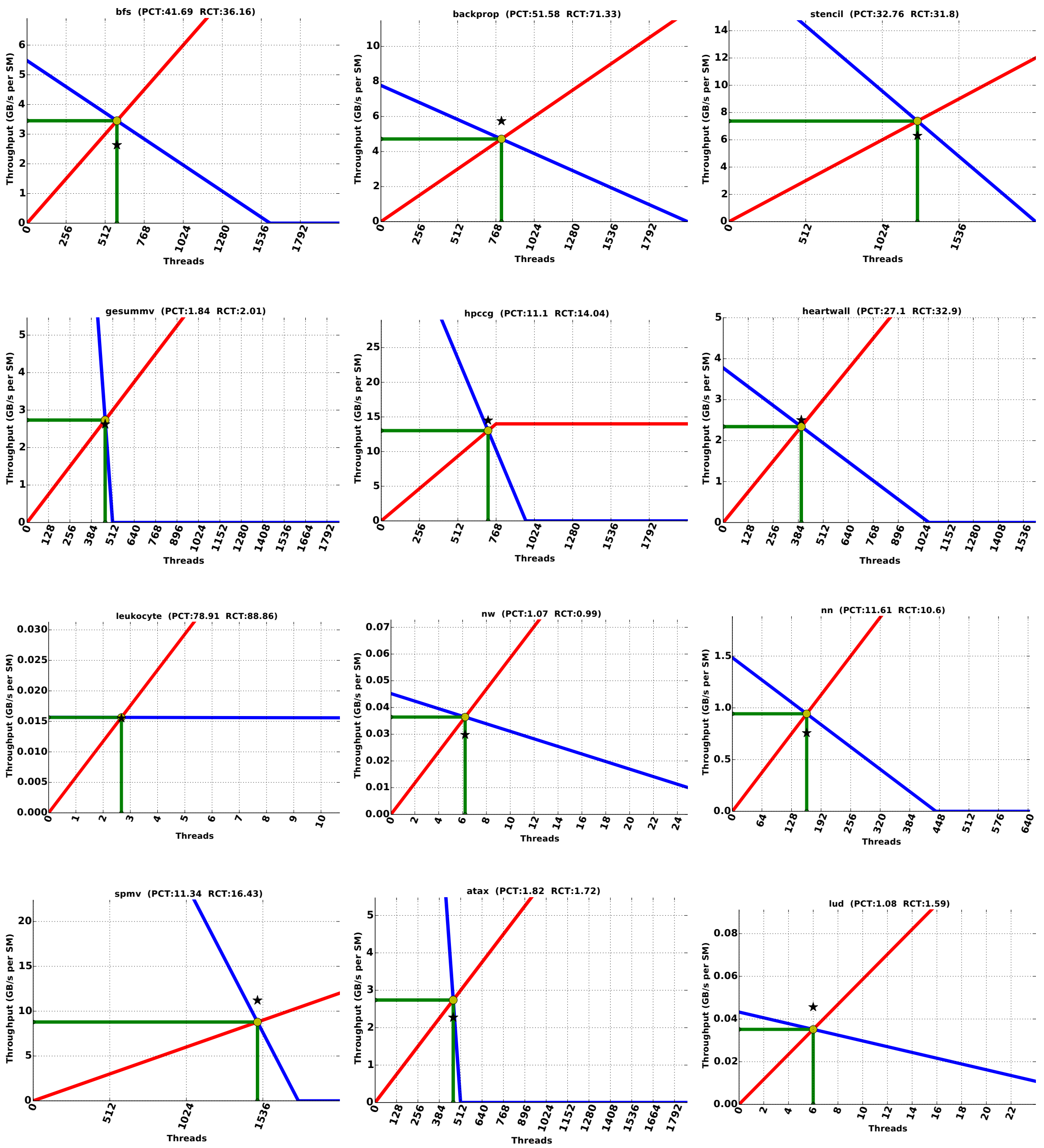

Figure 3.11: Validation Results on Kepler Platform.

As can be seen, for most of the applications, the dark star (measured memory throughput) is quite near the intersection (predicted by the X-model). Note that for SP scenarios, MS saturates at 2048 threads (64 warps), which is also the the maximum allowable threads per SM. This explains the linear behavior of $f(k)$ in most applications. `hpccg` is a DP application. Overall, using the computation throughput (PCT and RCT in Figure 11) as the metric, our model achieves 84.1% prediction accuracy. Consider only three parameters are extracted from the application, this is already quite accurate. The major factor that may impact the accuracy, is believed to be the coalesced memory access, as we do not count the coalesced access effect of MS.

3.6 Case Study

In this section, we show an example on how to leverage the X-model for evaluating different performance optimization options for real applications. We use a memory-intensive benchmark named `gesummv` from Polybench [131] as the target kernel. The platform we take for showcasing is Fermi GTX570, shown in Table 3.2. Note that this case study is to show the usage of the X-Model in detail; the general guideline is the same for other applications and platforms. Initially, 16 warps, equivalent to 512 threads, are allocated per thread block, which means all the 48 warp-slots per SM are fulfilled (with three thread blocks). The occupancy is 1. Besides, 16KB L1 cache on each SM is allocated by default.

To accurately reflect the present machine state for `gesummv`, we draw its X-graph based on the method discussed in Section 3.4. As shown in Figure 3.12, the isolated yellow points are the trace-points of $f(k)$ profiling via the bypassing approach in [87]. The green curve is the plot of $f(k)$ generated by connecting and smoothing these trace-points. We can observe that $f(k)$ and $g(x)$ intersect at the dropping slope of $f(k)$, which indicates that the L1 cache is thrashing currently and the machine shows a suboptimal performance. Under this thrashing condition, an intuitive tuning approach is to increase the L1 cache size, as discussed in Figure 3.8-B. Figure 3.13 shows the new X-graph in which the L1 is increased from 16KB to 48KB. However, very limited performance gain is observed after such tuning (only about 0.1GB/s MS throughput gain). The intersection is still at the dropping slope of $f(k)$, which indicates that the reason behind such poor performance improvement is not that the application is cache insensitive, but because the cache thrashing condition is still severe due to resource contention (e.g., limited MSHRs and miss queue entries) or bad data locality. However, compared with the 16KB L1 scenario (Figure 3.12), the cache peak of 48KB L1 in Figure 3.13 is much higher, which also implies that: (1) If the cache thrashing can be effectively resolved (e.g., via cache bypassing), the achievable performance can be much higher. In other words, the potential performance can be increased by reducing capacity misses through larger cache. (2). Our cache enlarging operation in Figure 3.12 is correct. The X-model here highlights its first usage: **investigating machine states and identifying the limiting factors for performance**.

To further improve the performance of the scenario shown in Figure 3.13, we generate other tuning

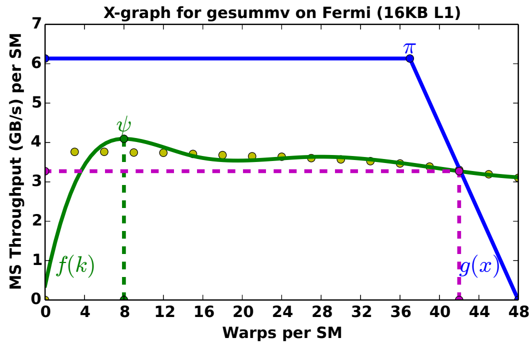

Figure 3.12: The X-graph for `gesummv` on Fermi with default 16KB L1 and 48 warps.

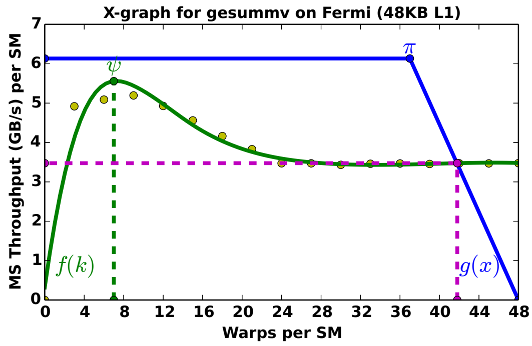

Figure 3.13: The X-graph for `gesummv` on Fermi with 48KB L1 cache size and 48 warps.

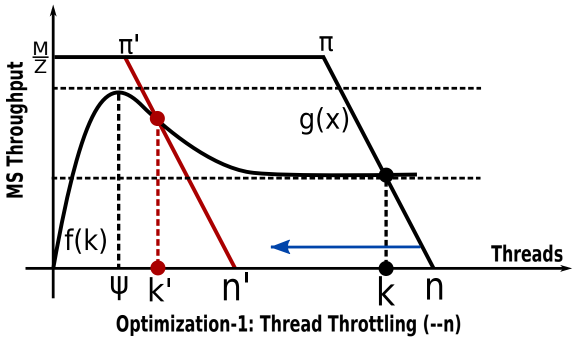

Figure 3.14: Thread throttling is to limit the number of threads in the machine so n drops to n'. As the intersection goes up while Z is unchanged, based on Principle 2, both CS and MS performance increase.

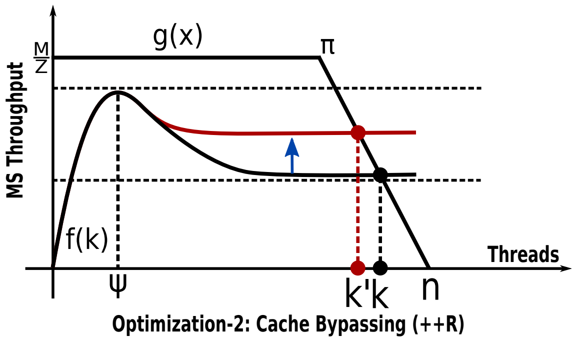

Figure 3.15: Cache bypassing is to mitigate cache trashing while keeping sufficient threads to exploit the MLP of the lower memory. With Z being unchanged, both CS and MS performance increase.

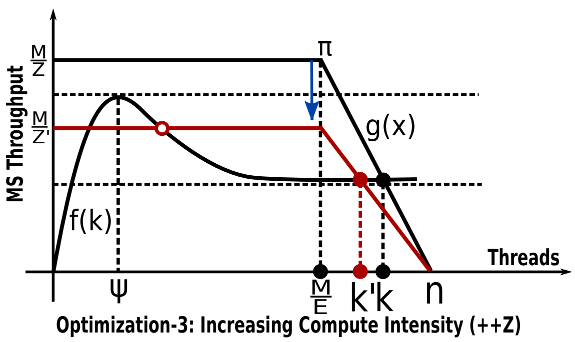

Figure 3.16: Increasing compute intensity (Z) or DLP. As Z increases and the intersection goes up slightly, with Principle 3, CS throughput is enhanced but MS throughput improves scarcely. As CS throughput is the primary metric, the machine performance increases.

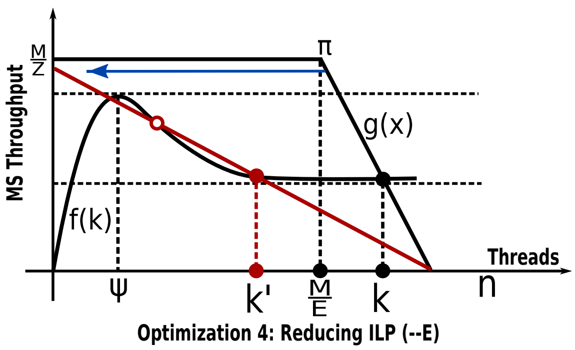

Figure 3.17: Reducing ILP (E). As the intersection goes up and Z keeps unchanged, based on Principle 2, both CS and MS performance increase. The circle marks the unstable interaction.

approaches by evaluating each model-tuning operation illustrated in Figure 3.4 and Figure 3.8, with the intention of increasing CS/MS throughput. After eliminating the ones that cannot improve CS/MS throughput under this thrashing condition (e.g., manipulating computation lanes M), we propose four optimization strategies for `gesummv`: *thread throttling* (Figure 3.14), *cache bypassing* (Figure 3.15), *increasing compute intensity* (Figure 3.16) and *reducing ILP* (Figure 3.17). They correspond to the operations of decreasing n (Figure 3.4-C), increasing R (Figure 3.4-A), increasing Z (Figure 3.4-D) and decreasing E (Figure 3.4-E), respectively. Here, we show the second usage of the X-model: **deriving and selecting the potential optimization approaches.**

Thread throttling [134, 99] is to restrict the number of concurrent threads on a SM to adapt the cache capacity or memory bandwidth [125]. Cache bypassing [135, 87] is to keep a limited number of threads accessing the cache while others bypass the cache to a lower memory hierarchy (in our case, bypass L1 to L2). Note, with proper cache bypassing, the cache is continuously contributing effective throughput. Therefore, the final exhibited throughput (throughput in the plateau of $f(k)$ is larger than original off-chip bandwidth R. Although both techniques are demonstrated to be effective for cache thrashing in various existing work, the explanation on when specific techniques would achieve the most performance gain as well as when they are going to fail, is unknown. The X-graphs in Figure 3.14 and Figure 3.15, however, can help us explain these directly. They show that the intersection goes up in both graphs under thread throttling and cache bypassing. The best performance is achieved when $g(x)$ coincides with the cache peak ψ in Figure 3.14 and when R rises to the same level as the cache peak in Figure 3.15. Eventually, further thread throttling or bypassing beyond the cache peak will start to degrade the performance again. Here, we show the third usage of the X-model: **reasoning and bounding the effectiveness of a technique.**

Furthermore, compared to thread throttling and cache bypassing, the two much less obvious tuning options are illustrated in Figure 3.16 and Figure 3.17. Figure 3.16 shows that although increasing Z can enhance the CS throughput for `gesummv` (as Z is increased, based on Principle 3, CS throughput is increased), the improvement for MS throughput is very limited (i.e., the height difference between the two intersections is tiny). Note that the Z value of an application is mostly decided by its algorithm. Therefore, to increase Z, algorithm modification is often required. Figure 3.17 shows something very interesting that has not been explored by any existing literature as a performance tuning method: *reducing ILP level (E) of an application can potentially increase the MS and CS throughput under cache trashing effect*. We leave the exploration on this new observation from our X-model as the future work. Nonetheless, we show the last usage of the X-model here: **exploring new opportunities for performance improvement**.

Finally, shown in Figure 3.18, we validate the tuning approaches suggested by the X-model above, including larger cache size, thread throttling and cache bypassing on a GPU hardware. We also show the performance of disabling L1 as a reference. The performance results are normalized to the default condition with 16KB L1 cache. Overall, using 48KB L1 cache achieves 7% speedup; thread throttling achieves 8% and 26% speedup for 16KB and 48KB L1 scenarios respectively; and cache bypassing achieves 22% and 36% speedup under two cache sizes respectively. These figures demonstrate that the tuning approaches offered through the X-model are effective with regard to performance optimization for a real parallel machine.

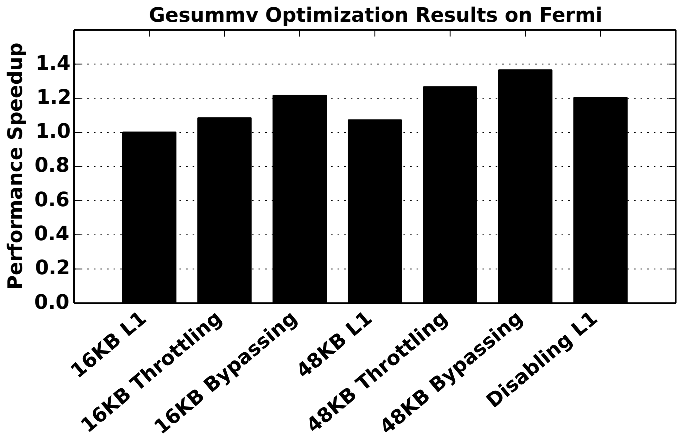

Figure 3.18: Validation of the tuning insights provided by the X-model.

3.7 Related Work

In this section, we discuss three existing analytic models that are widely-known and related to our X-model: the roofline model [128, 136], the valley model [137, 138], and the MWP-CWP model for GPUs [139, 133].

Roofline Model: The roofline model [128, 136] draws a roofline-like figure to show the variation of machine throughput with respect to the arithmetic intensity of the workload, which is essentially the relative relationship between DLP of the workload and DLP of the machine (i.e., Z and M/R). Both models aim at providing a visualizable and intuitive throughput model. However, the X-model is significantly different in three aspects. First and most important, the roofline model is generally for **sequential machines** and only addresses the influence of Z. The X-model, however, is for **parallel machines**. We address the impacts from various types of parallelism including ILP, TLP, MLP and DLP. Second, the roofline model is constructed based on **bottleneck analysis** whereas the X-model is built upon **flow balancing**. The roofline model is basically **static** for a certain machine, and by profiling Z of a workload, users can decide if the workload is memory-bound or computation-bound. The X-model, however, tracks the spatial state of the machine with a specific workload, which is the equilibrium between CS and MS. Any change of the parameters leads to the variation of the X-graph. Therefore, the X-model is **dynamic**. Finally, the X-model is much more **flexible** than the roofline model. In the roofline model, there is only one curve representing both MS and CS. In our X-model, we separate the MS curve from the CS curve so that each of them can be profiled, varied and analyzed independently. Therefore, X-model makes it possible to investigate more complex architectures (e.g., with complicated cache effects) by replacing $f(k)$ and $g(x)$ with more sophisticated and accurate shapes.

Valley Model: In [137, 138], Guz et. al. proposed an analytic model to describe the interaction between thread volume and shared cache for a multithreaded-manycore machine. Specially, they identified a *performance valley* between the cache efficiency zone and multithreaded efficiency zone for applications showing super-linear degradation of the hit rate with increased threads.

Although our modeling process for the cache effects in Section 3.3-B is analogous, the X-model itself is dramatically different. First, the valley model assumes that MS always remains the major bottleneck of the machine. We do not have this assumption so that factors such as ILP degree (E) can affect the cache performance, as discussed in Section 3.6. Second, the valley model assumes that allocated threads in the machine (i.e., n) share the cache storage. However, we argue that in the steady state of a parallel machine, within a certain time interval, only a fraction of the threads (MS threads) are essentially accessing MS. Therefore, the cache sharing should be only among these MS threads (k) instead of all threads of the machine (n), as reflected in Equation 3.3. Third, the memory latency in the valley model is fixed. That is why they introduced a bound from the CS part. In our X-model, the memory latency is changeable as the overall throughput is less than R. Finally, the CS and MS threads in the valley model are combined. The model focuses on their **joint effect** based on the MS bound assumption. As a comparison, the X-model separates the parallel machine into two curves and concentrates on their **relative effect**. Therefore, the X-model can offer more insights like the instable equilibrium and the sharp performance degradation discussed in Section 3.3-D.

MWP-CWP Model: MWP-CWP model [139, 133] is proposed to model execution time for GPUs specifically. It involves complex architectural level parameters and requires the support of simulation tools and PTX code, and it lacks the flexibility to play "what-if" scenarios for evaluating the effectiveness of different optimization techniques. Our X-model eliminates the “only GPU" part, so that it can be applied for general parallel machines. Although the intention of our model is to provide high-level evaluation for the present state of a parallel machine and propose useful intuition for optimizations, it can also be extended for execution time prediction if needed.

3.8 Conclusion

In this chapter, we propose a performance model named “X", which is a high-level and visualized analytic model for general parallel machines. Based on the spatial state of the machine, the X-model is able to comprehensively investigate the combined effects of various types of parallelism and the complex cache effects. With the model, developers and architects can easily draw an X-graph to identify performance bottlenecks, discern potential optimizations and derive novel intuitions.

CHAPTER 4

GPU Register Optimization: *Critical-Points Based Register-Concurrency Autotuning*

The unprecedented prevalence of GPGPU is largely attributed to its abundant on-chip register resources, which allow massively concurrent threads and extremely fast context switching. However, due to on-chip memory size constraints, there is a tradeoff between **per-thread register usage** and **overall thread concurrency**. This becomes a design problem in terms of performance tuning, since the performance "sweet spot" which can be significantly affected by these two factors is generally unknown beforehand. In this chapter, we propose an effective autotuning solution to quickly and efficiently select the **optimal number of registers per-thread** for delivering the best GPU performance. Experiments on three generations of NVIDIA GPUs (Fermi, Kepler and Maxwell) demonstrate that our simple strategy can achieve an average of 10% performance improvement, with a max of 50%, over the original version. Additionally, to reduce local cache misses due to register spilling and further improve performance, we explore three optimization schemes (i.e., bypass L1 for global memory access, enlarge local L1 cache and spill into shared memory) and discuss their impact on performance on a Kepler GPU. This work has been presented at Design, Automation and Test in Europe Conference (DATE-16) [86].

4.1 Introduction

The extraordinary emergence of general-purpose Graphic Processing Units (GPGPUs) is well-known for their massive thread-level-parallelism (TLP). To accommodate such an amount of active threads, GPUs have to encapsulate large register files. Moreover, to mitigate the negative impact from the memory-wall, GPUs adopt the "*latency hiding*" technique by keeping the contexts of all the active threads in the register files, which enables fast switch when stalls are encountered. Although the GPU register files are quite large compared to those on CPUs, such utilization can still impose great pressure on them. As the limited registers are evenly distributed among the active threads, the performance tradeoff between the per-thread register consumption and the overall concurrency appears: for the applications that are bounded by the limited register resource, although more registers per thread indicate superior single-thread performance without register spills, fewer registers per thread could increase concurrency, which may eventually result in aggregated performance improvement.

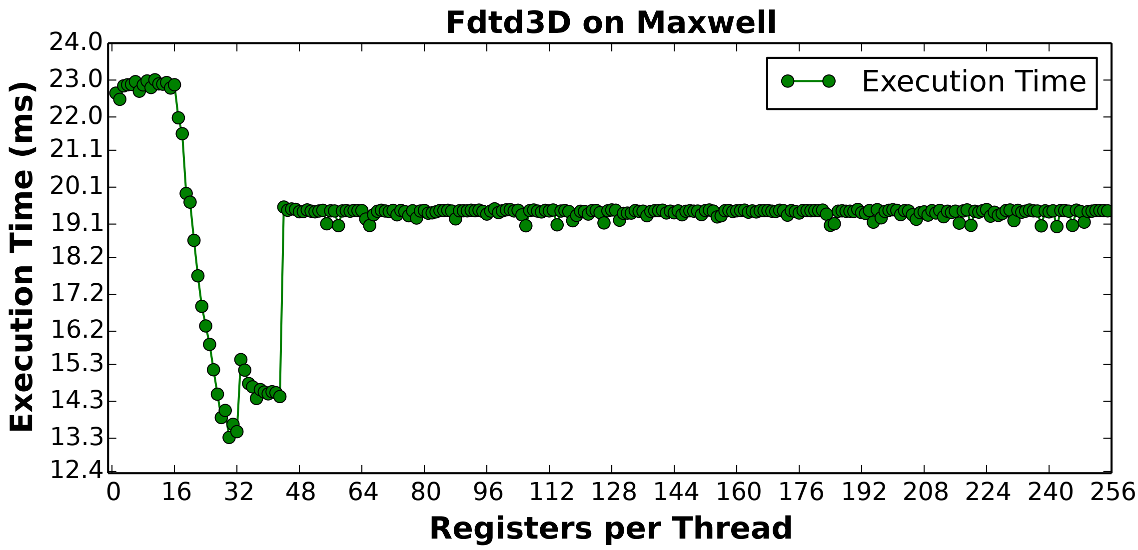

Figure 4.1: Profiling for different register number for Fdtd3d on a Maxwell GPU.

Therefore, finding the optimal per-thread register usage that delivers the best performance becomes an important issue for GPU software developers. Efficient register usage management is also considered as one of the biggest remaining issues of the current CUDA toolchain [140].

Figure 4.1 shows an example to explain the problem. It shows the execution time of *Fdtd3D* with respect to per-thread register usage for a Maxwell GPU. On the left, the execution time decreases with a higher register utilization. However, the curve is interrupted at $r = 32$ and $r = 43$ with a sudden and dramatic increase. The task is to find the r that corresponds to the shortest execution time. Although in this example, it is obvious that $r_{opt} = 32$, it is impractical to determine such a figure for every application we study, since the register range can be very large (e.g., 255 for Maxwell GPUs) and the position of the optimal point may also be input-dependent. Furthermore, not all applications show such an ideal curve, as will be seen later. Therefore, the problem is how to find an effective way to shrink the search space for r_{opt} and then efficiently locate it.

This chapter makes the following contributions:

- We study the underlying relationship between register count, concurrency and performance, based on which we propose the idea of critical-points (Section 4.3).
- We propose an efficient autotuning scheme to find the optimal register usage per thread. It is tractable, effective, and general for benefiting all GPU generations (Section 4.3).
- We explore three optimizations to further improve performance and reduce local cache conflicts due to register spills (Section 4.5).

4.2 GPU Thread Organization and Local Memory Access

In this section, we briefly review the GPU **thread organization** and the **local memory** access. A GPU kernel, which is a device function executed on the GPU hardware, contains thousands or tens of thousands concurrent threads that are primarily partitioned into multiple *thread blocks* (CTAs). When a kernel is launched, all the CTAs are distributed the streaming multiprocessors (SMs). It is possible that several CTAs are distributed to the same SM simultaneously, depending on the size of SM on-chip resources, such as the registers and the scratchpad memory (i.e., shared memory). These resources are evenly divided among the concurrent CTAs. The threads of a CTA are further grouped into a number of execution vectors, called *warps*, that perform the same operations on different data in a lockstep manner. A warp is the basic unit for instruction issuing, executing, L1 cache access and so on.

In addition to the register file, a GPU thread has several types of memory to access, including global (off-chip, the GPU main memory, L1 & L2 cached), local (off-chip, L1 & L2 cached), shared (on-chip, shared in a CTA), texture (on-chip, read-only and cached) and constant (on-chip, read-only and cached). The local memory is not actually a physical memory but rather an abstraction of the global memory. Its scope is thread-private, the same as for the register file. It is generally used for temporal spilling when there are not enough registers to hold all the required variables or the arrays that are declared inside the kernel but the compiler cannot resolve the indexing. It is also L1- and L2-cached, for both read and write. Register spilling in local memory may hurt the performance as it introduces extra instructions and memory traffic, especially when spilling results into extra cache misses.

4.3 CP-based Autotuning Method

In this section, we present our critical-points (CP) based autotuning method. We call it "auto" because the entire tuning process can be accomplished automatically without user intervention. All the required information can be extracted from the output of the compiler and the profiler. This method is based on the following key observations:

1. On the one hand, a GPU kernel requires a minimum number of registers to be successfully compiled (i.e., the lower bound of the register usage: r_{min}). On the other hand, a GPU kernel needs a maximum number of registers so that all the intermediate data are located in the registers (i.e., the upper bound of the register usage: r_{max}). Beyond r_{max}, allocating more registers is wasteful.
2. For a single GPU thread, more registers contributes to spill reduction and locality exploitation. Therefore, more registers could lead to better single thread performance.
3. For the massive TLP on GPUs, the concurrency (i.e., number of active threads) may *impact* performance significantly. Although more threads normally lead to better latency hidding and

pipeline utilization thereby a higher performance, it is not always true under certain scenarios: if a subsystem is already saturated (e.g., scalar processors are fully leveraged by exploiting the instruction-level-parallelism [103]), adding more threads brings no further performance gains. Even worse, excessive threads to an overloaded system could lead to dramatic conflicts and contention, degrading the overall performance [125].

Obviously, there is a performance tradeoff between register usage per thread and concurrency: *can the benefits from higher concurrency (i.e., fewer registers assigned to each thread) offset the drawbacks from register spills?* To answer this question, we first discuss the relationship between register usage and performance. We denote r as the number of registers per thread, and based on observation (1) we have

$$r_{min} \leq r \leq r_{max} \tag{4.1}$$

We label this region $[r_{min}, r_{max}]$ as the *Register Effective Region* (**RER**). Based on observation (2), with a larger r, more spill loads and stores are avoided, which contributes to a higher performance. If we use $g(r)$ to denote the performance function with respect to the per-thread register count, then

$$P = g(r) \propto r \tag{4.2}$$

Note that $g(r)$ is continuously increasing as every one more register eliminates a fraction of spills until all spills are eliminated.

Now let us turn to concurrency and explore why the change of r can lead to concurrency drop. Since the cost of registers per CTA is fixed, **the only factor that can directly impact concurrency is the maximum number of CTAs that can be dispatched simultaneously on an SM at runtime**. This CTA number is limited by the hardware restrictions and availability of on-chip resources, one of which is the amount of registers. Therefore, if we use w to denote the number of warps per CTA, then the number of CTAs that can be dispatched simultaneously on an SM is:

$$\begin{aligned} N_{CTA/SM} &= min\{\frac{All\_CTAs}{SMs}, \quad N_{max\_CTAs/SM}, \quad \lfloor\frac{N_{warps/SM}}{w}\rfloor, \\ &\lfloor\frac{N_{regs/SM}}{\lceil\frac{N_{regs/CTA}}{unit_{reg}}\rceil * unit_{reg}}\rfloor, \quad \lfloor\frac{N_{smem/SM}}{\lceil\frac{N_{smem/CTA}}{unit_{smem}}\rceil * unit_{smem}}\rfloor\} \end{aligned} \tag{4.3}$$

The five terms in the function are the total number of CTAs per SM (CTAs of the kernel/SM number), GPU restricted amount of CTAs per SM, GPU restricted amount of warps per SM, register limitation, and shared memory limitation per SM. The ceiling in the last two items are because a GPU allocates registers/shared memory to CTAs by a unit size, which is 64/128B for Fermi and 256/256B for both Kepler and Maxwell. In general, a kernel includes thousands of CTAs, so the first term is very large. $N_{max\_CTAs/SM}$ is 8 for Fermi, 16 for Kepler and 32 for Maxwell. If we assume that the shared memory is not the bottleneck, then the formula becomes:

$$
\begin{aligned}
N_{CTAs/SM} &= min\{\lfloor \frac{N_{warps/SM}}{w} \rfloor, \lfloor \frac{N_{regs/SM}}{\lceil \frac{N_{regs/CTA}}{unit_{reg}} \rceil * unit_{reg}} \rfloor, N_{max\_CTA/SM}\} \\
&= min\{\lfloor \frac{N_{warps/SM}}{w} \rfloor, \lfloor \frac{N_{regs/SM}}{\lceil \frac{32*w*r}{unit_{reg}} \rceil * unit_{reg}} \rfloor, N_{max\_CTA/SM}\}
\end{aligned}
$$

in which $N_{warps/SM}$, $N_{regs/SM}$, $unit_{reg}$ and $N_{CTA/SM}$ are constants while w is predefined by the application. The only variable left in the equation is the register number (r). If we use $f(concurrency)$ to denote the performance function corresponding to concurrency, then

$$
P = f(concurrency) = f(N_{thds/CTA} * N_{CTA/SM}) = f(N_{thds/CTA} * \lfloor \frac{N_{regs/SM}}{\lceil \frac{32*w*r}{unit_{reg}} \rceil * unit_{reg}} \rfloor) \tag{4.4}
$$

Based on observation (3) that a higher concurrency in general contributes to a better performance, we have

$$
P \propto 1/r \tag{4.5}
$$

By observing Eq (4.2) and (4.5), there is a clear conflict or tradeoff. It is possible to use the X-model proposed in Chapter 3 to analyze this register-related performance tradeoff. As shown in Figure 4.2-(A), on the one hand, increasing the register number per thread (r) leads to the reduction of CTAs per SM, or the decreasing of threads (n). As a result, the intersection point drops, both the CS and MS throughput decrease. On the other hand, with a higher register number per thread, some intermediate data or operands that are originally spilled in the local memory or loaded from the shared/global memory can now be temporally cached in the registers, so as to exploit the data's temporal locality. Consequently, fewer memory requests are required and the compute intensity (Z) is increased. As shown in Figure 4.2-(B), the intersection point drops. However, as Z increases and π is on the left of the intersection point, the CS throughput increases (Principle 3 in Section 3.2). Combining Figure 4.2-(A) and Figure 4.2-(B), we obtain the resultant X-graph depicting the tradeoff, as shown in Figure 4.2-(C). As increasing Z in Figure 4.2-(A) leads to CS throughput improvement while decreasing n in Figure 4.2-(B) leads to CS throughput degradation, these two effects are opposite. The final performance is a combination or tradeoff of the two: if ultimately there are more threads (x) entering CS (i.e., $\overline{n'k'} > \overline{nk}$), the performance will improve; otherwise, the performance degrades.

However, in reality the changing of n is not continuous (as shown in Figure 4.2-(A)), unlike $g(r)$, the correlation between $f(concurrency)$ and r shows only a few discrete steps due to the *floor()* function in Eq. 4.4. In fact, with the *floor()* function, an increment of r does not necessarily lead to a decrement of $\lfloor \frac{N_{regs/SM}}{\lceil \frac{32*w*r}{unit_{reg}} \rceil * unit_{reg}} \rfloor$. But once the increment of r triggers a drop of $\lfloor \frac{N_{regs/SM}}{\lceil \frac{32*w*r}{unit_{reg}} \rceil * unit_{reg}} \rfloor$, the concurrency degrades by a significant factor of $N_{thds/SM}$. We label the last points (i.e., register usage) before the drops as the **critical-points** (CPs). These significant changes in concurrency may lead to drastic variations in performance, which forms a series of stages (we label them **concurrency levels**).

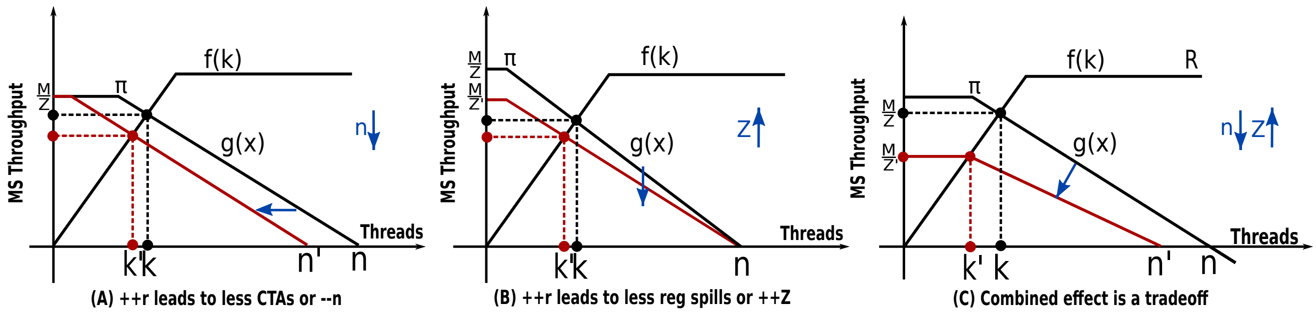

Figure 4.2: The Register-Concurrency Tradeoff Analyzed by X-Model.

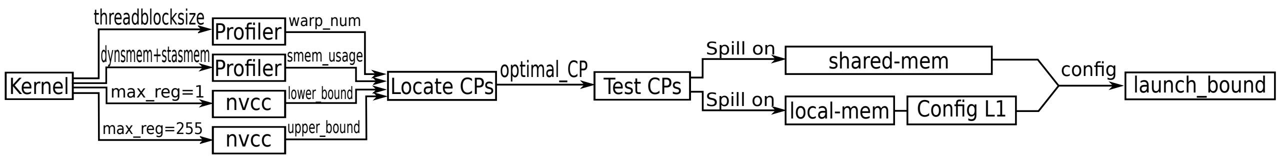

Figure 4.3: Autotuning Framework

Table 4.1: Experiment Platforms. Dri/Rtm means the CUDA driver version and toolkit version. M(CTAs) indicates the maximum allowable number of thread blocks per SM. M(Thds) is the maximum number of threads per SM. M(Regs/Thd) is the maximum number of registers per thread. Shared+L1 is the volume of shared memory and L1 cache per SM.

GPU	Arch	Dri/Rtm	SMxSP	M(CTAs)	M(Thds)	Regs	M(Regs/Thd)	Shared+L1
GTX570	Fermi-2.0	6.5/6.5	15x32	8	1536	32K	63	(48+16)KB
Tesla K40	Kepler-3.5	6.0/6.0	15x192	16	2048	64K	255	(48+16)KB
GTX750Ti	Maxwell-5.0	6.5/6.5	5x128	32	2048	64K	255	(64+0)KB

Such a performance curve is the result of a typical combination of effects from $g(r)$ and $f(1/r)$.

Therefore, the basic idea for the CP-based autotuning is the following: In the range of RER, different concurrency levels separate the performance curve with respect to the register count into several regions. Within each region, the performance at the CP is likely the optimal or very close to the optimal (see next section for details). Since a different concurrency level impacts performance but not necessarily leads to a better performance, we need to evaluate all the CPs to locate the global optimal in the autotuning process.

Our proposed autotuning framework is shown in Figure 4.3. First, we need to decide the boundaries of RER. This information can be extracted from the GPU compiler (e.g., *nvcc*) when passing the *-maxrregcount=1* and *-maxrregcount=max_reg_per_thd* (the value shown in Table 4.1) flag respectively, since the corresponding compiler decides this default boundary information for applications on different GPU architectures. We then profile the kernel to acquire the warp number and shared memory usage per CTA. Together with the hardware information, we are able to locate the CPs for a specific application based on Eq. 4.3. After that, the framework tests the performance of each CP and reports the optimal.

Table 4.2: Experiment Applications. U/L/D() indicates the upper-bound, lower-bound and default value of registers per thread on a specific architecture. F, K, M stand for Fermi, Kepler and Maxwell respectively.

Application	Abbr.	Kernel	Warps	Shared	U/L/D(F)	U/L/D(K)	U/L/D(M)	Source
cfd	`CFD`	cuda_compute_flux()	8	0	62/16/62	74/16/68	75/16/70	Rodinia[37]
hotspot	`HOT`	calculate_temp()	8	3072B	35/16/35	38/16/38	36/16/35	Rodinia[37]
leukocyte	`LEU`	IMGVF_kernel()	10	14586B	61/16/52	61/16/61	63/16/63	Rodinia[37]
myocyte	`MYO`	solver_2()	1	0	63/16/63	220/16/149	225/16/133	Rodinia[37]
nbody	`NBO`	integrateBodiesIf	8	4096B	63/16/24	252/16/38	255/16/37	SDK[42]
particles	`PAR`	collideD()	8	0	51/16/51	52/16/52	52/16/52	SDK[42]
ray-tracing	`RAY`	render()	4	0	51/16/50	55/16/49	56/16/56	SDK[42]
dxtc	`DXT`	compress()	2	2048B	63/16/63	90/16/89	93/16/90	SDK[42]
fdtd3d	`FDT`	FiniteDifferencesKernel()	16	3840B	55/16/45	50/16/40	53/16/45	SDK[42]
dct8x8	`DCT`	CUDAkernel2IDC()	3	3136B	42/16/35	37/16/33	35/16/34	SDK[42]
mri-gridding	`MGR`	gridding_GPU()	2	1536B	62/16/56	62/16/62	60/16/59	Parboil[38]
sgemm	`SGM`	mysgemm()	4	512B	63/16/33	175/16/53	164/16/48	Parboil[38]

4.4 Validation

In this section, we validate the critical-points based autotuning method on three generations of GPUs: Fermi, Kepler and Maxwell. The platform information is listed in Table 4.1. We take 12 applications from the Rodinia [37], SDK [42] and Parboil[38] benchmarks, as listed in Table 4.2. We also show the number of warps and amount of shared memory allocated per CTA in each application to compute the CPs. As discussed in Section 4.3, the flags *–maxrregcount=1* and *–maxrregcount=255 (63 for Fermi)* are passed to the *nvcc* compiler to acquire the lower (r_{min}) and upper bound (r_{max}) for the register usage of an application. We also obtain the default register usage from the compiler as the "**Baseline**" for performance comparison. The results for Fermi, Kepler and Maxwell are shown in Figure 4.4, 4.5 and 4.6 respectively. "**Proposed**" is the performance achieved by CP-based autotuning. "**Optimal**" is the performance improvement upper-bound given by exhaustive searching. We also show the occupancy change, the register usage points that have to be searched and the geometric mean for performance improvement across all applications in the figure. As can be seen, our autotuning approach achieves 7.9%, 8.8% and 5.5% speedup on average for Fermi, Kepler and Maxwell GPUs over the baseline cases, while the optimal results reported by exhaustive searching are 9%, 10% and 7%, respectively. Compared with the baseline cases, our method reduces the search space for r_{opt} by a factor of 15x, 20x and 13x on geometric average.

One interesting observation is that not every application's occupancy increases after the optimization (e.g., `NBO` and `SGM`), which indicates that a higher occupancy does not necessarily lead to a better performance. It also confirms the necessity to evaluate each different concurrency level (i.e., each CP). Also note that `CFD` shows very different behaviors on the three architectures (i.e., `CFD` shows significant performance improvement on Kepler, but almost none on Fermi and Kepler).

To further explore why in certain applications the CP set cannot capture the optimal (e.g., `MYO` and `MGR` in Figure 4.5) and why in `NBO`, the performance of CP is even worse than the baseline, we plot the execution time with respect to register number and occupancy level for the 12 applications on the three platforms, as shown in Figure 4.7, 4.8 and 4.9, respectively. We also draw the curves for normalized spilled loads & stores reported by compiler and the local cache hit rate measured by

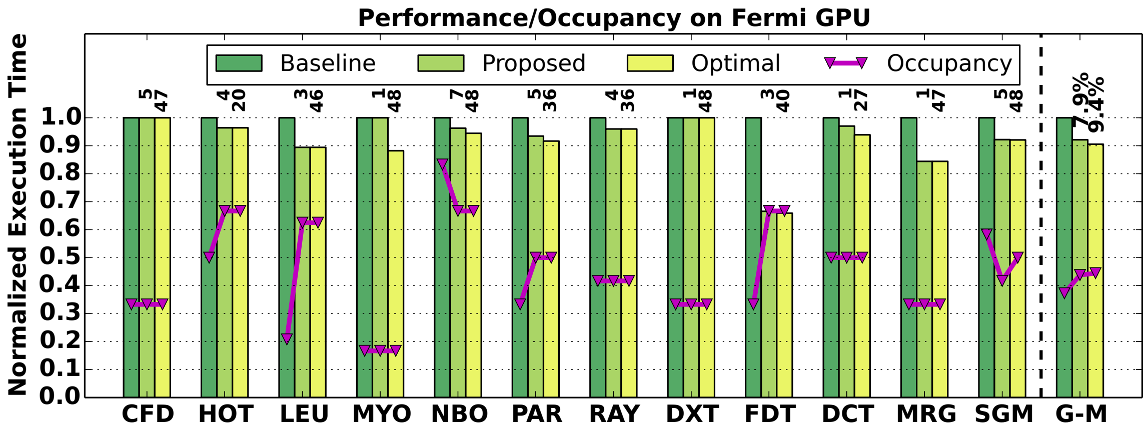

Figure 4.4: Execution time reduction on Fermi GPU. The rotated numbers on top of the application histograms indicate the size of search space.

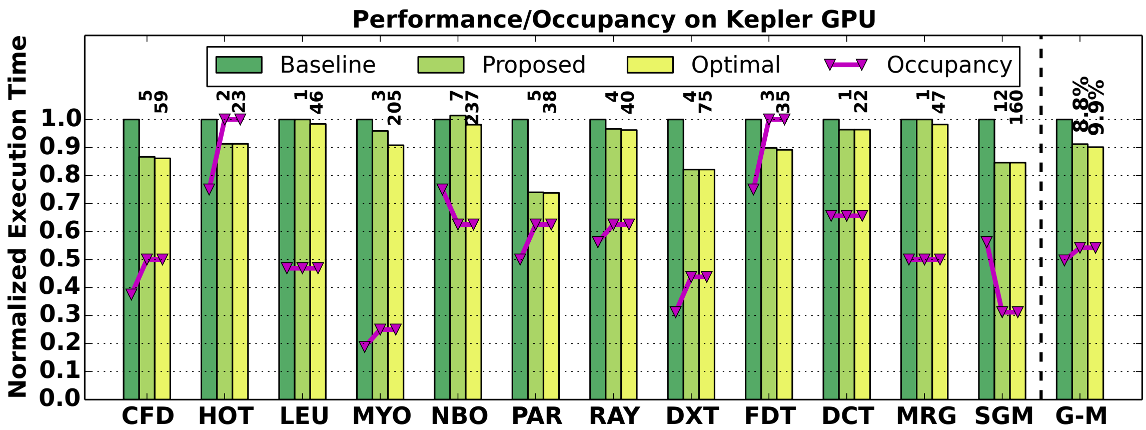

Figure 4.5: Execution time reduction on Kepler GPU.

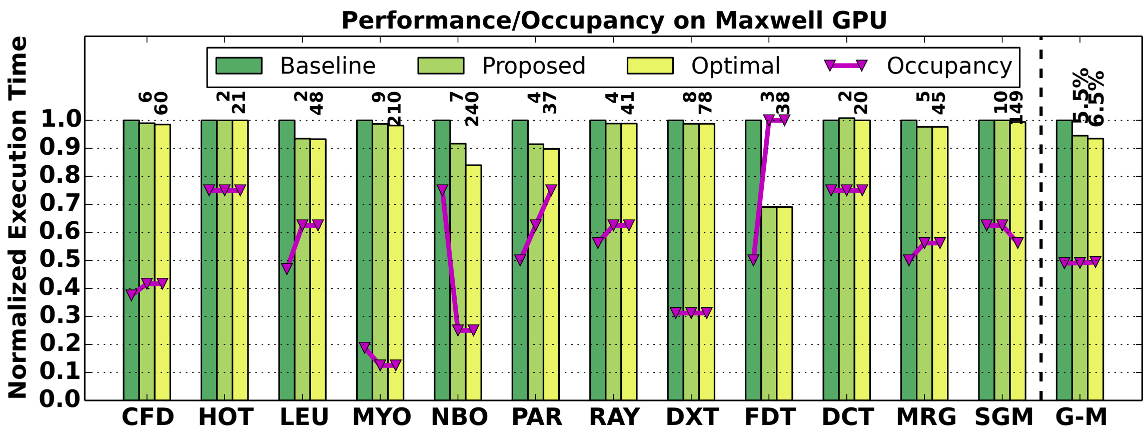

Figure 4.6: Execution time reduction on Maxwell GPU.

profiler. Using the Kepler results as an example, we have the following observations:

- Though we only plot the figures in the range of RER (using the lower- & upper-bound in Table 4.2), we can clearly observe that the point at which the spilled-load and store disappears (also the point where the local cache hit rate curve reduces to zero[1]) is always less than the upper-bound of RER. We call this point the *spill-disappear-point*. Although at this point, no spill occurs, there is still some rematerialization, because the compiler is able to reduce the register usage by recomputing the values of some intermediate variables based on the other registers. Such rematerialization incurs unnecessary computation overhead. Only beyond the RER upper-bound, all the intermediate data is stored in the registers, and there is neither spill nor redundant computation.

- The trend that execution time drops with more threads confirms the first observation. However, not all the applications are concurrency sensitive, e.g., `MYO` and `SGM`. Meanwhile, some applications such as `LEU`, `DCT` and `MRG` are limited by other on-chip resources, changing the register usage does not impact occupancy or concurrency. For example, `LEU` and `DCT` are limited by the shared memory usage. As each CTA in `LEU` requires 14586B shared memory space (see Table 4.2), 48KB shared memory can accommodate up to 3 CTAs. With 10 warps per CTA, the occupancy keeps constant at $3*10/64 \approx 0.47$. For `DCT`, each CTA consumes 3136B; 48KB thus is theoretically sufficient for 15 CTAs. However, as shared memory is allocated in a unit of 256B for Kepler (see Eq. 4.3 in Section 4.3), eventually only 14 CTAs are initiated per SM, which contributes to an occupancy of $14*2/64 \approx 0.44$. On the other hand, `MRG` is restricted by the maximum number of CTAs per SM (hardware limitation), which is 16 for Kepler (see Table 4.1). The occupancy thus stays at 0.5. From Kepler to Maxwell, as an SM supports more CTAs (from 16 to 32), we can observe that the occupancy changes as expected and the performance increases for `MRG` in Figure 4.6.

- The baseline point (i.e., the default register usage number imposed by the compiler) is neither the *spill-disappear-point* nor the upper-bound of RER. It is calculated by an unknown algorithm of the compiler. Additionally, the number of CPs for each application is generally around 5, which is much smaller than the RER range. The optimal point for performance is mostly captured by our approach for each application. The exceptions are `MYO`, `MGR` and `NBO` due to the dramatic performance oscillation within a concurrency level (especially `MGR` has only one concurrency level).

- Although in general the normalized spill LD&ST curves drop with increased number of registers until the *spill-disappear-point*, the curves for local cache hit rates are far more intractable. They commonly start at lower hit rate because there are many variables that have to be spilled due to significant shortage of registers. At the same time, a higher occupancy also implies more inter-CTA conflicts in the L1 cache. As more registers are allocated and fewer CTAs share the cache, the hit rate curve increases, and drops to zero at the *spill-disappear-point*

[1]The hit rate reduction here is actually not because of cache miss but no such local cache access due to zero register spilling.

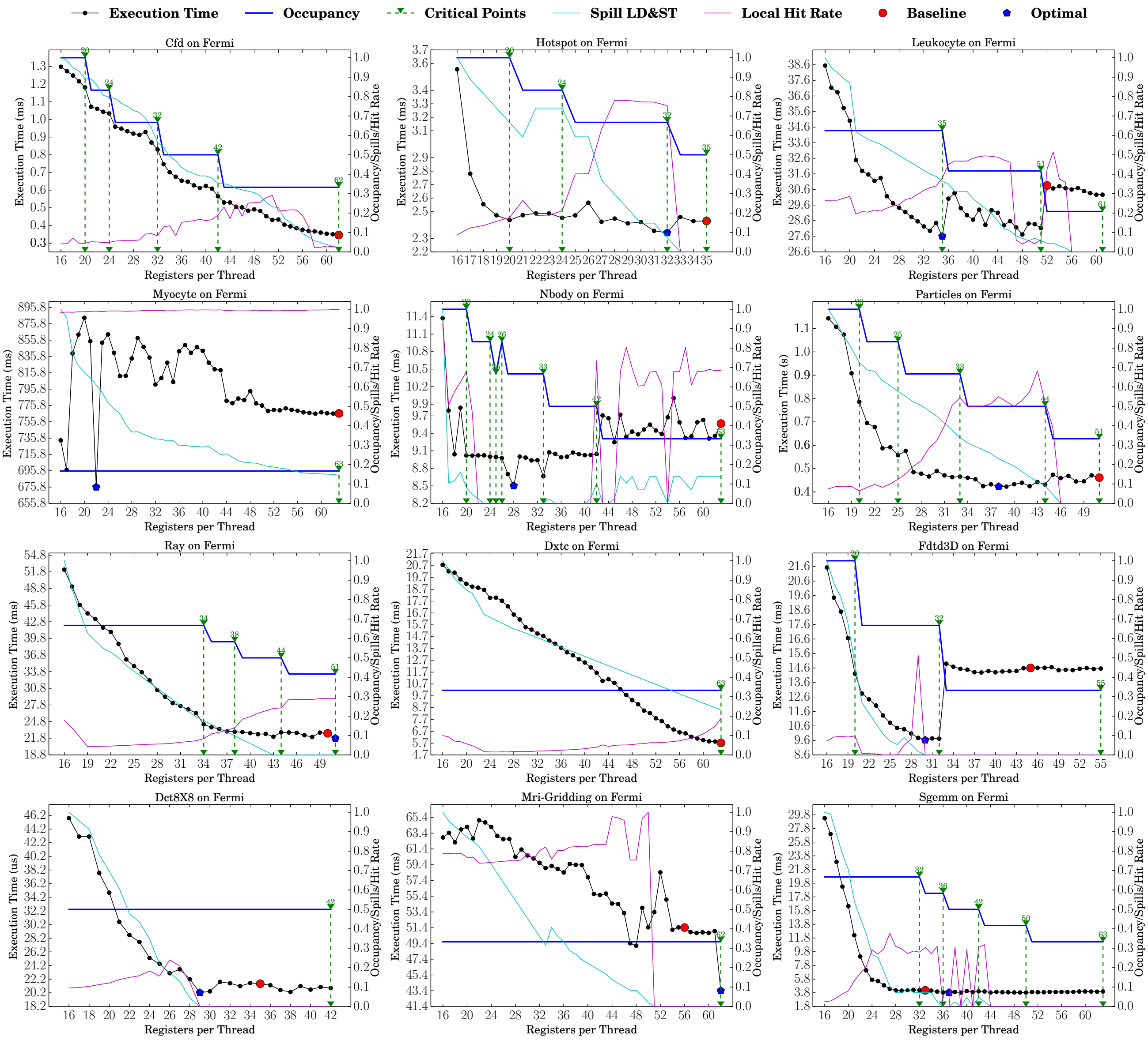

Figure 4.7: Detailed Application Profiling on Fermi GPU. Local hit rate is only for local cache hit rate of L1 not the total L1 hit rate.

because **there is no local memory access any more**[2]. Additionally, some steep fluctuation in `NBO` and `SGM` can be observed. This is because with different register numbers, the compiler algorithm may occasionally enforces some 4B to 16B local memory spills, which translate to a very high hit rates. Thus, the curves oscillate quickly and sharply within certain regions (e.g., register range between 90-110 for `SGM`). Also note that the local cache hit rates may suffer from global memory accesses, as they share the same cache storage.

[2]The hit rate curve drops to zero as there is no cache access. However, the underlying cache hit rate itself is not necessarily zero.

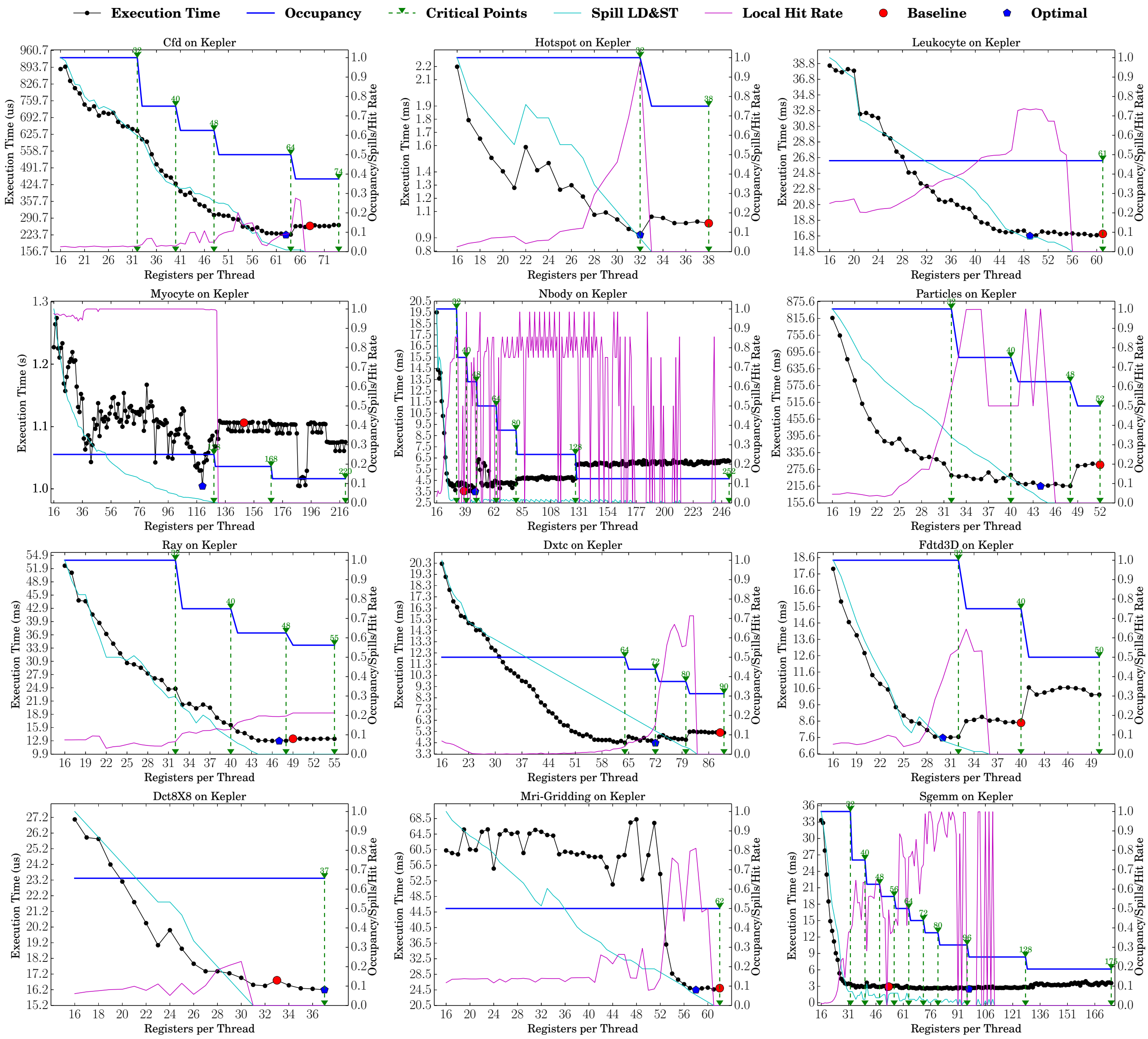

Figure 4.8: Detailed Application Profiling on Kepler GPU. Local hit rate is only for local cache hit rate of L1 not the total L1 hit rate.

4.5 Discussion

Take Kepler as an example (Figure 4.8), overall the local cache hit rates for the applications are not quite high. Possible reasons include compulsory misses (i.e., first-time spill), capacity misses (i.e., many registers from many active threads need to spill to a very small cache size of 16KB per SM), and conflict misses (i.e., shared by multiple CTAs and shared with the global accesses). To mitigate or even eliminate the latter two, we apply the following three optimizations:

- We configure a larger L1 cache (e.g., 32KB or 48KB, instead of 16KB) upon kernel invocation.
- We apply software-level strategies [141] to spill to the shared memory instead of the local memory .

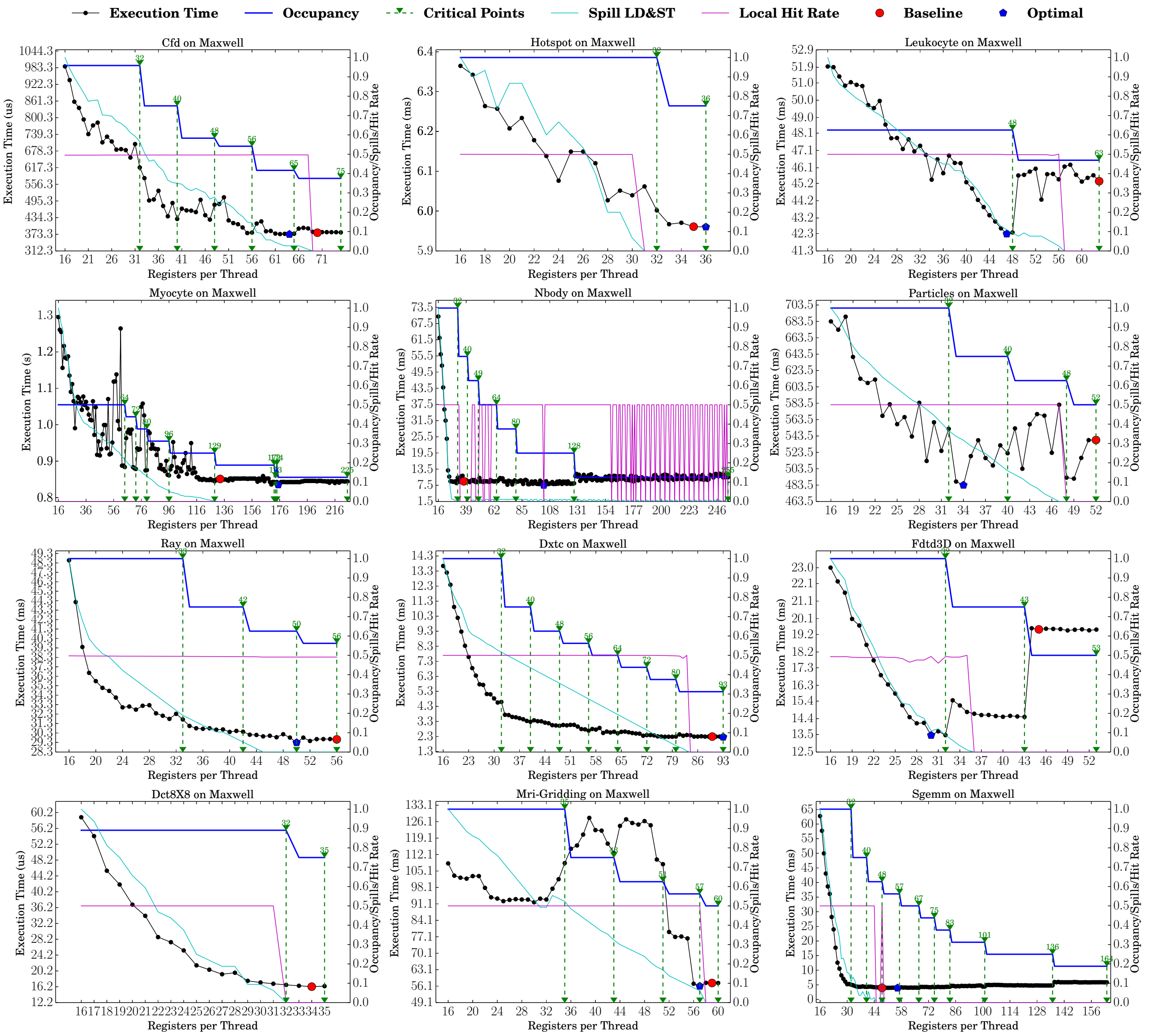

Figure 4.9: Detailed Application Profiling on Maxwell GPU. Local hit rate is only for local cache hit rate of L1 not the total L1 hit rate.

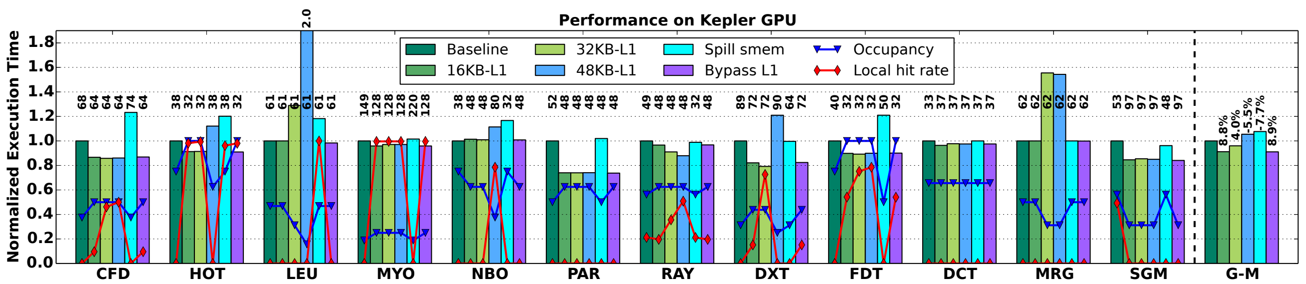

Figure 4.10: Test different L1 cache configurations, the design of spilling on shared memory and bypassing global access at L1 on Kepler GPU. The numbers on top of the histograms are the obtained register number by each scheme.

- We bypass the L1 cache to avoid possible conflicts from global memory access by setting "*-dlcm=cg*".

The results are shown in Figure 4.10. As can be seen, a larger L1 cache size enhances the local cache hit rate for `CFD`, `RAY`, `DXT` and `FDT`, which improves performance for `CFD`, `RAY` and `FDT`. The scenario for `DXT` is interesting, as a 32KB L1 increases performance but a larger 48KB L1 degrades performance drastically. This is because, although a 48KB entirely avoids L1 cache miss, the larger L1 cache capacity is achieved at the expense of a smaller shared memory (L1 cache and shared memory share the same storage in an SM). The reduced shared memory capacity limits the number of CTAs that can be allocated simultaneously per SM (see Eq. 4.3 in Section 4.3), which eventually degrades the concurrency and performance. Besides, spill on shared memory is not shown to be a good solution in our test, as it always delivers the lowest performance. Finally, bypassing global access does not impact local cache hit rate or performance (view that the time and local hit rate for "16KB-L1" and "Bypass L1" are the same); this is because on Kepler, all global memory access bypass L1 by default [10]. However, this is not the case for Fermi. In fact, we observed performance improvements for all applications except `MYO` on Fermi with L1 cache bypassed for global memory access.

4.6 Related Work

Previous work related to GPU register file mostly focuses on architectural improvement, seeking to reduce chip area and energy consumption [142, 143, 144, 145]. Gebhart et. al. [142] placed a small register cache on top of GPU's main register file so that the small register cache can filter a large portion of the accesses before going to the main register file. In this way, significant power consumption can be avoided. They also combined their register cache with a novel two-level warp scheduler for further energy reduction. Yu et. al. [143] integrated eDRAM into the SRAM based GPU register file to reduce energy. Later, Gebhart et. al. [144] combined register file, L1 cache and scratchpad memory of GPU as a unified storage space and dynamically tuned the partitioning among them. Recently, Lee et. al. [145] found that values written by threads in the same warp show great similarity therefore can be compressed to reduce power.

The work most related to ours is proposed by Hayes and Zhang [141]. Their work also concentrated on the tradeoff between register usage and concurrency while wrapped the on-chip scratchpad memory as a supplementary register file. A metric based on computation/memory interleaving degree is proposed to predict the best concurrency level at compile-time. However, their design is concurrency-centric. The calculation of the predicted concurrency (i.e., the metric) requires complicated parsing and analysis of the binary, while some of the input parameters are architecture-dependent and are very difficult to measure (e.g., the dispatch interval). Their work also presumes that local memory access is detrimental and should be completely eliminated. However, migrating the latency sensitive data from L1&L2-cached local memory to the shared memory with extra software management overhead may not be beneficial eventually (see Figure 4.10 in Section 4.5).

4.7 Conclusion

In this chapter, we proposed an autotuning approach to resolve the conflict between concurrency and register usage for GPUs. We discovered that the performance impact from register usage is almost continuous but from concurrency is discrete. The tradeoff between the two factors forms a special relationship such that a series of critical-points can be precomputed. These CPs denote the best performance of each concurrency level, and the global optimum is then selected among them. Our approach is **tractable**, **effective** and **general**. It leverages the existing features of the hardware and demonstrates immediate speedup for all three generations of GPUs over a dozen of real applications. The improvement is very close to the optimal one achieved by exhaustive search. Our method reduces the search space for the optimal register usage by up to 20x based on our observations and enhances the overall GPU performance, up to 1.5x. More importantly, our tuning method is fully automatic and can be easily integrated into the compiler or profiler.

CHAPTER 5

GPU Cache Optimization: *Adaptive and Transparent Cache Bypassing*

In the last decade, GPUs have emerged to be widely adopted for general-purpose applications. To capture on-chip locality for these applications, modern GPUs have integrated multi-level cache hierarchy, in an attempt to reduce the amount and latency of the massive and sometimes irregular memory accesses. However, inferior performance is frequently attained due to serious congestion in the caches resulting from the huge amount of concurrent threads. In this chapter, we propose **a novel compile-time framework for adaptive and transparent cache bypassing** on GPUs. It uses a simple yet effective approach to control the bypass degree to match the size of applications' runtime footprints. We validate the design on seven GPU platforms that cover all existing GPU generations using 16 applications from widely used GPU benchmarks. Experiments show that our design can significantly mitigate the negative impact due to small cache sizes and improve the overall performance. We analyze the performance across different platforms and applications. We also propose some optimization guidelines on how to efficiently use the GPU caches. This work has been presented at the International Conference for High Performance Computing, Networking, Storage and Analysis 2015 (SC-15) [87] and was nominated for best paper award and best student paper award.

5.1 Introduction

Graphics Processing Unit (GPU), the coprocessor originally designed predominantly for graphic rendering, nowadays has been proven unexpectedly successful in the domain of general-purpose applications (GPGPU) [146, 147, 44]. A crucial issue that confines the peak performance delivery, however, is the vast and sometimes irregular memory access from massively concurrent threads. This enforces considerable pressure on the bandwidth and efficiency of the memory system [43]. To reduce memory traffic and latency, modern GPUs have widely adopted hardware-managed cache hierarchies [148, 149]. However, traditional cache management strategies are mostly designed for CPUs and sequential programs; replicating them directly on GPUs may not deliver expected performance, as GPUs' relatively smaller caches can be easily congested by thousands of threads,

Table 5.1: Threads vs. Caches.

Processor	**L1 Cache**	**Threads/Core**	**Cache/Thread**
AMD Warsaw	16 KB	1	16 KB
Intel Haswell	32 KB	2	16 KB
Intel Xeon-Phi	32 KB	4	8 KB
Oracle M5	16 KB	8	2 KB
NVIDIA Fermi	48 KB	1536	32 B
NVIDIA Kepler	48 KB	2048	24 B
NVIDIA Maxwell	24 KB	2048	16 B
AMD Radeon-7	16 KB	2560	6.4 B

causing serious contention and thrashing. Table 5.1 lists the L1 cache[1] capacity, thread volume and per-thread L1 cache share for the state-of-the-art multithreaded processors. As can be seen, the per-thread cache share for GPUs is much smaller than for CPUs, which indicates that the useful data fetched by one thread is very likely to be evicted by other threads before actual (re-)usage. Such thrashing condition destroys locality and impairs performance. Moreover, the excessive incoming memory requests, particularly in an accessing burst period (e.g., the starting and ending phases of a kernel) if concerning the SIMT execution model [45] (see Section 5.2.1), can lead to significant delay when threads are queuing for the limited resources in caches, e.g., miss buffers, MSHR entries, a certain cache set, etc. [111, 135].

A naive response is to extend the cache capacity. However, it sacrifices the valuable die area that may otherwise be dedicated for more computation facilities. Therefore, instead of prototyping "*big-cached*" GPUs, designers are more prone to throttle the thread volume in order to reach a good balance between multithreading degree and cache efficiency [150, 125].

Traditional thread throttling mechanisms either advise users to refine their code using an ideal multithreading degree predicted from parsing the source code [29, 151], or suggest hardware modifications in the thread scheduler to limit active thread count, so as to match access footprints with the cache capacity [125, 99, 100]. However, the thread number from the user part (i.e. defined in the kernel configuration) is often determined by the underlying algorithm; altering it is not straightforward and may lead to the reimplementation of the algorithm, which demands tremendous user efforts. On the other hand, restricting threads according to cache capacity in the scheduler may diminish the utilization of the computation units and off-chip memory bandwidth [152]. Besides, the smart scheduler often requires either a brilliant compile-time analyzer or a powerful runtime detector. Further, the orchestrated hardware modifications can only be implemented in future products; it cannot benefit existing platforms anyway. Both of the above approaches are costly, from either application or hardware perspectives.

Thus the challenge is, can we design a throttling mechanism that is transparent to the user and the hardware, but is still adaptive and efficient? In this chapter, we give a solution: *during compilation, we can add a threshold so that only a limited number of threads can access the cache.* This chapter

[1] L1 cache refers to L1 data cache only.

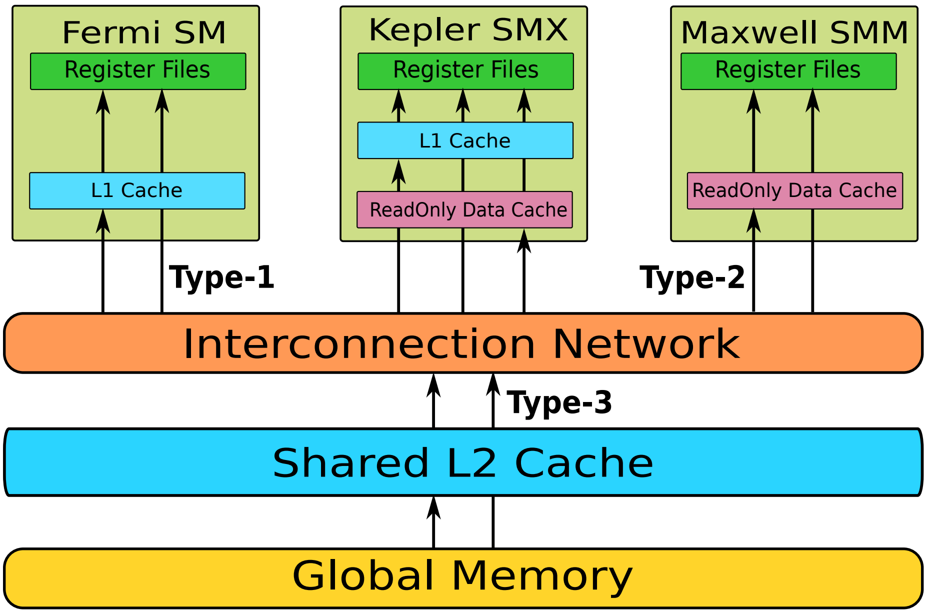

Figure 5.1: Global Memory Read Datapaths

makes the following contributions:

- We propose a novel and simple compile-time framework to do adaptive and transparent cache bypassing for global memory read, for all three types of GPU caches: *L1*, *L2* and *read-only* caches (Section 5.4.2).
- We propose a static and a dynamic approach to acquire the ideal bypass threshold (Section 5.4.4).
- We evaluate the bypassing framework on seven GPU platforms that covers all GPU generations with general caches inside: *Fermi*, *Kepler* and *Maxwell* with compute capability 2.0 to 5.2 (Section 5.5).
- We propose two software methods (Section 5.6.1) and investigate a hardware implementation (Section 5.6.2) to reduce the overhead of cache bypassing.
- Finally, we propose several optimization guidelines on the utilization of GPU caches (Section 5.5.3).

5.2 GPU Memory Access Datapaths

Since the majority of memory accesses are from/to global memory, the machine performance is much more sensitive to memory load than store (because load is often in the critical path as computation has dependence on the loaded data which is not the case for store). Therefore, we focus on *global memory read operations* only in this chapter. Regarding such operations, from Fermi to Kepler to Maxwell, there are three different datapaths with cache involved, as shown in Figure 5.1:

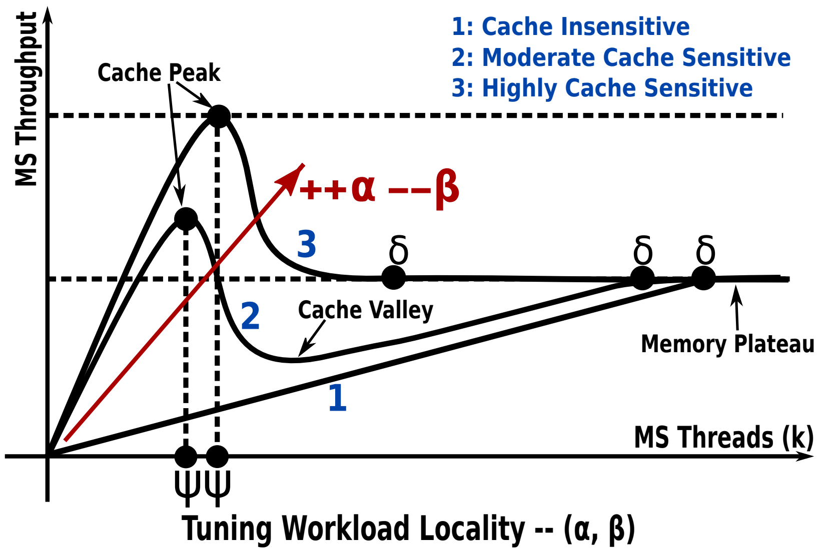

Figure 5.2: Plots for three types of GPU applications using the valley model.

- **L1 datapath** (Type-1 in Figure 5.1): from interconnection network to register files via L1 cache in both Fermi and Kepler[2] GPUs.
- **Read-only datapath** (Type-2): from interconnection network to register files via read-only cache in Kepler[3] and Maxwell GPUs.
- **L2 datapath** (Type-3): from global memory (GDDR) to interconnection network via L2 cache in Fermi, Kepler and Maxwell GPUs.

Accordingly, there are three possible approaches for cache bypassing during global memory read: *L1 cache bypassing*, *read-only cache bypassing* and *L2 cache bypassing*.

5.3 X-Model Analysis

In this section, we use the X-Model proposed in Chapter 3 to intuitively describe why cache bypassing can be effective for improving GPU performance. Based on the internal cache locality degree, we can characterize all GPU applications into three categories: *cache insensitive (CI)*, *moderate cache sensitive (MCS)* and *highly cache sensitive (HCS)* [99, 134]. Their corresponding curves using X-Model are already illustrated in Figure 3.8-(A). We duplicate it here in Figure 5.2 for easy reference and further discussion. As shown, the three categories are:

- **Cache insensitive** (CI) applications exhibit little data locality for global memory access. As thread volume expands, a higher utilization of the memory bandwidth is expected because the memory latency is increasingly hidden by context-switching among the extra threads.

[2]Only a fraction of Kepler GPUs support the L1 cache mode such as Tesla K40, K80, etc. [123].

[3]Only Kepler GPUs with compute capability larger or equal to 3.5 have the read-only cache.

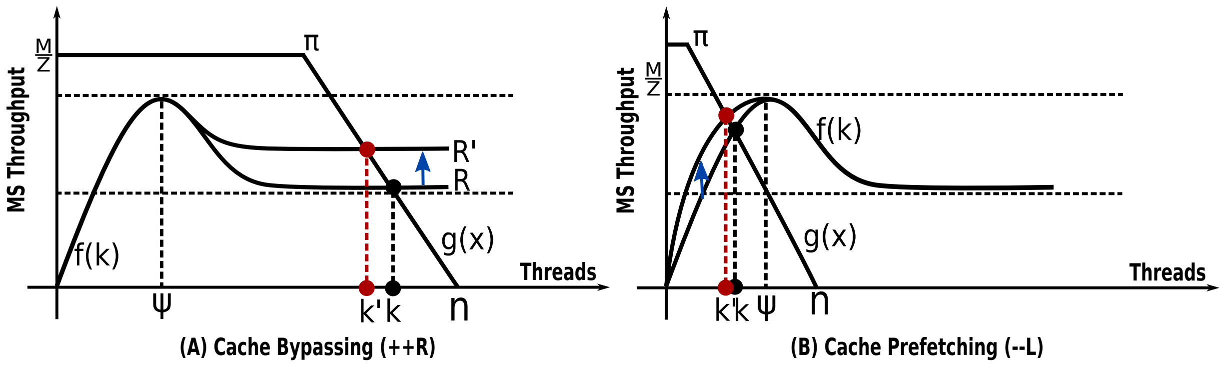

Figure 5.3: Improving cache performance via cache bypassing and cache prefetching using X-graph.

The memory hierarchy throughput curve increases monotonically with thread count until it approaches the bandwidth bound (*memory plateau* in Figure 5.3).

- **Moderate cache sensitive** (MCS) applications contain moderate data locality. As thread volume increases, more cache storage is leveraged. Meanwhile, the cache hit rate also goes up. However, when the aggregated working set exceeds cache capacity, thrashing occurs, which leads to a throughput degradation. The performance rising and dropping forms a peak (denoted as *cache peak*). Since the per-thread cache share for GPUs is much smaller than CPUs (see Table 5.1), the GPU cache peak is more to the left in the figure, implying that it is more easily congested. With further increased threads, the cache effect becomes obscure while the memory throughput increase becomes the major impact factor. Their joint-effects form the *cache valley*, as already discussed in Chapter 3. Beyond the valley, the cache effect vanishes while the memory throughput approaches the bandwidth bound, the throughput curve then remains constant at the memory plateau. The thread volume showing the best cache performance is the ideal thread volume, labeled as ψ.

- **Highly cache sensitive** (HCS) applications carry ample data locality, due to performance boosting of the cache, the memory system throughput increases much faster than MCS applications. Meanwhile, the cache peak of HCS applications is taller. In addition, due to the great data-reuse, the same cache size can sustain more parallel threads in the memory system, which explains why the position of ψ in HCS is more to the right. Note, as ψ is moving right, the cache valley may disappear. This is because the gap between the cache peak (ψ) and the memory plateau (δ) has narrowed.

For cache sensitive applications (MCS+HCS), there are two strategies that are widely used to improve performance:

- **Cache Bypassing:** As shown in Figure 5.3-(A), if there are too many memory requests that congest the cache (so $f(k)$ and $g(x)$ intersects beyond the cache peak), some of them can be bypassed from accessing the cache. The bypassing mitigates cache thrashing while still keeping sufficient threads to exploit the MLP of the lower memory. Thus, we see the rise of the memory plateau. As computation intensity Z is not changed, with the climbing of the intersection, both the CS and MS throughput increase.

- **Cache Prefetching:** As shown in Figure 5.3-(B), if the thread volume in the MS system is insufficient to fully exploit the cache capacity (so $f(k)$ and $g(x)$ intersects before the cache peak), we can add extra prefetching requests to saturate the cache while reducing the latency for requests hitting the prefetched cache-line. The extra prefetching requests improve the utilization of the cache with unchanged number of threads in MS. Therefore, we see the rising of the front-face of the cache peak when prefetching is applied. As Z keeps constant, with the climbing of the intersection, both CS and MS throughput increase.

In this work, we focus on cache bypassing. One can refer to [153, 154] and other references for GPU cache prefetching. Note, in the following part of this chapter, we use π other than ψ to denote the ideal thread volume to fit the cache.

5.4 Cache Bypassing

The proposed adaptive bypassing designs are presented in this section: we first describe the cache operators provided by the hardware. We then propose the horizontal bypassing design and compare it with the conventional vertical design. After that, we provide a case study. Finally, we show how to acquire the ideal bypass degree via a static and a dynamic approach.

5.4.1 Cache Operators

NVIDIA *Parallel-Thread-Execution (PTX)* ISA [155] introduces per-access cache operators for global memory read:

```
ld.global{.cop}{.nc}    %reg, [addr];
```

"*ld.global*" stands for global memory read. "*reg*" is the target register. "[*addr*]" is the source memory address. "*.cop*" is the cache operator which has different configurations:

- *.ca*: cache at both L1 (if available) and L2 with default LRU replacement policy.
- *.cg*: bypass L1 and cache at L2 with default LRU replacement policy.
- *.cs*: *streaming* cache at both L1 (if available) and L2. It assumes that the fetched data will be accessed only once so that **evict-first** replacement policy is adopted. This option is chosen to prevent the streaming data from polluting the useful cache lines.
- *.va*: cache as volatile. For global memory read, it is the same as *.cs*.

In addition, the "*.nc*" field has two options:

- Without *.nc*: normal memory load.

```
// ============ Bypass Header ============
mov.u32      %r0, %tid.x; //Thread index
shr.u32      %r0, %r0, 5; //Warp index
setp.lt.s32  %p0, %r0, $pi$; //Set Threshold
// ============== L1 Cache ===============
@%p0  ld.global.ca.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cg.s32  %r9, [%rd6]; //Bypass
// =========== Read-only Cache ===========
@%p0  ld.global.nc.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cg.s32  %r9, [%rd6]; //Bypass
// ============== L2 Cache ===============
@%p0  ld.global.cg.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cs.s32  %r9, [%rd6]; //Bypass
```

Listing 5.1: Adaptive cache bypassing

- With *.nc*: load from L2 to register via read-only cache.

Therefore, for a specific global memory read access, we can set up the following combinations for cache bypassing corresponding to Type-1,2,3 global memory read datapaths shown in Figure 5.1:

- For **L1** cached access, it is *ld.global.ca*; for L1 bypassed access, it is *ld.global.cg*.
- For **read-only** cached access, it is *ld.global.nc*; for read-only bypassed access, it is *ld.global.cg*.
- For **L2** cached access, it is *ld.global.cg*. For L2 bypassed access, since there is no particular L2 bypassing operator offered while the *.cs* option that adopts eviction-first policy reduces the impact on the original cache content, due to recent data accesses, to the smallest extent, we use *ld.global.cs* as an "imperfect substitution" for L2 bypassing if there is no L1 cache. Even with L1 available, streaming-style load at both L1 and L2 is the type of load that is the closest to L2 bypassing.

5.4.2 Horizontal Cache Bypassing

With the three configurations as a preamble, we can set up the horizontal cache bypassing framework. We define a **bypassing threshold**: *for warps with index less than the threshold, they perform cached read; for warps with index larger or equal to the threshold, they do cache bypassing*.

The design is shown in Listing 5.1. We first use the thread index to locate the warp it belongs to (by dividing index with the warp size 32). Here, it should be noted that the PTX predefined identifier *%warpid* [155] cannot be leveraged because it returns the physical warp-slot index, not the one defined in the user-program context. Since the physical warp-slot is dynamically bound to the warps, using it may destroy intra-warp locality, which is the major resource for potential data-reuse in HCS applications [99]. Note, it is also possible to embed PTX into the CUDA program using intrinsic functions. However, working at PTX level is easier for parsing and is transparent to the users.

Depending on whether the warp index is less than the bypassing threshold π (pi), a predicate register *p0* is configured. Then all the global loads in the PTX program are converted to conditional accesses: *if p0 is true, cache; otherwise, bypass*. Listing 5.1 shows the conditional statements for the three types

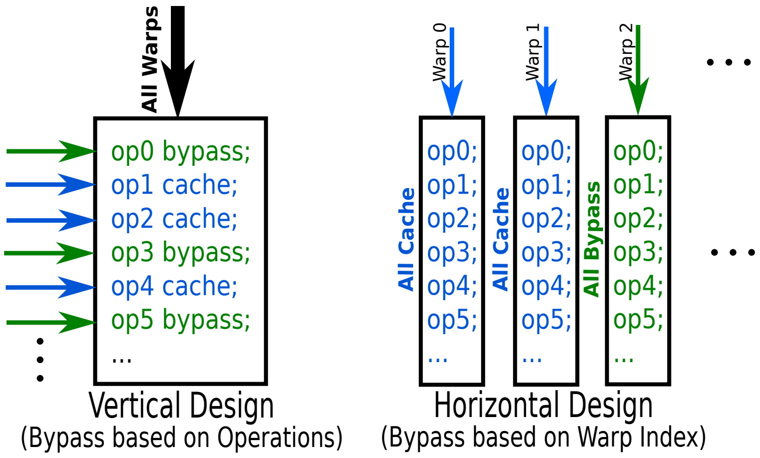

Figure 5.4: Bypass design approaches: vertical vs. horizontal.

of GPU caches. We use warp rather than thread here as the granularity for conditional bypassing to avoid the expensive warp divergence overhead (see Section 5.3.1) and conserve coalesced accessing patterns (see Section 5.3.2).

Such a design is quite clear yet efficient: overall, only a 1-bit predicate register is required per thread as the **space cost**. The general register used for calculating warp index is only required inside the bypassing header block (see Listing 5.1). Since the header block is always placed at the beginning of a kernel, this register can be recycled immediately after usage. Regarding the **time cost**, except one shift operation and one predicate register setting, the major overhead is the instruction issuing delay for the one additional load (two load instructions are issued, but only one is executed). Although such overhead becomes noticeable (see Section 5.4.3) when there are large amounts of memory accesses, it could be reduced by merging them together since the decision for bypassing or not is constant throughout the warps' lifetime. We discuss how to reduce this overhead in Section 5.6.

There are three reasons for cache bypassing to be beneficial to performance: first, it mitigates cache congestion so that the thread volume can match the cache capacity. In this way, the warps to be cached do not have to worry about their useful data being evicted before usage. Since the cache space per warp is sufficient to cover the accessing footprints, inner-thread and inner-warp locality are preserved and captured. Second, while the remaining warps bypass the cache, they do not need to wait for the shared resource in the cache (e.g., MSHR entry, an associative set entry, etc.) to be available before entering the memory pipeline. Last but not the least, the parallelism for the computation system is not sacrificed as we maintain the number of dispatched threads in the machine.

We would like to compare our proposed bypass design (marked as *horizontal approach*) with the existing cache operator based schemes (such as [150, 156], denoted as *vertical approach*):

- The **vertical approach** follows the conventional CPU's design paradigm that operates within a single thread scope. As shown in Figure 5.4, all threads/warps execute the same instruction stream while inside the stream, for each global memory read, one has to decide whether to bypass or not. The design spectrum is along the vertical *instruction direction*. Since every read instruction fetches different data, if there are m read, the design complexity is $O(2^m)$, for which m can be very large. Such a broad design space is quite difficult to traverse. Moreover, as all threads follow the same execution path, they tend to access the cache at the same time, which is more likely to congest the cache. However, this vertical design does not incur any extra time/space overhead at runtime. If assisted by a smart scheduler, it can distinguish and abolish data with little locality thus avoiding detrimental cache pollution.

- The **horizontal approach** on the other hand focuses on the most prominent characteristic of GPUs — multithreading. As shown in Figure 5.4, for each different warp, one has to decide if it belongs to the bypass group or cached group. However, as soon as the decision is made, all the global memory read in that warp follow. The design spectrum is along the horizontal *warp direction*. As warps in a CTA are identical, the design complexity for n warps is $O(n)$, where n is less than or equal to 32. (This is true for all existing NVIDIA GPUs [53]). In fact, for all applications we tested in Table 5.3 and all benchmarks in Rodinia [37], $n \leq 16$. Still, the memory requests may come in a burst, but bypassing enforces the number of warps that access the cache, which significantly mitigates the pressure on the cache. The drawbacks, however, are the small time and space cost.

There is no clear conclusion on which approach is better. They are **orthogonal** to each other: one focuses on code property and one focuses on concurrency. The horizontal design sees the kernel code as a blackbox, therefore, cannot distinguish those loads with little reuse. Caching such loads can be detrimental even with horizontal bypassing adopted. So a more attractive approach is a hybrid design: first bypass loads with little locality via vertical approach; then apply horizontal bypassing on the remaining loads if cache thrashing remains. We set this as a future work.

5.4.3 BFS Case Study

To make a clear explanation about how cache bypassing can benefit performance, a detailed case study is provided. We focus on Breadth-First-Search (BFS) in Table 5.3. The testing platform is Fermi (*Platform-1* in Table 5.2). To avoid possible interference due to insufficient data size, we use the largest dataset (*graph-1MW_6.txt*) in the benchmark. Except inserting the bypassing header and converting global memory read in the PTX routine (as in Listing 5.1), we do not make any other modifications to the kernel code or kernel configurations (i.e., threadgrid, threadblock, shared memory allocation, etc.). We vary the threshold value from 0 to the number of warps defined in the application (16 in this example). Also, the results for bypass-all (denoted as `bpa`) and cache-all (denoted as `cha`) are shown for reference. All result figures are the average value for 5 execution runs.

Figure 5.5, 5.6 and 5.7 illustrate the kernel execution time with respect to the increased bypassing threshold on L1, L2 and L1-L2 together with 16KB L1. Figure 5.8, 5.9 and 5.10 show the time with 48KB L1. There are two L2 bypassing results with different L1 configurations. The reason is that the L2 bypassing does not actually bypass L2 but accesses the L1 and L2 in a streaming fashion on

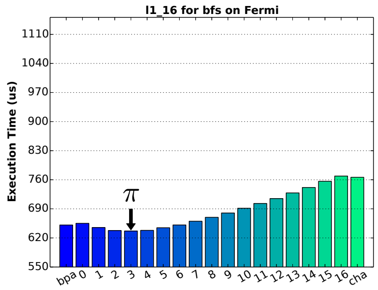

Figure 5.5: BFS cache bypassing on 16KB L1.

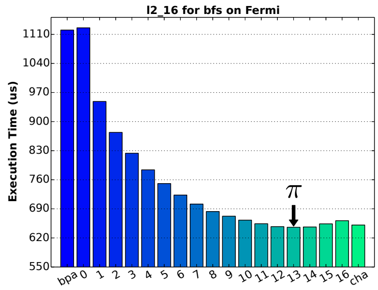

Figure 5.6: BFS cache bypassing on L2 with 16KB L1.

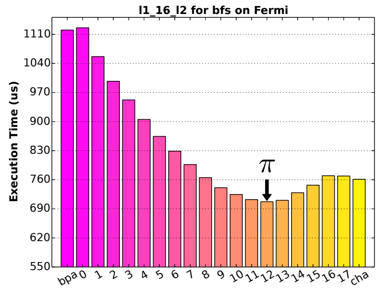

Figure 5.7: BFS cache bypassing on 16KB L1 and L2 simultaneously.

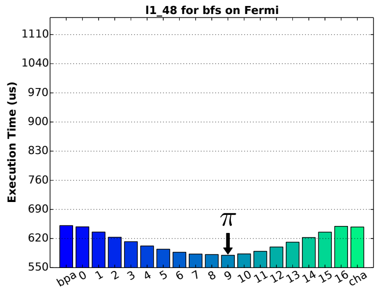

Figure 5.8: BFS cache bypassing on 48KB L1.

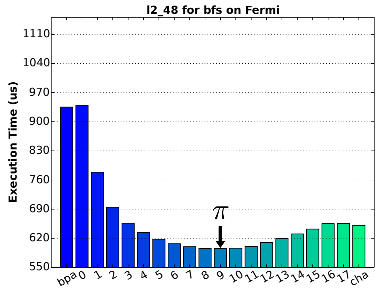

Figure 5.9: BFS cache bypassing on L2 with 48KB L1.

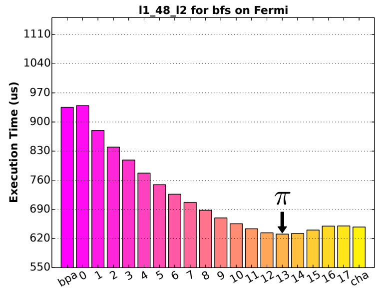

Figure 5.10: BFS cache bypassing on 48KB L1 and L2 simultaneously.

Fermi (see Section 4.1). That's why the L1 configuration affects L2 bypassing performance. Besides, Figure 5.7 and 5.10 show the L1-L2 combining bypass effects. Comparing the six figures, we have the following observations:

1. The shapes of the curves confirm the valley model described in Section 5.3.1. As can be seen, π marks the position of the **cache peak**. In Figure 5.5, $\pi = 3$ indicates that the footprint for one warp is slightly more than 5KB (16KB/3) which is confirmed by $\pi = 9$ (48KB/9) in Figure 5.8. Meanwhile, the **cache valley** is quite obvious in Figure 5.5, as the performance degrades significantly beyond the cache peak, to a degree that is even much worse than no caching at all. A larger L1 alleviates the valley effect (from Figure 5.5 to Figure 5.8), but still, no clear gain is attained (`bpa` and `cha` are similar in Figure 5.8). As a comparison, for both cases bypassing filters out the excessive requests which leads to a more efficient utilization of the L1 cache.

2. Regarding L2 (Figure 5.6 and 5.9), `cha` performing better than `bpa` implies that the valley effect mitigates in L2. Also, the fact that the bypassing benefit is larger for L2 than L1 implies that the overall machine performance is more sensitive to L2 cache than L1. However, it should be noted that the best bypassing performance is always attained on L1 cache (compared with Figure 5.5 and 5.8). This means **bypassing on L2 only is not sufficient**.

3. We also evaluate bypassing on both L1 and L2 at the same time (Figure 5.7 and 5.10). This approach is equivalent as *if cache, then cache at both L1 and L2; otherwise, bypass them all*. Note, unless using additional thresholds for L1 and L2 respectively, this is the only combining

approach. As can be seen, the performance is worse than bypassing on L1 and L2 alone, which means the **bypassing benefit on L1 and L2 are not cumulative**.

4. About the execution overhead for bypassing. Recall that the decision boundary for caching or bypassing is "*less than*", the threshold value equals to zero thus has the same context meaning as `bpa`, but additionally contains the space and time overhead of the bypassing framework. Therefore, the small discrepancies between `bpa` and $\pi = 0$, `cha` and $\pi = 16$ in the figures are such overhead. However, it should be noted that in Figure 5.8, the overhead appears to be "negative" ($\pi = 0$ is less than `bpa`), this is because in the added bypassing operations (and bypassing head) may alter the original warp scheduling decision at runtime, which leads to such "rare" effect.

5.4.4 Acquire Ideal Bypassing Threshold

There is one question left: *how to acquire the ideal threshold π ?* In this chapter, we propose a static and a dynamic approach.

Static Approach: The static approach is straightforward: *just exhaustively assess all the selective values for the threshold*. Here, it highlights the advantages of horizontal bypassing over the vertical one: we only need to test 32 times at most. In fact, to reach acceptable SM occupancy, most applications have less than 16 warps in their thread block configurations. As discussed, this is true for all the applications in Rodinia and the ones we tested in Table 5.3. As a comparison, with only 10 loads in the kernel, a vertical scheme would have 1024 different configurations (see Section 4.2).

The advantage of the static approach is that it always returns the optimal threshold for the current dataset. Meanwhile, as GPUs normally run fast, executing a kernel 16 times is a not significant overhead. This makes the static approach a good option for program auto-tuning. The drawback, however, is that the attained threshold may correlate with the testing dataset. To overcome this "over-fitting" problem, people could use a more representative dataset or profile with multiple datasets to confirm the trend (see Section 5.2 and Section 5.9).

Dynamic Approach: The dynamic approach is a runtime voting method. As shown in Figure 5.11, we assume that there are 1024 CTAs in total for the kernel and each CTA has six warps based on the application logic. The kernel is then amended to generate the sampling procedure in three steps: first, seven CTAs (instead of 1024) are initiated with consecutive bypass values, from $x = 0$ to $x = 6$. Then, for each CTA, a thread (e.g., *tid=0*) is enforced to measure the execution time of the entire CTA with the associated threshold level. The timing result is submitted **atomically** to a global-scope bypassing threshold π. Finally, if the eventual value of π equals to zero or six, the runtime manager discards the conditional statement and uses `bpa` or `cha` instead. Again, with $max(\pi) \leq 32$, we can assess all selective options with a few sampling CTAs. The sampling procedure can be integrated into the runtime library to avoid user involvement.

This approach is practical and easy to implement. However, it has its drawbacks: first, it works only

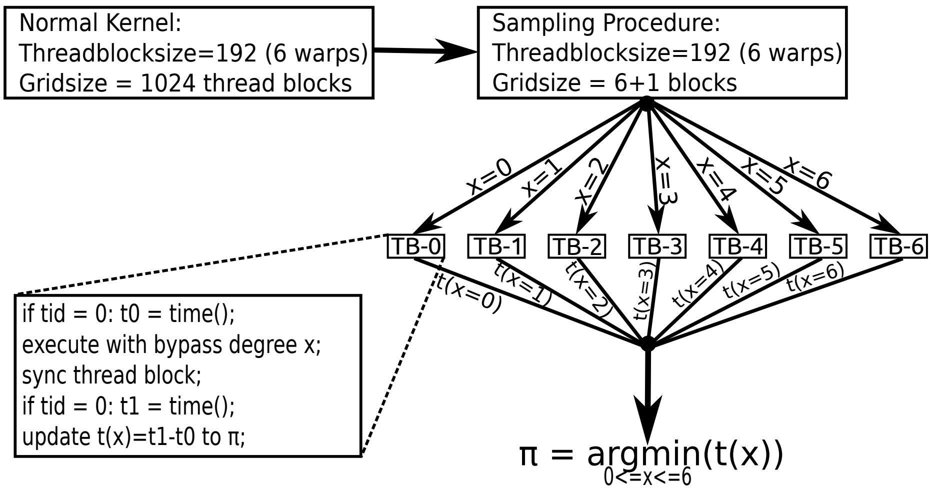

Figure 5.11: Sampling and voting for optimal bypassing threshold π.

for L1 cache bypassing. Second, it cannot handle inter-CTA unbalancing (i.e., irregular applications may have different workload for different CTAs). Third and most importantly, during the sampling phase only one CTA is allocated per SM, so this CTA essentially occupies the entire L1 cache. But in a real execution, this is not the case; generally multiple CTAs are sharing the L1 cache simultaneously. Therefore, the sampled threshold may not be accurate. Regarding this problem, as we cannot alter the CTA scheduling policy via software approaches, a possible solution would be: allocate sufficient CTAs to saturate all SMs. Instead of profiling different π with different CTAs (as in Figure 5.11), we now profile in different SMs: before setting the timer, the pilot thread first acquires the sm_id of the resident SM from the special register `%smid`. Then, with different sm_id, a different π is assessed. In this way, the sampling phase simulates the actual execution more accurately.

5.5 Evaluation

In this section, we validate the proposed bypassing framework. In order to evaluate the general effectiveness of the framework, we use seven GPU platforms that covers **ALL** existing NVIDIA GPU generations with general cache integrated, say from compute capability (**CC**) 2.0 to 5.2[4], as shown in Table 5.2. We take 16 cache sensitive (HCS+MCS) applications from the Rodinia [37], Parboil [38], Mars [33] and Polybench [131] benchmarks. Since all the applications in the Mars benchmark share the common Map-Reduce kernel library, we only use one application (*SSC*). Besides, the Mars applications cannot compile properly on other platforms, so we only show the results of *SSC* for Fermi with CC-2.0. We use Normalized IPC as the performance metric since cache hit rate does not necessarily lead to better overall performance for GPUs [99, 157]. The normalized IPC here is simply the reciprocal of the execution time; we do not count the added bypass instructions when calculating IPC. Again, except inserting the bypassing header and converting global memory read in

[4]CC-3.2 and 5.3 are for embedded systems only.

Table 5.2: Experiment Platforms

Plat.	GPU	Arch-Code	CC.	Cores	GPU Freq	Mem Band	Dri./Rtm.	CPU	gcc
1	GTX570	Fermi-110	**2.0**	15 SMx32	1464 MHz	152 GB/s	6.5/4.0	Intel Q8300	4.4.7
2	Tesla K80	Kepler-210	**3.7**	13 SMXx192	824 MHz	240 GB/s	7.0/7.0	Intel E5-2690	4.4.7
3	GTX750Ti	Maxwell-107	**5.0**	5 SMMx128	1137 MHz	86.4 GB/s	6.5/6.5	Intel i7-4770	4.4.7
4	GTX460	Fermi-104	**2.1**	7 SMx32	1400 MHz	88 GB/s	6.5/6.5	Intel i7-920	4.6.3
5	GTX690	Kepler-104	**3.0**	8 SMx192	1020 MHz	192 GB/s	7.0/6.5	Intel i7-5930K	4.8.4
6	Tesla K40	Kepler-110	**3.5**	15 SMXx192	876 MHz	288 GB/s	6.0/6.0	Intel E5-2620	4.4.7
7	GTX980	Maxwell-204	**5.2**	16 SMMx128	1216 MHz	224 GB/s	6.5/6.5	Intel i3-4160	4.8.2

Table 5.3: Benchmark Characteristics

Application	Description	abbr.	Warps	Input dataset	Source
bfs	Breadth First Search	BFS	16	graph1MW_6.txt	Rodinia[37]
backprop	Back Propagation	BKP	8	65536	Rodinia[37]
b+tree	B+ Tree Operation	BTE	8	mil.txt-command.txt	Rodinia[37]
kmeans	K-means Clustering	KMN	8	kdd_cup	Rodinia[37]
stencil	3-D Stencil	STE	4	128x128x32.bin-128-128-32-100	Parboil[38]
particlefilter	Particle Filter	PTF	16	128x128x10, np:1000	Rodinia[37]
spmv	Sparse Matrix-Vector Multiplication	SPV	6	Dubcova3.mtx - vector.bin	Parboil[38]
streamcluster	Stream Cluster	STC	16	10-20-256-65536-65536-1000	Rodinia[37]
srad	Speckle Reducing Anisotropic Diffusion	SRD	16	100-0.5-502-458	Rodinia[37]
bicg	BiCGStab Linear Solver	BIC	8	default	Polybench[131]
atax	Matrix Transpose Vector Multiply	ATX	8	default	Polybench[131]
gesummv	Scalar Vector Matrix Multiply	GES	8	default	Polybench[131]
mvt	Matrix Vector Product Transpose	MVT	8	default	Polybench[131]
syrk	Symmetric Rank-K Operations	SYR	8	default	Polybench[131]
syr2k	Symmetric Rank-2K Operations	SYK	8	default	Polybench[131]
similarityscore	Similarity Measure between Documents	SSC	16	256-128	Mars[33]

the PTX routine (as in Listing 5.1), we do not make other modifications to the kernel code or kernel configurations. Note, for read-only caches, we only apply bypassing to loads that are accessing the "read-only" variables or arrays as the read-only caches are non-coherent. In this chapter, we show the results for Platform 1 to 3. For the results of other platforms, please refer to Section 5.9.

Platform-1 – Fermi: The results for 16KB L1, 48KB L1 and L2 on Fermi with CC-2.0 are shown in Figure 5.12, 5.13 and 5.14. For comparison purposes, we normalize the performance to `bpa`[5]. `G-M` is the geometric-mean-value. Similar to the case study in Section 5.4.3, the differences between *bypass* and *opt* imply the bypassing overhead.

As can be seen in Figure 5.12, the 16KB L1 cache is far from sufficient to cover the data footprints, which leads to the inferior performance of `cha` compared with `bpa` (11% worse). Therefore, using the L1 cache naively is detrimental. However, this situation is effectively improved by the proposed bypassing scheme, which leads to 24% speedup over `bpa` and 39% over `cha`. The serious thrashing problem of 16KB L1 has been significantly mitigated by extending the cache size to 48KB. As shown in Figure 5.13, `cha` is 17% better than `bpa` now. Nonetheless, the effect of cache bypassing is more prominent: it demonstrates 45% speedup over `bpa` and 24% over `cha`. Regarding L2 in Figure 5.14,

[5] `bpa` is the default behavior for L1 and read-only caches of Kepler and Maxwell GPUs. However, on Fermi L1 and all L2 caches, the default is `cha`.

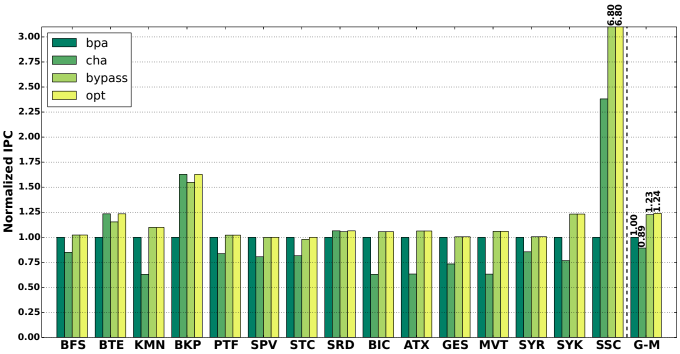

Figure 5.12: 16KB L1 cache bypassing on Fermi GPU.

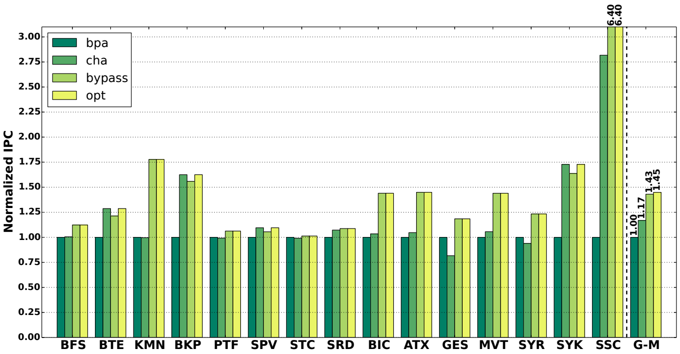

Figure 5.13: 48KB L1 cache bypassing on Fermi GPU.

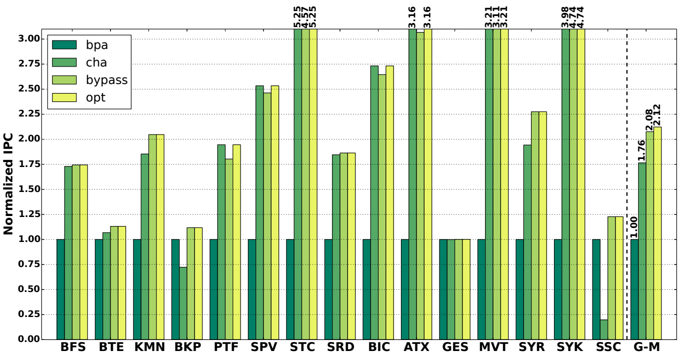

Figure 5.14: L2 cache bypassing on Fermi GPU.

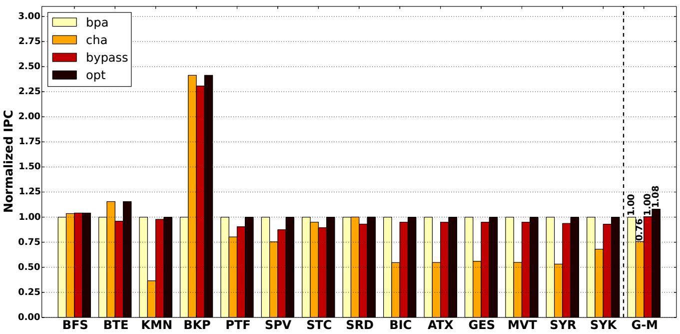

Figure 5.15: 16KB L1 cache bypassing on Kepler GPU.

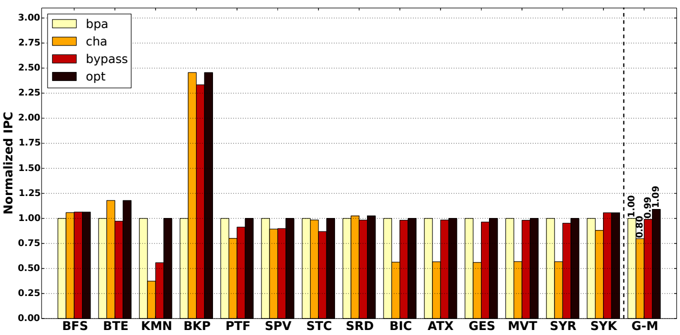

Figure 5.16: 32KB L1 cache bypassing on Kepler GPU.

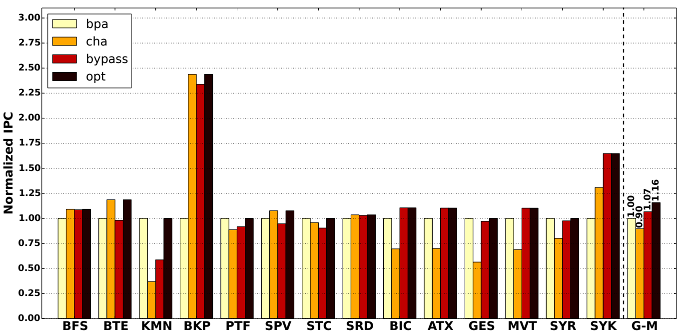

Figure 5.17: 48KB L1 cache bypassing on Kepler GPU.

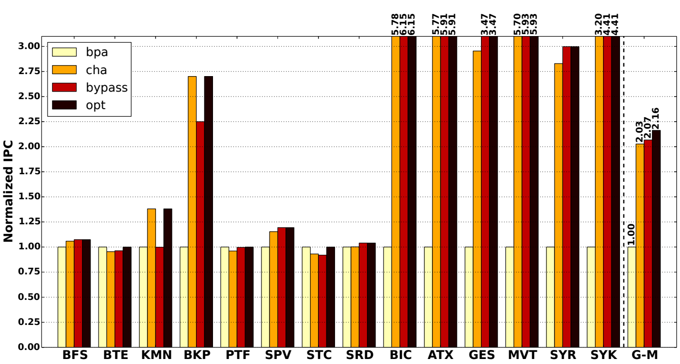

Figure 5.18: Read-only cache bypassing on Kepler GPU.

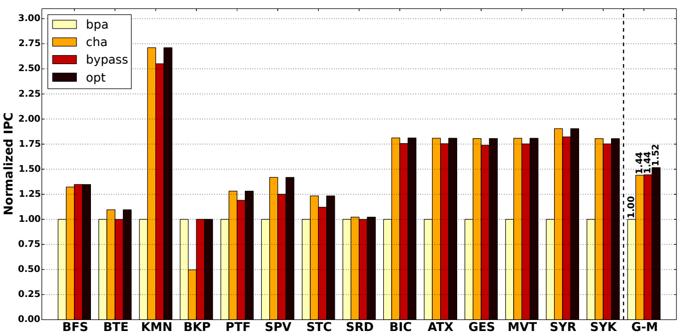

Figure 5.19: L2 cache bypassing on Kepler GPU.

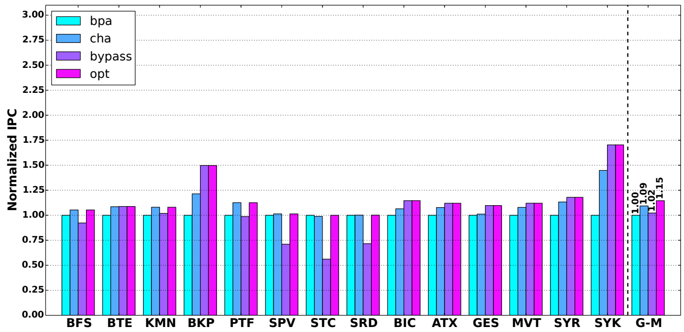

Figure 5.20: Read-only cache bypassing on Maxwell GPU.

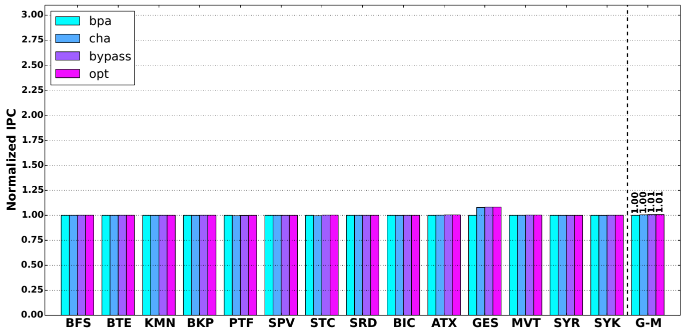

Figure 5.21: L2 cache bypassing on Maxwell GPU.

the fact that cha is much better than bpa indicates that caching in a streaming fashion (in both L1 and L2) is much worse than caching normally in L2 for most cases (except BKP and SSC). Also, our scheme achieves 1.12x speedup over bpa and 20% over cha in L2 cache. Besides, it should be noted that for all the three tests on Fermi with CC-2.0, the overhead introduced by the bypassing framework is quite small (1%, 2% and 4%).

Platform-2 – Kepler: Next we validate cache bypassing on a Kepler platform with CC-3.7 – the latest Tesla-K80 GPU. The results for 16KB, 32KB, 48KB L1, read-only and L2 caches are shown in Figure 5.15, 5.16, 5.17, 5.18 and 5.19, respectively.

Unlike Fermi, the L1 cache in Kepler is harmful on average in all configurations albeit the degree is declining (24%, 20% and 10% worse for 16KB, 32KB and 48KB). Meanwhile, the effectiveness of cache bypassing also remains evident, with a speedup of 8%, 9%, 16% over bpa and 42%, 36%, 29% over cha. The scenario for read-only cache is, however, completely different. As shown in Figure 5.18, the benefit of exploiting the read-only cache is 2.03x speedup of cha over bpa. In addition, the bypassing framework leads to 2.16x speedup over the default bpa approach. The condition of L2 is similar to Fermi.

Platform-3 – Maxwell: Lastly, we run the experiments on the Maxwell architecture with CC-5.0. Since Maxwell completely discards L1 cache and uses the entire on-chip storage for shared memory, we can only establish read-only cache and L2 cache bypassing. The results are shown in Figure 5.20 and 5.21.

Different from Kepler, the read-only cache for Maxwell is not that beneficial, which exhibits a 9% speedup. Moreover, cache bypassing brings only 15% better performance than bpa for read-only cache bypassing and almost none for L2 cache. In addition, it should noted that the overhead for cache bypassing is more significant on Maxwell: 13% for read-only cache. We explain the reasons for L2 bypassing results in Section 5.4 and the overhead problem in Section 5.9.

5.5.1 Performance Analysis Across Platforms

Figure 5.22 summarizes the geo-mean performance gains for all the applications with all possible caches & cache configurations for the seven GPU platforms in Table 5.2. As can be seen, for Fermi

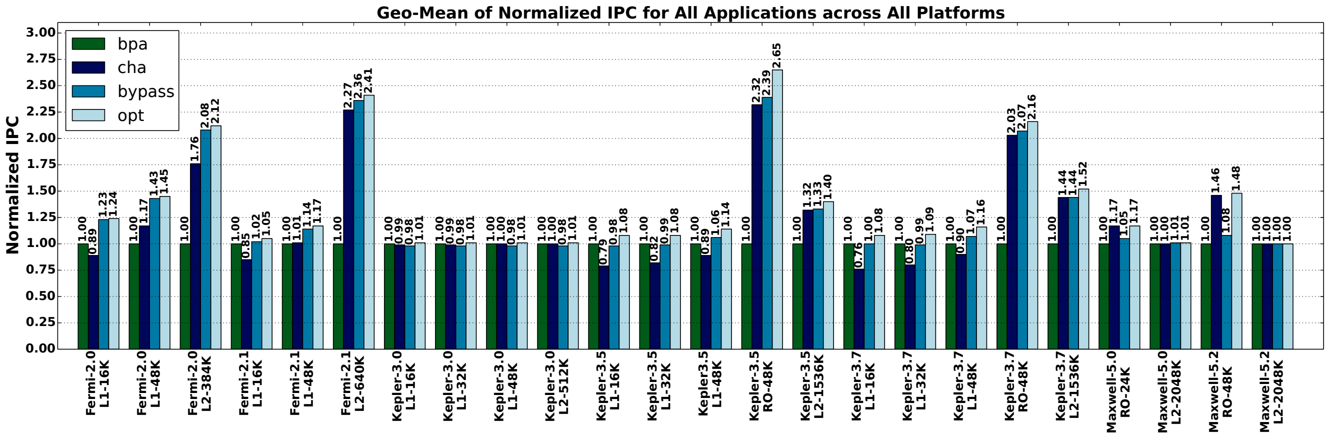

Figure 5.22: Performance for all applications across all platforms. For the x-ticks, the left is the major architecture and compute capability of the platform while the right is the cache type and size.

CC-2.0 and 2.1, cache bypassing is quite effective, especially on large L1 caches and L2 caches. Note that `cha` with 16KB L1 degrades performance by 11% and 15% respectively compare to `bpa`. This explains why from Kepler, L1 cache no longer remains the default datapath for global memory access.

For Kepler CC-3.0, the bars are identical (Kepler-3.0 L1-16K/32K/48K in Figure 5.22). This is because in Kepler CC-3.0, the L1 cache is only for local memory access [53]. Therefore, bypassing L1 or not does not impact global memory access. For CC-3.5 and 3.7, bypassing works perfectly for read-only caches and L2 caches. Again, L1 cache is detrimental while the bypassing framework eliminates such negative effects effectively.

Regarding Maxwell CC-5.0 and 5.2, bypassing improves performance for read-only cache. However, there is no performance gain on L2. This is because in Maxwell, the "`.cs`" suffix has been abandoned. Therefore, bypass or not generate exactly the same code. We validate this by checking the SASS code — `.cs` and `.ca` produce identical binary file.

5.5.2 Performance Analysis Across Applications

For applications, regarding their behaviors against threshold variation, we can characterize them into five categories: *bypass-favorite, cache-favorite, cache-congested, cache-insensitive* and *irregular*. For *bypass-favorite* applications, the performance continuously degrades with a higher bypass threshold. This may be due to the rapidly increased L2 traffic induced by the larger L1 cache-line size [157]. `bpa` is the best choice for these applications. Conversely, for *cache-favorite* applications, the performance keeps increasing with a higher threshold. These applications have good locality while the footprints are small enough to be effectively captured by the cache. This condition occurs mostly on L2 and `cha` is the optimal choice. *Cache-congested* applications are those with good locality but experience congestion due to insufficient cache size, such as `bfs` in the case study. The shapes of the graphs of these applications are convex while the optimal threshold attains in the middle. These applications are the best candidates for cache bypassing. *Cache-insensitive* applications (e.g., `stencil`) have little locality while the overhead from the bypassing framework is quite obvious in the figures. Finally, *irregular* applications show an irregular shape that has no clear trend (e.g., `syrk`). This may be due to the irregularity of the algorithms or datasets. To view the typical figures for each category discussed, please refer Section 5.9. Note, for the first four **regular** categories, the trend is not very sensitive with the variation of the dataset. Therefore, if we can determine the trend by profiling on a typical dataset, the same option (i.e., `bpa`, `cha` or a certain threshold value) may be applied to other datasets.

5.5.3 Optimization Suggestions

In addition to the bypassing analysis, we propose several optimization suggestions for general cache utilization:

- In Fermi, if there is no big pressure on shared memory usage, always adopt the 48KB L1 configuration. Otherwise, bypass L1 via ptxas option "*dlcm=cg*" if no bypassing is applied.
- In Kepler, try to use the read-only cache instead of the L1 unless you know it will be beneficial to use L1.
- In Kepler and Maxwell, apply the read-only cache bypassing just on the data that are "read-only" in the kernels. Otherwise, you may suffer from performance degradation (e.g., about 6% for Maxwell in our experiments).
- In all architectures, using "__restrict__ const" on read only data reduces register usage (up to half in our observation) and improves code generation quality [123] (e.g., about 16% performance gain for Maxwell L2).

5.6 Discussion

In this section, we discuss the possibility to reduce bypassing overhead (i.e., predicate register checking per load) via software and hardware approaches. We also clarify *why the proposed cache bypassing design incurs more overhead on Kepler and especially Maxwell than on Fermi*.

5.6.1 Software Approach

The major reasons for the larger overhead in Kepler and Maxwell than in Fermi, is that after we insert the bypass branches into the *PTX* program, when converting *PTX* into binary, the *ptxas* assembler performs aggressive optimizations, which attempts to combine the many "small divergence" together. In our observation of the *SASS* code, instead of being divergent only at the load operations, the optimized code diverges in much larger code sections and uses completely different registers. This leads to higher register usage and poor instruction cache performance. However, such case is not observed in the code generation for Fermi. Therefore, a direct reaction for reducing overhead is to modify the *SASS* code directly rather than *PTX*. However, there is no official *SASS* assembler available till now and *ptxas* is not open-source. A homemade assembler such as "*maxas*" may help, but is out of the scope of this thesis.

Another simple software method is to replicate the whole kernel so that a warp branches from the beginning: *if bypass, a warp executes the copy of kernel with bypassing; otherwise, executes the copy without bypassing*. However, we did not apply this optimization in this chapter because: first, it doubles the static code size of the kernel. Second, it may lead to thrashing in the SMs' instruction caches. Please refer to the discussion about "code overlaying" in [94]. Finally, one has to carefully handle the possible interplay between warp branching and CTA-wise synchronizations. Nonetheless, we would like to evaluate this optimization as future work.

5.6.2 Hardware Approach

The hardware method is to realize the judging process of bypassing in the cache controller. We use a 5-bit register (32 warps at most), to conserve the bypassing threshold. The register is configured when the kernel is launched. Then, for each memory request, upon it arrives at the cache, its warp index is compared with the threshold register, if less, it is appended to the cache waiting queue, otherwise, it is forwarded to the request queue of the lower-level memory devices. For example, if bypassing L1, the request is forwarded to the *MRQ* [153] and is later injected into the interconnection network.

Migrating the bypassing functionality into the hardware eliminates the 1-bit predicate register cost per thread as well as the corresponding assessment of it upon each time's memory access, which improves performance and reduces power. In fact, we implemented this hardware design in GPGPU-Sim [43] using GTX480 (Fermi) architecture with 16KB L1. The simulation results show that the hardware implementation is slightly better than the software regarding both performance and power (2% performance improvement and 2% energy reduction). However, as GPGPU-Sim does not perfectly mimic the behaviors of the real hardware (e.g., based on our previous work [111], Fermi hardware uses an XOR-based hashing in the L1 cache, but such a module is not implemented in GPGPU-Sim), there is a big mismatch for some applications (e.g., SSC and BKP) between the simulation outcome and the real hardware measurement (i.e., Figure 5.12). Therefore, we did not include the figures here but put them in Section 5.9.

5.7 Related Work

Recently warp-throttling and cache bypassing for enhancing the performance of GPU caches became hot topics [99, 100, 150, 156, 135, 134, 158, 159].

Rogers et al. [99] proposed a cache-conscious wavefront scheduler (CCWS) to limit the number of active wavefronts to be allocated when lost locality was detected. CCWS was later refined as divergence-aware warp scheduling (DAWS) [100], which used a divergence-based cache footprint predictor to assess the L1 cache capacity that was able to capture intra-warp locality within loops. Xie et al. [150] developed a compiler framework to parse the application code and select a set of load operations that bypassing them at L1 could reduce the most L2 cache traffic, based on an ILP or a heuristic optimizer. These operations were then appended with the "*cg*" suffix for bypassing the L1 cache at runtime. The design was tested on a Kepler GTX-680 platform. To compare, their design was a "vertical" bypass design. The "bypassing set" selecting process, as proved in their paper, was an NP-hard problem. Besides, their design was only for the L1 cache of Fermi and a small number of Kepler GPUs. Further, L2 traffic reduction did not necessarily lead to the shortest execution time. Very recently, Li et al. [156] proposed another vertical design for GPU L1 cache bypassing. By integrating a locality filter in the L1 cache, memory requests with low reuse or long reuse distance can be excluded from polluting L1. Jia el al. [135] proposed a dynamic hardware approach that bypasses memory load requests when experiencing resource unavailability stalls, particularly cache associativity stalls. While their design might greatly reduce stall waiting, blindly bypassing memory requests whenever there were resource bound might be a bit aggressive, which could hamper performance. The design was runtime resource based which had little relevance to the features of the applications. Chen et al. [134] developed a hardware bypassing mechanism to protect hot cache lines from early eviction based on *lost locality score* detection. Meanwhile, as cache bypassing may lead to congestion at NoC or DRAM, a warp-throttling function for the warp scheduler was supplemented to limit the number of active warps if necessary. Such a design was also runtime hardware based. Mekkat et al. [158] concentrated on CPU-GPU heterogeneous platforms and observed that GPU applications with sufficient thread-level parallelism could tolerate long memory access latency. Therefore, memory requests from GPU threads could bypass LLC while leaving the space for cache sensitive CPU applications. Li et al. [159] implemented a priority-token based hardware design for L1 cache bypassing. In the design, each active warp is allocated with "an additional scheduler status bit". Several "oldest" running warps are granted with high priority while their status bits are set, meaning that only these warps can access the L1 cache. The value of the bit is then appended to each memory request so that the L1 cache is notified.

Most of these schemes, however, concentrated on the architectural design of the memory hierarchy and suggested complicated hardware refinement, which required significant efforts and were not able to **bring instant performance gain to the existing GPUs**. Besides, the validation of the schemes were performed on simulators. As a comparison, our design is purely software (except Section 5.6.2) and is straightforward to implement. It leverages the reconfigurability of the existing hardware, thus is beneficial to most existing GPUs. Our design can be embedded into the compiler toolchain or encapsulated as a runtime library. Xie et. al. [150] adopted similar cache suffix-based approach as ours. However, as discussed, their bypassing scheme was vertical-based. The search space is much larger. Besides, they focused on L1 only and validated using a single platform GTX-680 (In fact, we are confused about why a Kepler with CC-3.0 can exploit L1.). The very recent work by Li et. al. [159] is a horizontal design. However, it is hardware based such that significant area and runtime overhead are introduced: e.g., the additional status bit registers, the extended memory request length, the delay of token management, etc. In addition, reassigning tokens upon each barrier impairs intra-warp locality and may lead to unnecessary inter-warp thrashing. Furthermore, they also concentrated on L1 only and validated using the GPGPU-Sim simulator. However, as discussed in Section 5.2 and Section 5.9, the simulator does not accurately simulate the complete behavior of the GPU caches. Our work confirms that cache bypassing can derive performance on real hardware, in a much simpler software approach that is transparent and adaptive.

5.8 Conclusion

In this chapter, we proposed an adaptive cache bypassing framework for GPUs. It used a straightforward approach to throttle the number of warps that could access the three types of GPU caches – L1, L2 and read-only caches, thereby avoiding the fierce cache thrashing of GPUs. Our design is purely software-based thus is able to benefit existing platforms directly. It is easy to implement and is transparent to both the users and the hardware. We validated the framework on seven GPU platforms that covered all GPU generations. Results showed that adaptive bypassing could bring significant speedup over the general cache-all and bypass-all schemes. We also analyzed the performance variation across the platforms and the applications. In addition, we proposed software and hardware approaches to further reduce bypassing overhead and provided several optimization guidelines for the utilization of GPU caches.

5.9 Further Discussion

In this section, we first show the experiment figures for the four extra GPU platforms that are not shown in the main context. We then show the simulation results for the hardware approach that attempts to reduce bypass overhead. Finally, we analyze the performance patterns of the applications with respect to different bypassing threshold, which may explain why certain applications can benefit more significantly from cache bypassing than others.

5.9.1 Addtional Experiment Results

In this section, we show the experiment figures for the four additional GPU platforms (Platform-4 to 7) which are not included in the main context of this chapter. The platform information is also listed in Table 5.2. The application information is listed in Table 5.3. The results for 16KB L1, 48KB L1 and L2 cache bypassing on Fermi GPU with CC-2.1 are illustrated in Figure 5.23, 5.24 and 5.25. The results for 16KB, 32KB, 48KB L1 and L2 cache bypassing on Kepler GPU with CC-3.0 are shown in Figure 5.26, 5.27, 5.28 and 5.29. The results for 16KB, 32KB, 48KB L1, read-only cache and L2 cache bypassing on Kepler GPU with CC-3.5 are illustrated in Figure 5.30, 5.31, 5.32, 5.33 and 5.34.

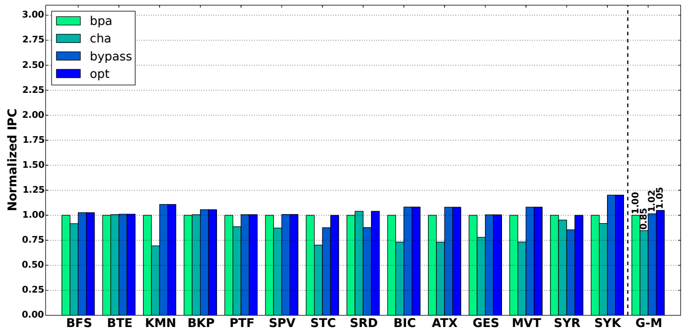

Figure 5.23: 16KB L1 bypassing on Fermi CC-2.1.

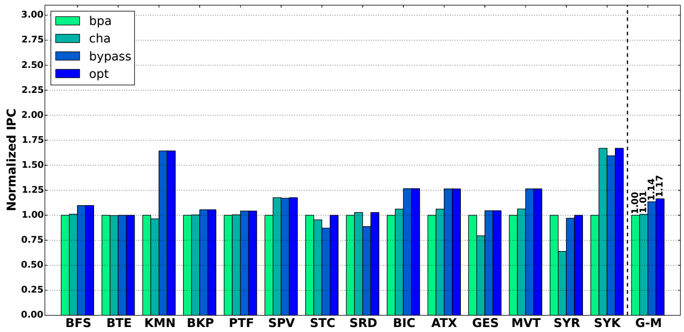

Figure 5.24: 48KB L1 bypassing on Fermi CC-2.1.

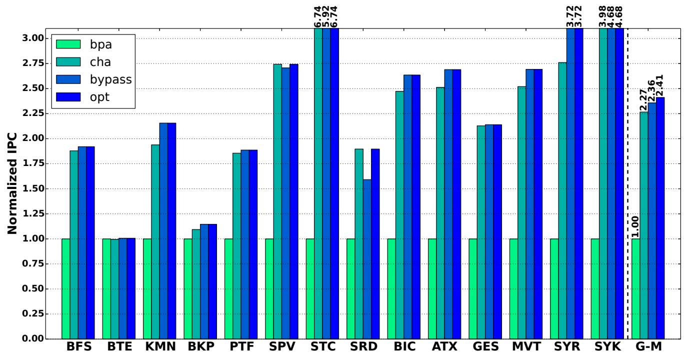

Figure 5.25: L2 bypassing on Fermi CC-2.1.

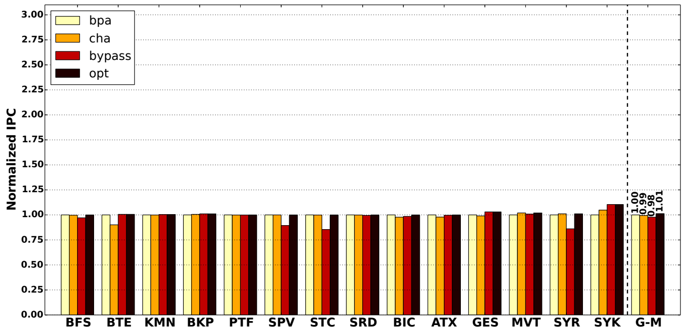

Figure 5.26: 16KB L1 bypassing on Kepler CC-3.0.

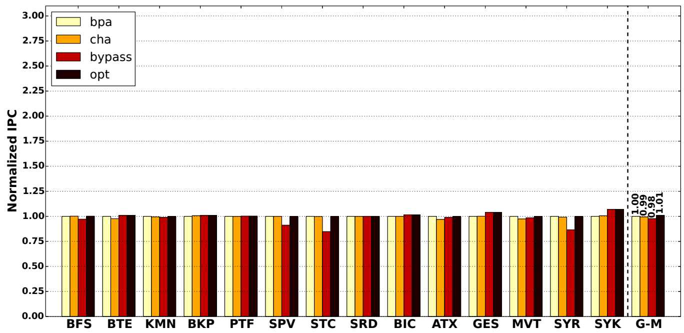

Figure 5.27: 32KB L1 bypassing on Kepler CC-3.0.

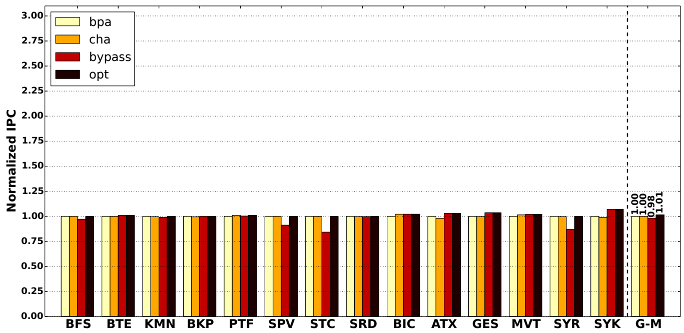

Figure 5.28: 48KB L1 bypassing on Kepler CC-3.0.

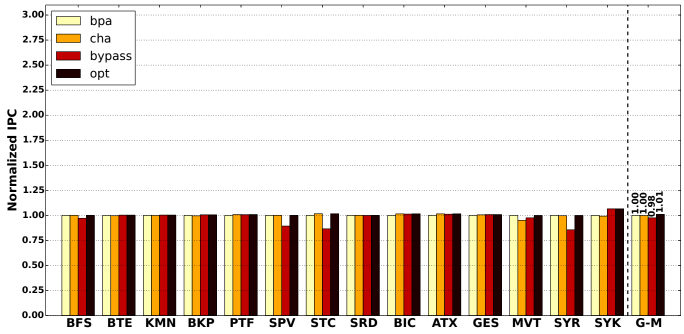

Figure 5.29: L2 bypassing on Kepler CC-3.0.

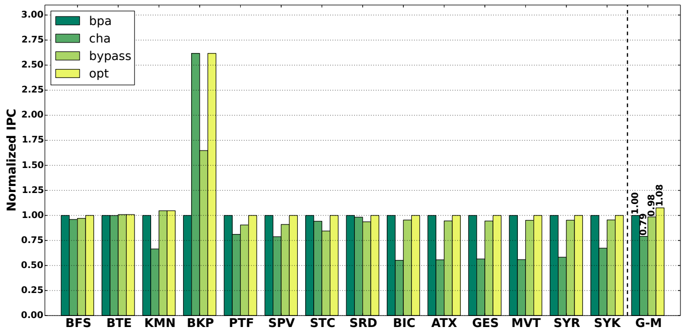

Figure 5.30: 16KB L1 bypassing on Kepler CC-3.5.

Finally, the results for read-only cache and L2 cache bypassing on Maxwell GPU with CC-5.2 are shown in Figure 5.35 and 5.36.

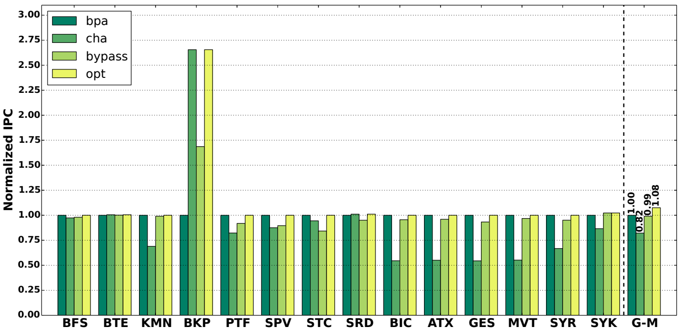

Figure 5.31: 32KB L1 bypassing on Kepler CC-3.5.

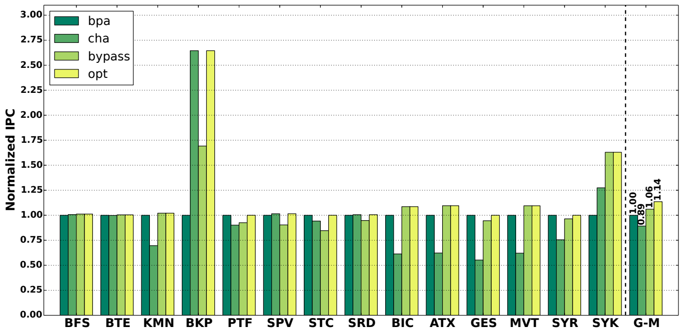

Figure 5.32: 48KB L1 bypassing on Kepler CC-3.5.

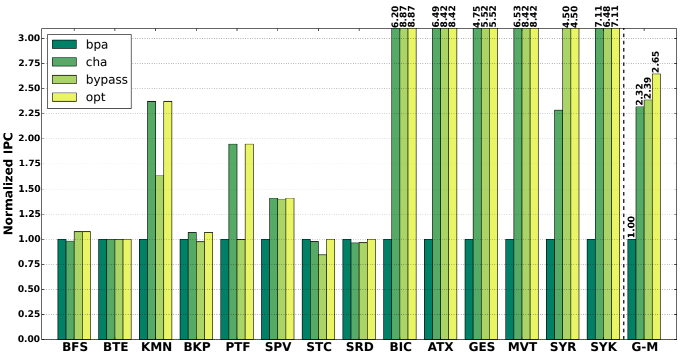

Figure 5.33: Read-only cache bypassing on Kepler CC-3.5.

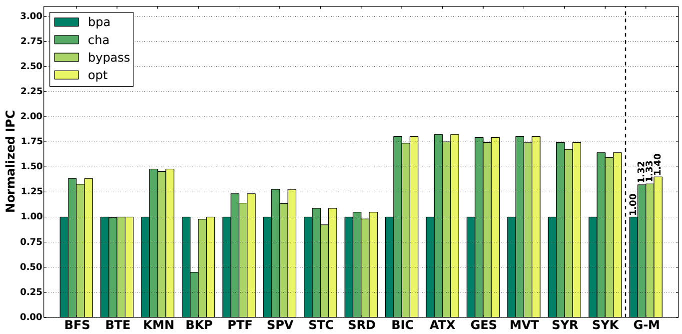

Figure 5.34: L2 bypassing on Kepler CC-3.5.

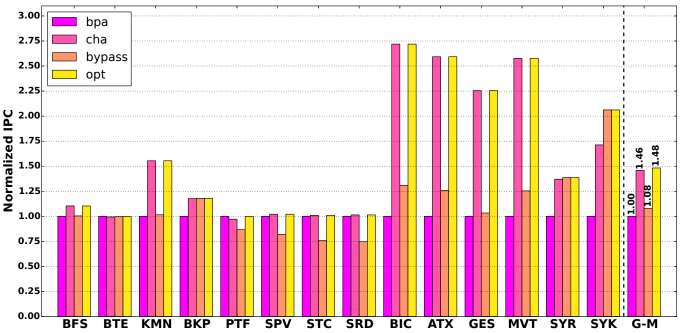

Figure 5.35: Read-only cache bypassing on Maxwell CC-5.2.

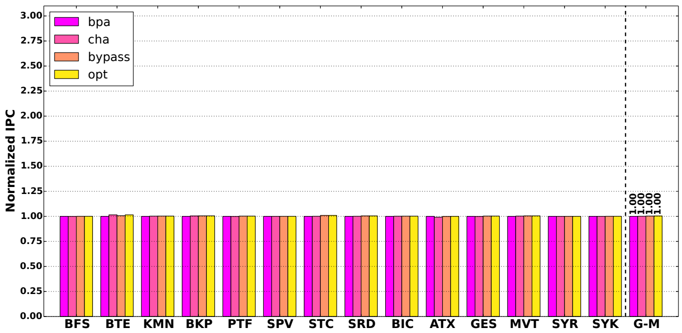

Figure 5.36: L2 bypassing on Maxwell CC-5.2

5.9.2 Hardware Design

In this section, we discuss the possibility to reduce bypassing overhead via hardware approach. The idea is to implement the judging process of bypassing (shown in Listing 5.1) in the cache controller instead of in the program. We use a 6-bit register[6] to conserve the bypassing threshold. The register is configured when the kernel launches. Then for a memory request, upon it arrives at the cache, its warp index is compared with the threshold register; if the value is less, it is appended to the cache waiting queue, otherwise, it is forwarded to the request queue of the lower-level memory devices. For example, if bypassing L1, the request is forwarded to the *MRQ* [153] and is later injected into the

[6] As discussed in the conference paper, the maximum number of warps is 32.

Table 5.4: GPGPU-Sim Configurations

Architecture	Fermi (GTX480), 15 SMx32, 700MHz
L1 cache	16KB, 32 sets, 128 B/line, LRU, 32 MSHRs
L2 cache	768KB, 6 channels, 64 sets, 128 B/line, LRU, 32 MSHRs
DRAM	6 MCs, FR-FCFS

interconnection network (Figure 1 of the conference paper).

Migrating the bypassing functionality into the hardware eliminates the 1-bit predicate register cost per thread as well as the corresponding assessment upon each time's memory access, which improves performance and reduces power. We implemented such a design in GPGPU-Sim Version 3.3.2 [43] with the power module GPUWattch [160].

The simulation configuration is shown in Table 5.4. We compare the performance and power for `cha`, `bpa`, the `software` and `hardware` implementations with the optimal threshold value profiled. The results are shown in Figure 5.37 and 5.38 for performance and power. Note, we do not include the applications of *syrk* and *syr2k* because simulation of them takes days and still cannot finish.

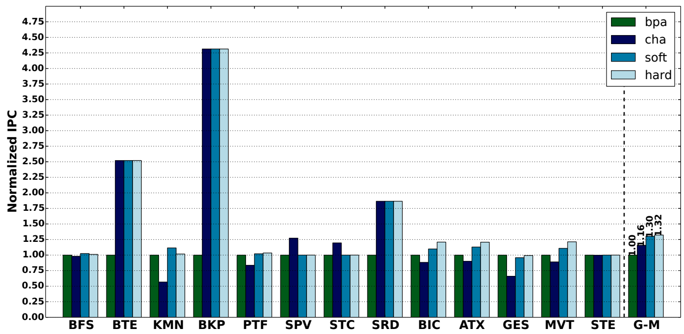

Figure 5.37: Simulation Results for Normalized IPC.

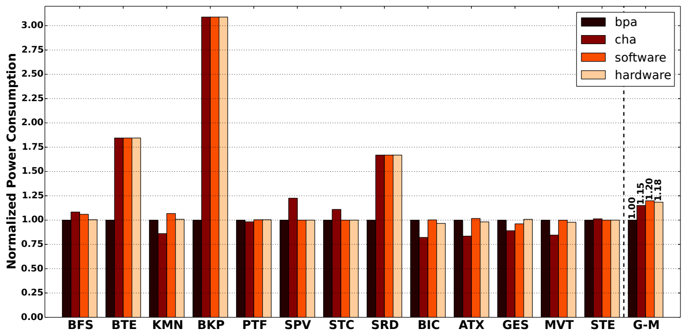

Figure 5.38: Simulation Results for Power.

Comparing Figure 5.37 with the real hardware testing results in Figure 5.12 of Section 5.5, there are evident mismatches, e.g. `bpa` is better than `cha` in real hardware, but is inferior in the simulation, `cha` of *SPV* and *STC* exhibit the best in simulation but are the worst in real hardware testing, etc. This is because GPGPU-Sim does not accurately mimic the complete behaviors of the real hardware. For example, based on our previous work [111], Fermi uses an XOR-based hashing for the L1 cache, but such module is not realized in GPGPU-Sim.

As can be seen from Figure 5.38, the hardware implementation can reduce the power consumption by 4% with respect to bpa. Without *SSC*, the figures are hardware:1.20x vs. software:1.18x, which is 2% differences. Note, although the improvement for the hardware implementation is not prominent, it is the simulation result for the Fermi architecture, on which the overhead introduced is already quite small (less than 4%, see Section 5.5). We expect more profit from Kepler and Maxwell, although only Fermi architecture is supported by the simulator.

5.9.3 Application Bypass Patterns

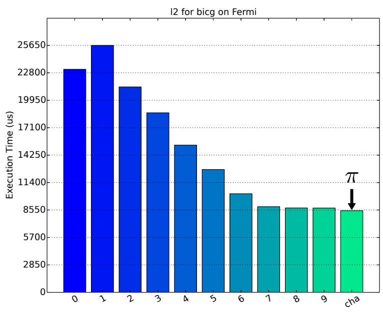

Figure 5.39: Cache-favorite: `BIC` on 16 KB L1.

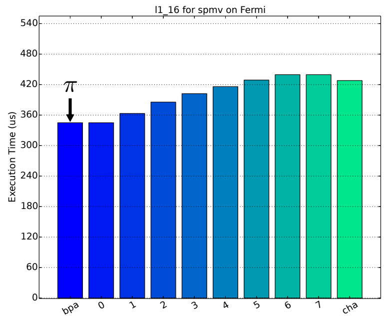

Figure 5.40: Bypass-favorite: `SPV` on 16 KB L1.

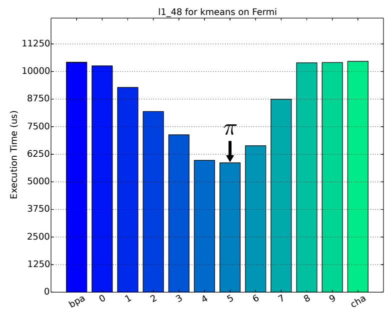

Figure 5.41: Cache-congested: `KMN` on 48 KB L1.

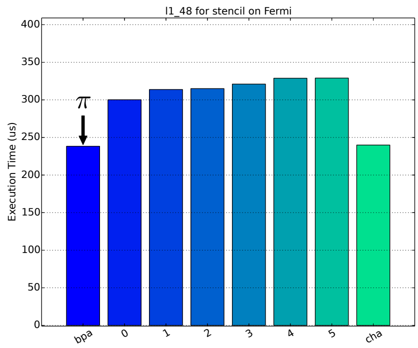

Figure 5.42: Cache-insensitive: `STE` on 48 KB L1.

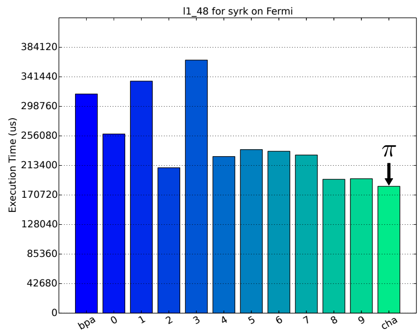

Figure 5.43: Irregular: `SYR` on 48 KB L1.

In this section, we show the typical figures for each of the application categories based on the performance trend according to the variation of the bypassing threshold. In Section 5.3, we characterize all the tested applications in Table 5.3 into five categories: *bypass-favorite*, *cache-favorite*, *cache-congested*, *cache-insensitive* and *irregular*. Here we show the figures for Fermi with CC-2.0 (i.e. Platform-1) as the examples.

- **Bypass-favorite**: As shown in Figure 5.40, the performance of bypass-favorite applications continuously degrades with a higher bypass threshold. `bpa` is the best choice. Applications such as *atax, gesummv, mvt, particlefilter* for 16KB L1 in Kepler CC-3.5 and CC-3.7 belong to this category.
- **Cache-favorite**: As shown in Figure 5.39, for cache-favorite applications, the performance keeps increasing with higher threshold. `cha` is the optimal choice. Most applications on L2 of Fermi and Kepler fall in this category (Maxwell does not essentially supports L2 bypassing, as discussed in Section 5.1).
- **Cache-congested**: As shown in Figure 5.41, for cache-congested applications, the curves are convex which looks like a bowl. Therefore, the optimal value falls in the middle. Applications such as *bfs, kmeans, bicg, mvt*, etc fall in this category and demonstrate the best bypassing performance.

- **Cache-insensitive**: As shown in Figure 5.42, the performance of cache-insensitive applications keeps almost steady with respect to bypassing threshold. For these applications (such as *stencil* and *streamcluster*) both `bpa` and `cha` show much better performance than adding the bypass framework. Meanwhile, `bpa` and `cha` are quite similar. Cache-insensitive applications show the worst performance for cache bypassing as it only introduces overhead. This scenario can be obtained in all figures with the application *stencil*.
- **Irregular**: As shown in Figure 5.43, irregular applications show a messy shape that no clear trends are shown. *syrk* and *syr2k* are in this category.

CHAPTER 6

GPU Compute Units Optimization: *SFU-Driven Transparent Approximation Acceleration*

Approximate computing, the technique that sacrifices certain amount of accuracy in exchange for substantial performance boost or power reduction, is one of the most promising solutions to enable power control and performance scaling towards exascale. Although most existing approximation designs target the emerging data-intensive applications that are comparatively more error-tolerable, there is still high demand for the acceleration of traditional scientific applications (e.g., weather and nuclear simulation), which often comprise intensive transcendental function calls and are very sensitive to accuracy loss. To address this challenge, we focus on a very important but long ignored approximation unit on today's commercial GPUs — the special-function unit (SFU), and clarify its unique role in performance acceleration of accuracy-sensitive applications in the context of approximate computing. To better understand its features, we conduct a thorough empirical analysis on three generations of NVIDIA GPU architectures to evaluate all the single-precision and double-precision numeric transcendental functions that can be accelerated by SFUs, in terms of their performance, accuracy and power consumption. Based on the insights from the evaluation, we propose a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. Our design is software-based and requires no hardware or application modifications. Experimental results on three NVIDIA GPU platforms demonstrate that our proposed framework can provide fine-grained tuning for performance and accuracy trade-offs, thus facilitating applications to achieve the maximum performance under certain accuracy constraints. This work has been presented at the 30th ACM International Conference on Supercomputing (ICS-16) [82].

6.1 Introduction

Despite the conventional belief that being exact remains the default attribute for computing, for many promising applications, such as big data, machine learning and multimedia processing, extremely accurate compliance of the produced results is often not an essential requisite. This undoubtedly offers new opportunities for application speedup or the associated power reduction at the expense of modest precision loss [161]. Such precision loss is only acceptable when it is within the tolerance range of the user-defined quality-of-service (QoS) [162], which heavily depends on the specific application domain. Besides, many of these applications are data-parallelism intensive, making them well-suited candidates for the emerging general-purpose GPU computation (GPGPU) [44]. Concerning the above reasons, approximate computing has become an attractive research topic for GPUs [81, 163, 78, 164, 165, 166].

However, most existing GPU approximation designs are targeted for data-intensive applications [81, 163, 164, 166], which are comparatively more error-tolerable. Furthermore, they primarily rely on the spatial or temporal locality (or reuse) among the nearby-data or the consecutive functions so as to approximate the requested data/computation based on their neighboring [81, 163, 165, 166] or locally stored historical values [163, 78, 164, 166]. Such approaches, although quite efficient, may commit uneven errors across data elements or even catastrophic failures since the locality is not always held and the distortion to the final results could be considerable. Moreover, for the numerical-intensive scientific applications (e.g., various simulation and molecular dynamics) that are usually sensitive to accuracy loss, the current techniques are often not suitable. This is because even a relatively smaller error introduced in an intermediate result may potentially propagate and be significantly amplified when such applications are deployed in a supercomputer environment with thousands of working GPUs [167, 168]. Therefore, gaining performance while offering lower but still tractable assurance on the accuracy loss becomes the major obstacle for applying approximation techniques to accuracy-sensitive applications on GPUs.

To address this challenge, we explore a very important but often ignored approximation unit on GPUs — the special-functional unit (SFU), and unveil its crucial role in performance acceleration for accuracy-sensitive scientific applications in the context of approximate computing. To better understand its approximation potentials, we first evaluate all the nine single-precision and four double-precision numeric transcendental functions that could be accelerated by SFUs, in terms of performance, accuracy and power. Using the insights, we then leverage the GPU SIMT execution model to dynamically partition warps into executing two versions of the numerical computation: an accurate but slower version and a faster but approximate version (i.e., using SFUs), and then tune this partition ratio to control the trade-offs between the performance and accuracy, or power and accuracy. This software approach successfully introduces a relatively large, uniform and fine-grained tuning space. To accompany this design, we also propose an efficient heuristic searching method to quickly locate the optimal partition ratio that delivers the best performance under user-defined QoS. Finally, we compact the approach and its searching method into a transparent, tractable and portable SFU-centric approximate acceleration framework, which is then validated on multiple GPU architectures for its effectiveness. This chapter makes the following contributions:

- This is the first work that specifically focuses on unleashing the approximation potentials of SFUs on GPUs. We explore its design, implementation, and fine-grained invocation methods. Also, we exhaustively evaluate the transcendental functions that can be accelerated by SFUs in terms of their latency, throughput, accuracy, resource cost, power, energy and the number of different operations contained.

Table 6.1: Invoking SP Transcendental Functions via CUDA and PTX APIs

Func.	CUDA API Intrinsics		PTX API Instructions	
	SP-Accurate Version	**SFU-Approximate Version**	**SP-Accurate Version**	**SFU-Approximate Version**
x/y	x/y	__fdividef(x,y) & –ftz=true	div.rn.f32 %f3,%f1,%f2;	div.approx.ftz.f32 %f3,%f1,%f2;
$1/x$	1/x	Not-Provided	rcp.rn.f32 %f2,%f1;	rcp.approx.ftz.f32 %f2,%f1;
$\sqrt{x}$	sqrtf(x)	Not-Provided	sqrt.rn.f32 %f2,%f1;	sqrt.approx.ftz.f32 %f2,%f1;
$1/\sqrt{x}$	1/sqrtf(x)	rsqrtf(x) & –ftz=true	sqrt.rn.f32 %f2,%f1; rcp.rn.f32 %f3,%f2;	rsqrt.approx.ftz.f32 %f2,%f1;
x^y	powf(x)	__powf(x) & –ftz=true	Very Complex	lg2.approx.ftz.f32 %f3,%f1; mul.ftz.f32 %f4,%f3,%f2; ex2.approx.ftz.f32 %f5,%f4;
e^x	expf(x)	__expf(x) & –ftz=true	Very Complex	mul.ftz.f32 %f2,%f1, 0f3FB8AA3B; ex2.approx.ftz.f32 %f3,%f2;
$\log(x)$	logf(x)	__logf(x) & –ftz=true	Very Complex	lg2.approx.ftz.f32 %f2,%f1; mul.ftz.f32 %f3,%f2, 0f3F317218;
$\sin(x)$	sinf(x)	__sinf(x) & –ftz=true	Very Complex	sin.approx.ftz.f32 %f2,%f1;
$\cos(x)$	cosf(x)	__cosf(x) & –ftz=true	Very Complex	cos.approx.ftz.f32 %f2,%f1;

- By leveraging the GPU SIMT execution model, we propose a runtime warp-partition method to introduce a fine-grained and nearly-linear tuning space for the performance-accuracy trade-offs on GPUs. This approach is well-suited for the scientific applications that enforce high accuracy constraints.
- Based on this approach, we propose a transparent, tractable and portable design framework to automatically tune the performance and accuracy of a GPU application and returns the best attainable performance subjecting to user-defined QoS. This framework can be integrated into the GPU compiler toolchain, hence bringing cheap, instant and significant performance gain with tractable assurance on accuracy loss.
- This is the first work to exploit hardware warp-slot id for fine-grained performance tuning and is the first to accelerate double-precision computation on GPUs via SFU-driven approximations.

6.2 SFU Design and Implementation

The basic knowledge about GPU and its various function units have already been discussed in Chapter 2. In this section, we zoom in specially on the SFUs and explore its design and operation. Based on the experiments on real hardware, we have observed interesting features of SFU implementation for approximating both SP and DP floating-point computation, which has not been covered by previous work.

6.2.1 SFU Design

To accelerate the commonly-used transcendental functions in numeric routines as well as the texture-fetching interpolation operations from graphic applications, NVIDIA GPUs since Fermi begin to integrate an array of special hardware accelerators in the SMs, called Special-Functional Units (SFUs).

Table 6.2: Invoking DP Transcendental Functions via CUDA and PTX APIs

Func.	CUDA API Intrinsics		PTX API Instructions	
	DPU-Accurate Version	**SFU-Approximate Version**	**DPU-Accurate Version**	**SFU-Approximate Version**
x/y	x/y	Not-Provided	div.rn.f64 %fd3,%fd1,%fd2;	rcp.approx.ftz.f64 %fd3,%fd2; mul.f64 %fd5,%fd3,%fd1;
$1/x$	1/x	Not-Provided	rcp.rn.f64 %fd2,%fd1;	rcp.approx.ftz.f64 %fd2,%fd1;
$\sqrt{x}$	x/y	Not-Provided	sqrt.rn.f64 %fd2,%fd1;	rsqrt.approx.ftz.f64 %fd2,%fd1; rcp.approx.ftz.f64 %fd3,%fd2;
$1/\sqrt{x}$	1/sqrt(x)	Not-Provided	sqrt.rn.f64 %fd2,%fd1; rcp.rn.f64 %fd3,%fd2;	rsqrt.approx.ftz.f64 %fd2,%fd1;

Table 6.3: Experiment Platforms. "Plat." stands for platform. "Dri./Rtm." stands for CUDA Driver/Runtime Version.

Plat.	GPU	Architecture	Code	CC.	Frequency	SMs	SPs	SFUs	Warp Slots	Memory Bandwidth	Dri./Rtm.
1	GTX-570	Fermi	GF-110	**2.0**	1464 MHz	15	32	4	48	152 GB/s	6.5/6.5
2	GTX-TitanZ	Kepler	GK-110	**3.5**	824 MHz	13	192	32	64	288 GB/s	7.5/6.5
3	GTX-750Ti	Maxwell	GM-107	**5.0**	1137 MHz	5	128	32	64	86.4 GB/s	7.5/6.5
4	Jetson TK1	Kepler	GK-20A	**3.2**	852 MHz	1	192	32	64	17 GB/s	7.0/7.0
5	Jetson TX1	Maxwell	GM-20B	**5.3**	998 MHz	2	128	32	64	25.6 GB/s	7.0/7.0

The numeric transcendental functions include *sine, cosine, division, exponential, power, logarithm, reciprocal, square-root* and *reciprocal square-root* [104, 169]. Their implementations are based on the quadratic interpolation method through *enhanced-minmax-approximations* in the hardware design [170]. Such an approximation process is accomplished in three steps: (1) a **preprocessing** step to reduce the input argument into a dedicated range, (2) a **processing** step to perform quadratic polynomial approximation on the reduced argument via table look-up for the required coefficients, and (3) a **postprocessing** step to reconstruct, normalize and round the result to its original argument domain. Please refer to [170, 171] for more details.

6.2.2 SFU Implementation

For **single-precision (SP)** floating-point computation, CUDA provides both an accurate implementation following IEEE-754 standard (labeled as **SPU version**) and an approximate implementation (labeled as **SFU version**) for the 9 transcendental functions, shown in Table.6.1. As can be seen, only 7 of the 9 transcendental functions have CUDA intrinsics. For the lower-level *Parallel-Thread-Execution* (PTX) assembly representation, we find that the SFU version for each transcendental function is comprised of a single or several SFU instructions, while the SP version is often a complex software-simulated procedure running on SPUs (or a procedure making modifications to the gross results obtained from the SFUs).

To initiate the SFU version, the two most naive approaches are (1) invoking the corresponding CUDA intrinsics (e.g., *__sinf* [172] in Table 6.1) within the program, or (2) specifying the compiler option "*-use_fast_math*" to force the utilization of the SFU version in the generated *cubin* binary. However, using "*-use_fast_math*" applies to the entire program, which prevents the transcendental functions to benefit from fine-grained tuning. For instance, "*-use_fast_math*" option implies "*-ftz=true*", which will flush all the denormal values (i.e., floating-point numbers that are too small to be representable in the current precision[1]) in the program to zero. Although this will speedup the processing for transcendental functions on SFUs, it also increases the inaccuracy of the normal SP computation. If we make "*-ftz=false*", it will however, decrease the maximum speedup for SFUs. Thus, "*-use_fast_math*" is not suitable for fine-grained performance tuning. On the other hand, using CUDA API intrinsics to exploit SFU also has two problems: (1) Not all of them are supported, e.g., $1/x$ and $\sqrt{x}$; and (2) the flush-to-zero (*-ftz*) configuration cannot be set/unset by the CUDA intrinsics. Table.6.1 shows that only the PTX instructions can provide the full coverage for all the 9 transcendental functions, and the flexibility to enable/disable the *-ftz* without affecting other transcendental functions and regular computation. We will further discuss this matter in Section 6.5.1.

Regarding **double-precision (DP)** floating-point computation shown in Table.6.2, **no** CUDA intrinsics are offered for approximating the nine functions. However, at the PTX assembly level, we discover that *reciprocal* ($1/x$) and *reciprocal-square-root* ($1/\sqrt{x}$) can be approximated for acceleration via SFUs. This is confirmed by checking the usage of "MUFU" instructions in the generated *cubin* binary, which are the instructions specifically targeted for SFU usage. With $1/x$ and $1/\sqrt{x}$, two other functions *div* and *square-root* can also be implemented indirectly. Therefore, there are in total four transcendental functions that can be approximated by SFUs for DP computation. To the best of our knowledge, no existing literature or tutorial has discussed how to employ these four SFU-based approximations to accelerate DP-based applications, as there is no support from either CUDA intrinsics or compiler options. We will demonstrate that, if they are properly used, significant performance improvements can be achieved for applications with intensive DP computation (see Section 6.5.3). Note that "*ftz*" is mandatory for these approximate functions in DP, i.e., the "*.ftz.*" suffix of the PTX instructions in Table 6.2. We label the DPU-based implementation as **DPU version**.

6.3 Measurement and Observation: Exploration of SP, DPU and SFU

First, we would like to study the runtime characteristics of the GPU transcendental functions (have not been explored previously) before they can be properly deployed into the real applications. In this section, we design dedicated microbenchmarks to measure the *latency*, *relative-error*, *register usage*, *SPU/SFU/DPU operations contained*, *throughput per SM* as well as *power* and *energy cost* for the 9 SP and 4 DP transcendental functions. This information will serve as the motivation of our proposed design.

Our evaluation platforms are listed in Table 6.3. Three generations of NVIDIA GPUs (*Platform 1,2,3*) including Fermi, Kepler and Maxwell, are used for testing the *function latencies*. For *relative-error*, we perform both SPU/DPU and SFU-based transcendental calculation over 100,000 random data and compare their results to the versions offered by the host Intel CPU. The average difference over the elements is then used as the relative-error. *Register usage* is collected based on the statistics reported

[1] Also known as underflow, it is $\pm 2^{-126}$ for SP and $\pm 2^{-1022}$ for DP.

Table 6.4: SPU Version vs. SFU Version Characterization for SP. Ver. stands for the version. Lat. is the measured latency in clock cycles. Rel-Err is the relative-error with respect to CPU results. Reg is the register consumption. F/P/D is the number of operations executed by SFU, SPU and DPU respectively in the function computation. T/M is the operation throughput per SM in the unit of Gop/s.

Func.	Arch.	Ver.	Lat.	Rel-Err.	Rg.	F/P/D	T/M	Func.	Arch.	Ver.	Lat.	Rela-Err.	Reg.	F/P/D	T/M
x/y	Fermi	SPU	2335	0	12	1/16/0	1.7	$1/x$	Fermi	SPU	1692	0	13	1/4/0	2.8
		SFU	2068	2.3433E-8	10	1/1/0	5.7			SFU	1651	1.1266E-8	8	1/0/0	5.7
	Kepler	SPU	1098	0	13	1/14/0	6.0		Kepler	SPU	715	0	14	1/4/0	7.9
		SFU	981	2.3433E-8	10	1/1/0	24.0			SFU	597	1.1266E-8	8	1/0/0	23.2
	Maxwell	SPU	236	0	14	1/14/0	4.1		Maxwell	SPU	219	0	14	1/4/0	4.7
		SFU	36	2.3433E-8	10	1/1/0	25.3			SFU	21	1.1266E-8	10	1/0/0	26.6
$\sqrt{x}$	Fermi	SPU	1708	0	10	1/6/0	2.6	$1/\sqrt{x}$	Fermi	SPU	1728	0	13	2/10/0	1.4
		SFU	1651	3.0763E-8	8	2/0/0	2.9			SFU	1651	2.7610E-8	8	1/0/0	5.7
	Kepler	SPU	711	0	10	1/6/0	6.4		Kepler	SPU	864	0	14	2/10/0	3.8
		SFU	613	3.0763E-8	8	2/0/0	12.9			SFU	597	2.7610E-8	8	1/0/0	23.2
	Maxwell	SPU	226	0	10	1/6/0	5.0		Maxwell	SPU	464	0	14	2/10/0	2.5
		SFU	47	3.0763E-8	10	2/0/0	14.8			SFU	21	2.7610E-8	10	1/0/0	27.1
x^y	Fermi	SPU	6073	3.0822E-8	14	3/59/0	8.0	e^x	Fermi	SPU	1681	2.3937E-8	10	2/7/0	1.9
		SFU	2110	8.0587E-8	10	2/1/0	43.1			SFU	1655	4.0603E-8	8	1/1/0	5.7
	Kepler	SPU	1496	3.0822E-8	15	3/60/0	9.1		Kepler	SPU	700	2.3937E-8	8	2/7/0	4.5
		SFU	997	8.0587E-8	10	2/1/0	156.7			SFU	612	4.0603E-8	8	1/1/0	23.4
	Maxwell	SPU	1029	3.0822E-8	16	3/60/0	3.8		Maxwell	SPU	160	2.3937E-8	8	2/7/0	4.7
		SFU	56	8.0587E-8	10	2/1/0	65.8			SFU	31	4.0603E-8	10	1/1/0	20.6
$\ln(x)$	Fermi	SPU	1779	4.6541E-9	11	1/19/0	1.2	$\sin(x)$	Fermi	SPU	1727	8.7079E-9	13	0/17/0	1.1
		SFU	1649	6.3260E-7	8	1/1/0	5.7			SFU	1660	9.6523E-7	8	1/0/0	5.7
	Kepler	SPU	834	4.6541E-9	11	1/19/0	2.1		Kepler	SPU	804	8.7079E-9	13	0/17/0	2.9
		SFU	608	6.3260E-7	8	1/1/0	22.9			SFU	602	9.6523E-7	8	1/0/0	25.0
	Maxwell	SPU	298	4.6541E-9	11	1/20/0	1.8		Maxwell	SPU	222	8.7079E-9	17	0/17/0	2.3
		SFU	38	6.3260E-7	10	1/1/0	26.3			SFU	25	9.6523E-7	10	1/0/0	22.5
$\cos(x)$	Fermi	SPU	1740	1.4455E-8	13	0/18/0	1.0								
		SFU	1646	1.1584E-6	8	1/0/0	5.7								
	Kepler	SPU	824	1.4455E-8	13	0/18/0	2.9								
		SFU	600	1.1584E-6	8	1/0/0	25.0								
	Maxwell	SPU	229	1.4455E-8	17	0/18/0	2.1								
		SFU	25	1.1584E-6	10	1/0/0	22.5								

Table 6.5: DPU Version vs. SFU Version Characterization for DP.

Func.	Arch.	Ver.	Lat.	Rel-Err.	Rg.	F/P/D	T/M	Func.	Arch.	Ver.	Lat.	Rela-Err.	Rg.	F/P/D	T/M
x/y	Fermi	DPU	1889	0	19	1/0/15	7.8	$1/x$	Fermi	DPU	2485	0	16	1/0/8	10.5
		SFU	1204	2.5561E-7	10	1/0/1	28.8			SFU	2166	2.5545E-7	8	1/0/0	42.3
	Kepler	DPU	1236	0	20	1/0/15	8.4		Kepler	DPU	774	0	14	1/0/10	13.0
		SFU	1104	2.5561E-7	10	1/0/1	30.4			SFU	902	2.5545E-7	8	1/0/0	44.9
	Maxwell	DPU	1793	0	20	1/0/15	2.2		Maxwell	DPU	1761	0	13	1/0/10	3.4
		SFU	2057	2.5561E-7	10	1/0/1	7.9			SFU	1346	2.5545E-7	9	1/0/0	11.7
$\sqrt{x}$	Fermi	DPU	2319	0	13	1/0/13	8.5	$1/\sqrt{x}$	Fermi	DPU	2551	0	16	2/0/21	5.3
		SFU	2171	2.8951E-7	10	2/0/0	42.1			SFU	2165	2.2110E-7	10	1/0/0	42.4
	Kepler	DPU	949	0	14	1/0/13	9.1		Kepler	DPU	1296	0	14	2/0/23	7.0
		SFU	921	2.8951E-7	8	2/0/0	44.3			SFU	897	2.2110E-7	8	1/0/0	44.9
	Maxwell	DPU	1947	0	14	1/0/13	2.4		Maxwell	DPU	3317	0	14	2/0/23	1.6
		SFU	1355	2.8951E-7	9	2/0/0	11.7			SFU	1340	2.2110E-7	9	1/0/0	11.7

by the CUDA compiler. For the *operation throughput per SM*, sufficient transcendental function calls are initiated in the microbenchmark and all of them are completely independent with each other to fully exploit the instruction-level-parallelism (ILP) of the hardware. We observe the profiled throughput curve until the values become stable, which are then used as the maximum sustainable throughput for that operation. These values are then divided by the SM number to get the per-SM

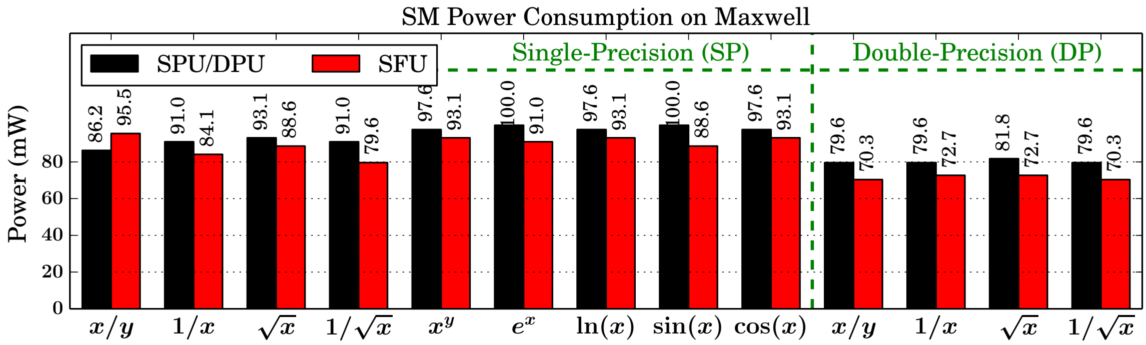

Figure 6.1: Power Consumption Measured on Jetson TX-1.

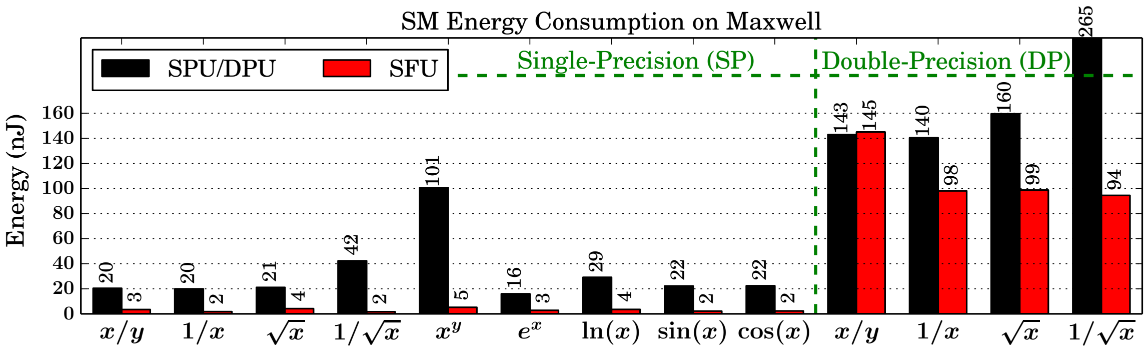

Figure 6.2: Energy Consumption for Jetson TX-1.

throughput. All these results are shown in Tables 6.4 and 6.5 for SP and DP, respectively.

The existing approaches to obtain GPU **power consumption** are often based on either simulator approximation (e.g., *GPUWattch* [160]) or the power-draw value reported by *nvidia-smi* [173]. However, neither of them reports real GPU power consumption. In this work, we propose a new approach that is more accurate and reliable. It leverages the latest Maxwell-based NVIDIA Jetson TX-1 GPU (*Platform 5* in Table 6.3, which is mainly designed for embedded utilization) and measures the power of the board's computation module only (i.e., the quad-core CPU and dual-SM GPU). This is achieved by measuring the voltage alteration of the resistance $R264$, which is in series with the computation module when a GPU kernel is running, and then compare it with the baseline state when the compute module is idle. Inside the kernel, we use a loop to keep the transcendental functions repeatedly being executed until the average voltage of the resistance converges to a steady value. As the voltage change is quite small, we also design an amplifier circuit so that such small voltage change can be sensitively tracked by an oscilloscope[2]. The measured power results are shown in Figure 6.1. We also tried to measure the power of the Kepler-based Jetson TK-1 board (*Platform 4* in Table 6.3). However, we found that there is no series resistance to the core module for this board. The only one

[2]The resistance $R264$ is in series with the compute module. The voltage difference measured by the oscilloscope in a long steady state, after being divided by the amplification factor, is then divided by the resistance value $R264 = 0.005\Omega$ to obtain the electric current of the compute module. The current is then multiplied by the measured $Vdd = 19.6V$ to acquire the actual GPU power consumption.

that seems promising (i.e., resistance *R5C11*) is in series with the entire board (including GDDR, fan and other I/O modules), so the voltage is quite hard to stabilize. Thus, we do not show the TK-1 power results in this chapter. With the measured power, we can calculate the energy consumption with the measured function latencies. The energy results are shown in Figure 6.2.

Table 6.4 and 6.5 show that the SFU itself only injects small errors in the individual function calculation. However, these small errors can quickly propagate and get amplified across the program semantics, causing intolerable accuracy for some applications. Also, dramatic differences in latency and throughput have been observed between SPU and SFU versions on both Kepler and Maxwell platforms. Furthermore, we find that latency is not as good as throughput per SM (T/M) for indicating the real performance difference between the two versions. For example, $\ln(x)$'s throughput difference on Kepler is as high as 9.9x, while the latency difference is only 37%. This implies that the SFU appears to be a super-pipelined unit. For power and energy, Figures 6.1 and 6.2 show that (1) the power consumption using SPU/DPU is slightly higher than that using SFU, except for x/y in SP; and (2) due to the huge performance differences between the SP and DP versions on the Maxwell platform, the overall energy consumption of DP versions (including their SFU approximations) is significantly higher than that of the SP versions, in spite of their lower power. These observations motivate us to propose our design for tackling the performance-accuracy trade-offs using SFU approximation on GPUs, which will be discussed next.

6.4 SFU-Driven Approximation Acceleration: A Software Approach

From the experiments, we observe that SFUs can significantly boost the performance for transcendental-function intensive applications. But meanwhile their approximations also introduce errors that are sometimes too large to be accepted. Although Table 6.4 and 6.5 demonstrate that SFUs only introduce relatively small errors in each transcendental computation, the process about how these small errors propagate and eventually accumulate to intolerable results is often complicated. This is the reason why within a single thread context choosing the proper functions to approximate while keeping the overall error under control remains quite difficult [174, 175, 176]. Additionally, compared with the data-intensive applications, the numerically intensive applications are often much more sensitive to accuracy. Therefore, a fine-grained accuracy tuning scheme is in great demand so that the most desirable performance can be achieved under more strict accuracy requirement. Ideally, such a fine-grained tuning range should be within a small accuracy offset and comprises consecutive accuracy tuning points. In other words, applied techniques should be controlled to some extent and not cause *significant accuracy difference between two discrete tuning points* (e.g., techniques such as loop perforation [176] and specific optimization transformations [81] often cause large accuracy differences between tuning points).

GPU offers massive identical threads operating upon different data elements. If part of the threads on GPU could execute the approximate version while the remaining ones process the accurate version (such a design paradigm is labeled as **horizontal design**), it essentially opens the door to a new design direction that is perpendicular to the conventional ones, which seek to choose the appropriate functions for approximation in a single-thread context (labeled as **vertical design**). Comparatively, the horizontal design should have a much simpler and more tractable accuracy-performance trade-off relationship than the vertical one, as the error effects are similar from various threads but very different across functions. We will demonstrate our exploration on the trade-off relation between performance and accuracy for the proposed horizontal design in Section 6.5.3. In fact, the horizontal design is one of the most highlighted features that differentiates a GPU from the CPU family, which can also be applied to resolve other design trade-offs, such as the one between thread volume and cache-performance in Chapter 5.

Furthermore, the parallelism granularity is an important issue for enabling the horizontal design. Since warp divergence incurs significant overhead, instead of working at the fine-grained thread level, we focus on the medium-grained warp level to reduce the design space and eliminate the warp-divergence overhead. For the rest of this chapter, we will demonstrate how to *practically and properly schedule the candidate warps between the accurate but slower SPU/DPU version and the approximate but faster SFU version*. More specifically, we will answer the following questions:

- How to implement the SPU/DPU and SFU versions of transcendental functions in a fine-grained flexible way (i.e., for each computation rather than for the whole kernel)?
- How to control the approximation degree?
- How to decide the optimal warp scheduling so that the best performance can be achieved under a QoS constraint?

6.4.1 Flexible SPU/DPU/SFU APIs Invocation

There are three types of APIs that can be applied for approximating transcendental functions on GPU: *CUDA*, *PTX* and *SASS* (see Section 2.3.2). Modifying SASS code requires enormous knowledge about the detailed hardware implementation, which is often concealed by the vendors. Migration is also very difficult for SASS code because it is hardware specific. Most importantly, there is no official SASS assembler. Therefore, SASS is excluded as an option to implement approximation.

On the other hand, PTX APIs are the specific PTX instructions, as listed in the right side of Table 6.1. As previously discussed, for the SFU version, all the 9 transcendental functions can be approximated via PTX APIs in the following format with at most three instructions:

```
function.approx.ftz.f32 %f3, %f1, %f2;
```

"*approx*" stands for the approximate version, "*ftz*" indicates that flushing-to-zero is true for denormal values, and "*f32*" is for SP. However, for the accurate SPU version, we discover that only *div, rcp, sqrt* and *rsqrt* can be expressed via 1 to 2 PTX instructions. The other five transcendental functions require complex representations when using PTX instructions. For instance, for *sin* and *cos*, the

```
//CUDA API to implement accurate SPU version
float expRT = expf(-R*T);
//PTX API to implement approx SFU version with denormal
asm("mul.ftz.f32 %0, %1, 0f3FB8AA3B;":"=f"(tmp):"f"(-R*T));
asm("ex2.approx.ftz.f32 %0, %1;":"=f"(expRT):"f"(tmp));
```

Listing 6.1: CUDA-based SPU version vs. PTX-based SFU version.

SPU-based implementations contain more than 140 lines of PTX code without counting the loops inside. Manipulating such a big block of PTX routines while keeping consistent with its upper and lower context (e.g., register naming, memory consistency, etc.) remains very tedious and error-prone. Therefore, we cannot implement both accurate and approximate transcendental computation on GPU solely with PTX instructions.

As discussed in Section 6.3, all the SPU-based CUDA APIs have their original expressions, shown in the left side of Table 6.1. But for the SFU approximation, *reciprocal* and *square-root* do not have their CUDA intrinsics; the only option is to recompile the entire source file with "*-use_fast_math*". However, this is too coarse-grained and may affect other kernels unexpectedly. Moreover, one cannot flexibly control the denormal behavior for a single function by using CUDA intrinsics in the SFU approximation version. Specifying *-ftz=true/flase* would change all the kernels in the current source file.

To summarize, CUDA APIs cover all the accurate SPU versions and show the convenience for program transformation, while PTX APIs cover all SFU versions and offer the maximum flexibility for approximation. Therefore, our design combines the two via the embedded PTX [177]. Listing 6.1 for example shows the two versions of the *exp* function.

Note that there is another strong reason for implementing the SPU versions via PTX APIs. As shown in Table 6.2, there is no CUDA intrinsics offered at all for the DP approximation. This chapter proposes the first SFU-driven approximation approach for DP computation via PTX APIs on GPU.

6.4.2 Control Approximate Degree Horizontally

A way is needed to control the approximation degree such that the trade-offs between performance and accuracy can be made according to the required QoS. Ideally, to allow fine-grained tuning, the approximation degree range should be relatively large (within in a certain accuracy expectation though) while the gap between discrete degrees remains small. In our horizontal design, this is achieved by *tuning the partition of the homogeneous warps between the SPUs/DPUs and the SFUs.*

Our basic approach is that we set a threshold for the approximate degree (labeled as λ) at the beginning of the kernel. In case a transcendental function is invoked, during its execution,

- for warps with hardware index less than the threshold ($warp\_id < \lambda$), they use the SFU version via embedded PTX instructions.

```
#define PI 3.14159265358979f
__device__ inline void BoxMuller(float& u1,float& u2){
  float r=sqrtf(-2.0*logf(u1)); float phi=2*PI*u2;
  u1=r*cosf(phi); u2=r*sinf(phi);
}
__global__ void BoxMullerGPU(float *d_Random,int nPerRng){
  const int tid=blockDim.x*blockIdx.x+threadIdx.x;
  for (int iOut=0;iOut<nPerRng;iOut+=2)
    BoxMuller(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
              d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
}
```

Listing 6.2: The Original Mersenne Kernel.

- for warps with hardware index larger than or equal to the threshold ($warp\_id \geq \lambda$), they perform the SPU/DPU version via CUDA APIs.

The warp index used here is not the common software warp-id in the programming context calculated by dividing the thread-id with the warp size, but essentially the hardware warp-slot id of a GPU SM, which can be acquired by fetching from the special register – "*%warpid*" via PTX instructions. There are three reasons for using the hardware warp-id in our design: (1) The hardware warp-ids contain a larger tuning range, since its corresponding warp-slots are for an entire SM while the software warp-ids are only for a CTA. More specifically, an SM usually accommodates multiple CTAs (up to 16 for Kepler and Maxwell), so tuning according to hardware warp-slots is more fine-grained. For example, assume a SM has 16 CTAs and each contains 4 warps. Therefore, all the warp-slots of the SM are occupied and the occupancy is 1. If software warp-id is used to partition the warps, the tuning range is from 0 to 4. However, if the hardware warp slot id is applied, the tuning range becomes from 0 to 64 (48 for Fermi, see Table 6.3). (2) Using hardware warp slot ids can achieve better load-balancing. Unlike using software warp-ids, warps are dynamically binded to the hardware warp-slots at runtime. This will average out the scenarios where some warps are always scheduled and consequently finished earlier than other warps in a CTA (i.e., the starvation problem). For example, specifying "*if* $warp\_id < 8$" using hardware warp-id has almost the same performance as the scenarios such as *if* $warp\_id \geq 56$ and *if* $warp\_id < 4$ *or* ≥ 60. (3) The change of approximate degree is 1 warp among two consecutive tuning steps for using hardware warp-slot id, but *num_CTA* per SM for using software warp-id. (4) Obtaining the hardware warp-id can be completed in a single register-read operation. However, it requires an additional integer division (or right-shifting) instruction to gain software warp-id. Additionally, when transcendental functions are invoked inside a loop, to reduce the branching overhead (though there is no warp-divergence), we put the warp partition process outside the loop to reduce its overhead.

We demonstrate this process using an example. Listing 6.2 shows the the BoxMullerGPU kernel from Mersenne [42], in which *log, sqrt, sin* and *cos* functions are invoked repeatedly inside a "*for*" loop. Listing 6.3 shows the modified SFU-driven approximate tuning kernel. As can be seen, a new approximate device function "`BoxMuller_sfu`" is generated using embedded PTX for the SFU version. Then by specifying the "`Lambda`" variable either statically at compile-time or dynamically

```
#define PI 3.14159265358979f
__device__ inline void BoxMuller_sfu(float& u1,float& u2){
  float r, t1, t2; float phi=2*PI*u2;
  asm("lg2.approx.ftz.f32␣%0,␣%1;":"=f"(t1):"f"(u1));
  asm("mul.ftz.f32␣%0,␣%1,␣0f3F317218;":"=f"(t2):"f"(t1));
  asm("sqrt.approx.ftz.f32␣%0,␣%1;":"=f"(r):"f"(-2.0*t2));
  asm("cos.approx.ftz.f32␣%0,␣%1;":"=f"(u1):"f"(phi));
  asm("sin.approx.ftz.f32␣%0,␣%1;":"=f"(u2):"f"(phi));
  u2=u2*r; u1=u1*r;
}
__global__ void BoxMullerGPU(float *d_Random,int nPerRng){
  const int tid=blockDim.x*blockIdx.x+threadIdx.x;
  unsigned warpid;
  //const bool flag=(threadIdx.x>>5)<Lambda;//software_warp_id
  asm("mov.u32␣%0,␣%%warpid;":"=r"(warpid));//hardware_warp_id
  const bool flag=(warpid<Lambda);//approx degree
  if(flag){//SFU approximate version
    for(int iOut=0;iOut<nPerRng;iOut+=2)
      BoxMuller_sfu(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
                    d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
  }else{//SPU accurate version
    for(int iOut=0;iOut<nPerRng;iOut+=2)
      BoxMuller(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
                d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
  }}
```

Listing 6.3: Transformed Mersenne Kernel.

at runtime, we are able to change the partition of warps between SFUs and SPUs, which serves as the approximate degree for fine-tuning the trade-offs between performance and accuracy.

The overhead of the proposed design is very small. Since we work at the medium-grained warp level, warp-divergence is avoided. In terms of spatial overhead, only the flag variable has a lifetime across the kernel and costs a 1-bit predicate register per thread. Furthermore, as observed in Table 6.4 and 6.5, the SFU versions always consume fewer registers than the SPU versions. Therefore, adding a branch statement (i.e., *if-else*) should not incur additional registers (in this way the occupancy keeps unchanged). Also, because the predicate-register checking is internally supported by the GPU hardware as one stage of the pipeline, the only overhead is the issuing delay for this extra branching. Such branching overhead can be significantly mitigated by being moved outside the loop, as shown in Listing 6.3. Other overheads such as the delay for fetching the hardware warp-id, comparing with the threshold and setting the flag (i.e., the predicate register [88]) are negligible.

6.4.3 Exploring Performance-Accuracy Trade-off

In this subsection, we explore the trade-off relationship between performance and accuracy on a wide range of scientific applications using the approach discussed previously. By doing so, we can build a strategy to answer how to decide the optimal approximate degree to achieve the best performance under certain QoS. We select applications that contain transcendental numeric functions in their kernels from Rodinia [37], Parboil [38], SDK [42], Polybench [40] and Shoc [39] benchmark

Table 6.6: Benchmark Characteristics

Application	Description	abbr.	Domain	Hotspot Kernel	Transcendental Funs	Ref
BlackScholes	Black-scholes option pricing	BLA	Compute Finance	BlackScholesGPU	*sqrt,div,log,exp,rcp*	[42]
single	Monte Carlo single Asian option	SIN	Compute Finance	generatePaths	*sqrt,exp*	[42]
MonteCarlo	Monte-Carlo option pricing	MCO	Compute Finance	MonteCarloKernel	*exp*	[42]
cp	Coulombic potential	COP	Molecular dynamics	cenergy	*rsqrt*	[178]
cutcp	Distance-cutoff coulombic potential	CUT	Molecular dynamics	lattice6overlap	*rsqrt*	[38]
lavaMD	Particle potential and relocation	LAV	Molecular dynamics	kernel_gpu_cuda	*exp*	[37]
nbody	Fast n-body simulation	NBO	Molecular dynamics	integrateBodies	*rsqrt*	[42]
oceanFFT	FFT-based ocean simulation	OCN	Molecular dynamics	generateSpectrum	*rcp,sqrt,sin,cos*	[37]
backprop	Back propagation	BKP	Machine Learning	layerforward	*pow,log*	[37]
nn	K-nearest neighbors	KNN	Machine Learning	euclid	*sqrt*	[37]
corr	Correlation computation	COR	Linear algebra	reduce_kernel	*div,sqrt*	[40]
gaussian	Gaussian elimination solver	GUS	Linear algebra	Fan1	*div*	[37]
mersenne	Mersenne-twister random generator	MEN	Simulation	BoxMullerGPU	*log,sqrt,sin,cos*	[42]
cfd	Redundant flux computation	CFD	Simulation	comp_step_factor	*sqrt,rcp,div*	[37]
s3d	Combustion process simulation	S3D	Simulation	ratt2_kernel	*div*	[39]
mri-q	Q matrix for MRI reconstruction	MRQ	Image processing	ComputeQ_GPU	*sin,cos*	[38]
bilateralFilter	Bilateral smoothing filter	BIF	Image processing	d_bilateral_filter	*div,exp*	[42]
srad	Speckle reducing anisotropic diffusion	SRD	Image Processing	srad	*rcp,div*	[37]
grabcutNPP	GrabCut with NPP	NPP	Image Processing	GMMDataTerm	*log,exp*	[42]
imageDenoising	Image Denosing	IMD	Image Processing	KNN	*exp,rcp*	[42]

suites, as listed in Table 6.6. We apply the program transformation discussed and plot the curves of normalized application execution time and relative-errors[3] (against the SPU/DPU version) with respect to the variation of approximate degree λ on *Platform-1,2,3* in Table 6.3. The figures for the 20 single-precision applications on Maxwell are shown in Figure 6.3. We also plot the figures for the 4 applications that contain double-precision computation in Figure 6.4. Since the shapes on Fermi and Kepler are similar, they are omitted here. From the figures, we have the following observations:

1. Without considering the accuracy loss, our SFU-driven method demonstrates very significant performance speedup on the commodity GPU hardware (e.g., up to 5.1x for SP on Maxwell). We want to particularly highlight the DP scenarios (e.g., `CFD, S3D` and `COR`), as conventional wisdom believes SFU is specific for SP acceleration on GPUs. Based on our finding, other than directly programming in embedded PTX, there is currently no other software-level approach that can easily achieve such a kind of DP acceleration.

2. Although the performance gains from using SFU versions are impressive, they do incur accuracy losses. For some cases, these losses are intolerable for scientific applications (e.g., `BLA, CUT, NB, GUS, MEN, CFD, MRQ`) because the SP/DP version on GPU is already not as accurate compared to the CPU counterpart (see Table 6.4). Note that these applications are only small benchmarks or proxy applications on a single GPU that are available to us. In the future, when large-scale numeric applications containing hundreds of these proxy kernels run on thousands of GPU nodes in a supercomputer, a relatively small distortion to a result (e.g., `COP` on SP and `COR` on DP) can result in a significantly erroneous outcome. Thus, there is a clear trade-off between performance gain and accuracy loss.

[3] How to calculate the QoS for applications from various domains still misses a unified approach [179]. Here we use mean-relative-error as an example. However, other metrics can be applied to our design as well via the replacement of the error-calculation method.

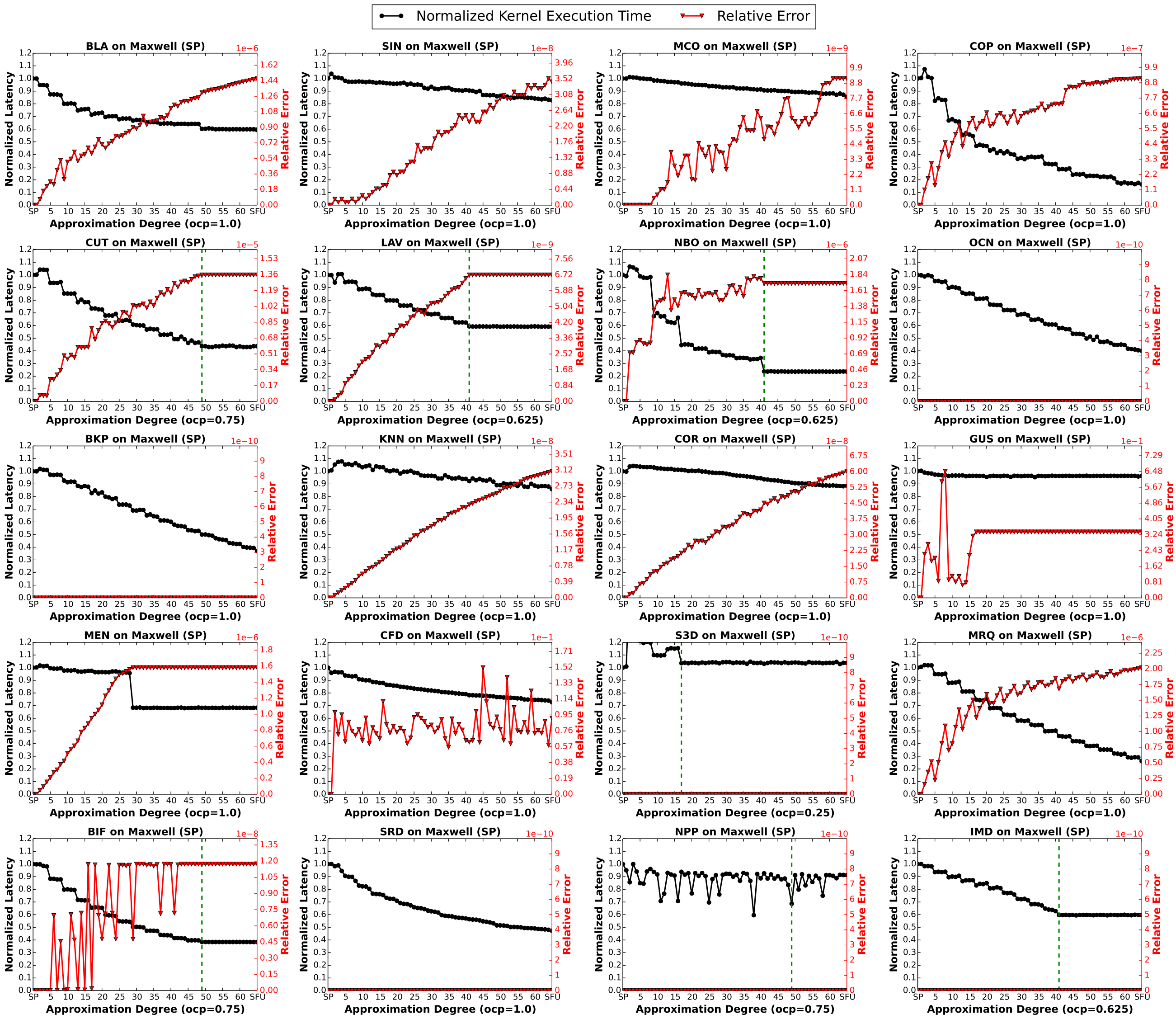

Figure 6.3: Performance-Accuracy Trade-offs for SP Applications on Maxwell GPU. The green dot line is based on the occupancy (i.e., *ocp* in the x-label). It indicates the border of the tuning space beyond which both the time and error curves keep steady. The error is relative to the pure SPU version.

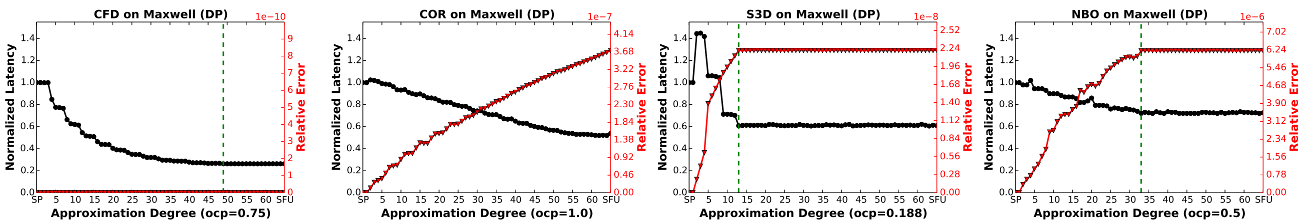

Figure 6.4: Performance-Accuracy Trade-offs for DP Applications on Maxwell GPU.

3. Different to our expectation that the point for best performance might be located in the middle of the curve, where SFUs and SPUs are exploited simultaneously, the results show that using our approach, the best performance is **almost always** achieved when all the warps are executed

in SFUs, while the worst when all of them are executed in SPUs/DPUs [4]. Correspondingly, the least accuracy loss occurs for pure SPUs/DPUs while the most for pure SFUs.

4. More importantly, the results show that the **trade-off relationship between performance and accuracy with respect to approximate degree is nearly-linear**. There are five obvious exceptions here: `OCN, BKP, SRD, NPP` and `IMD`. All of them represent the scenario where kernels use SP floating-point as the basic data-type during initial computation, and then convert them to integers for the finial results of the applications. This actually matches their domains, which are image processing and machine learning.

5. For some figures, there appears a flat region at the end of the curve where the performance and accuracy become constant (i.e., beyond the green dot line). This is because for some applications, not all the hardware warp-slots are fully occupied due to the low occupancy (e.g., cases with $ocp < 1$ in Figure 6.3 and 6.4). For example, the performance and accuracy when setting $\lambda = 49 \sim 64$ are essentially the same as those under $\lambda = 48$, if only 48 hardware warps slots are filled (i.e., $ocp = 0.75$). Therefore, the tuning space may be reduced by skipping these redundant tuning points.

6.4.4 Finding the Optimal Approximate Degree

In this subsection, we attempt to find the optimal approximate degree concerning the user-defined QoS. Assume the execution time function with respect to approximate degree λ is $T(\lambda)$ (e.g., the black curves in Figure 6.3) while the error function is $E(\lambda)$ (e.g., the red curves in Figure 6.3). Then the searching problem can be formalized as:

$$\min(T(\lambda)) \mid E(\lambda) \leq QoS$$

This problem is difficult to solve if $T(\lambda)$ and $E(\lambda)$ are general functions. However, as $T(\lambda)$ is negatively correlated to $E(\lambda)$ and from Figure 6.3 we observe that $T(\lambda)$ is monotonically decreasing with λ, the problem thus can be reformulated as

$$\max(\lambda) \mid E(\lambda) \leq QoS$$

or simply finding the root of equation $E(\lambda) = QoS$ provided that $E(\lambda)$ is continuous. However, as λ here is discrete, it is essentially the last point before the root of $E(\lambda) = QoS$.

A naive approach to find the optimal λ is to start searching from the pure-SFU version with $\lambda = 64$ or 48, and evaluate all the points along the reduction of λ until $E(\lambda) \leq QoS$. This simple approach is labeled as **SMP**. To accelerate the searching process, based on the nearly-linear observations about $E(\lambda)$, we further propose a linear-approaching method motivated from *Newton's Method*. We

[4] We have observed an exception here for `SIN` on Fermi, in which the optimal performance point locates in the middle. This explains why later in Figure 6.7, `SIN`'s SFU bar is lower.

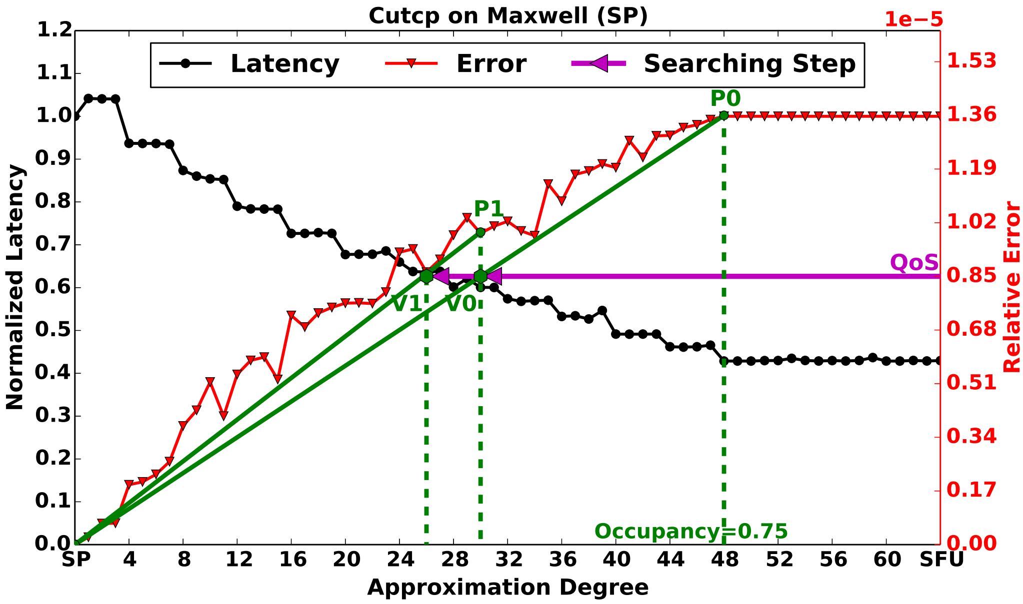

Figure 6.5: The proposed linear-approaching method (HEU) to locate the optimal λ for *cutcp* on a Maxwell GPU. The searching process terminates after two steps when QoS is satisfied.

use the *cutcp* application as an example. As illustrated in Figure 6.5, assume the QoS of this case is $0.85E-5$. To start, we first run the transformed kernel with $\lambda = 0$, which corresponds to the pure SPU/DPU version and dump the results. The performance $T(\lambda = 0)$ can also be measured if we want to calculate the speedup later. Next, we execute the kernel with $\lambda = 64$ (48 for Fermi) which corresponds to the SFU version. Similarly, we measure $T(\lambda = 64/48)$ and dump the results. Additionally, we measure the occupancy of the SFU version to reduce the search space (discussed in Section 6.5.3). For *cutcp*, the occupancy of the SFU version is 0.75, which indicates that the searching space is from 0 to 48. Then, by calculating the relative-error of the SFU version, we locate the position of $P0$ in Figure 6.5. Based on the nearly-linear observation about $E(\lambda)$, we draw a line from $P0$ to the origin and intersects it with the QoS level (the magenta line). The intersection is denoted as $V1$, where $\lambda = 30$. We run the kernel again with $\lambda = 30$ and calculate the relative-error $E(30)$, which locates $P1$. If $P1$ is less than QoS, it is the new lower-bound and we move the origin to $P1$; if $P1$ equals to the QoS, we return $P1$; if $P1$ is larger than QoS, it is the new upper-bound and we set $P1$ as the updated terminal point, as shown in Figure 6.5. We then connect $P1$ to the origin to form a new straight line, which intersects QoS at $V2$ where $\lambda = 26$. We run the kernel again with $\lambda = 26$ and find that $E(26)$ at V2 happens to be the same as the QoS. Therefore, the search process terminates and returns $\lambda = 26$. Otherwise, it repeats such a process until $E(\lambda)$ is finally equal to QoS. We label this heuristic method as **HEU**. Note that this linear-approaching method converges only when $E(\lambda)$ is roughly smooth. However, this is not always the case (e.g., `NBO, CFD, BIF` in Figure 6.3). In these scenarios, **HEU** may get trapped in a local optimal value. Therefore, in order to ensure $E(\lambda^*) < QoS$, when it is not satisfied, we add an extra phase to assess the points along the reduction of λ from the local optimal, all the way untill $E(\lambda^*) < QoS$.

Compared to the naive *SMP* approach and the exhaustive search that traverses the entire λ searching

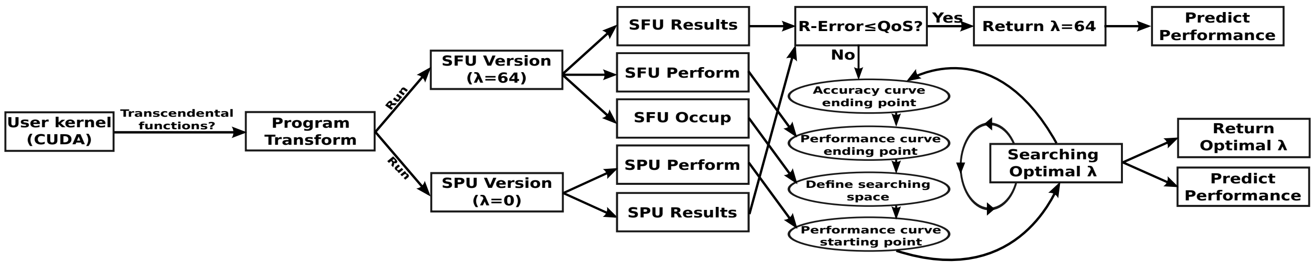

Figure 6.6: SFU-Driven Transparent Approximate Acceleration Framework.

space (labeled as **OMG**), our proposed *HEU* method can be much more efficient (will be validated in Section 6.7). The *HEU* method is also integrated into our SFU-driven approximation framework, which will be discussed next.

6.5 The Overall Framework

In this section, we describe the overall framework for our SFU-driven approximation acceleration design. As shown in Figure 6.6, when the application kernel is given, the framework first checks if it invokes any transcendental functions (SP or DP), especially the ones within a loop or nested loops. If so, it performs the program transformation discussed in Section 6.5.2. Such a transformation can be fully automatic as the mapping between the embedded PTX and the corresponding transcendental functions are fixed. Then the framework will perform the heuristic method discussed in Section 6.5.4 to find the optimal λ for achieving the best performance under a certain QoS. The only difference is that if the relative-error of the SFU version is less than QoS (e.g., `OCN, BKP, SRD, NPP` and `IMD`), it is returned immediately. Note that the "SFU/SPU result" indicated in Figure 6.6 is for the entire application instead of a single kernel. During the search, one can also profile the number of SPU/DPU/SFU operations performed in each step, and then combine the power/energy information in Figure 6.1 and 6.2 to calculate the power/energy consumption.

Our design is highlighted for its *transparency, tractability* and *portability*. It is **transparent** because it is a pure-software design that converts the code at compile time and runtime, so that it requires no extra efforts from both application developers and hardware designers. It also brings significant, instant and cheap speedup with guaranteed accuracy. Meanwhile, it is **tractable** because it is simple to understand and can be fully automatic (i.e., integrated into the CUDA toolchain). In addition, the horizontal approach it adopts introduces the nearly-linear performance-accuracy trade-off curves with a relatively large, uniform and fine-grained tuning space. Finally, regarding **portability**, our design works for all the current generations of GPUs with SFUs equipped, and it does not rely on architecture-related properties except for the limitation of the hardware warp-slots (Table 6.3).

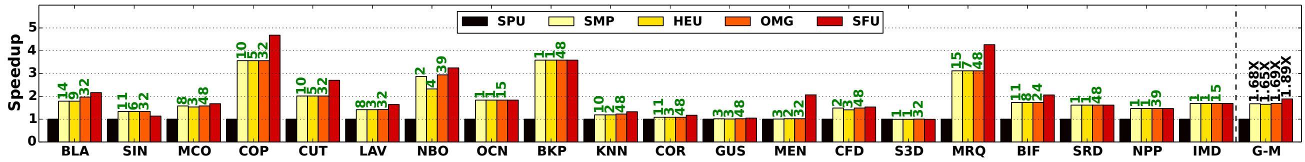

Figure 6.7: Speedup for QoS_ratio=0.8 on Fermi GPU in SP.

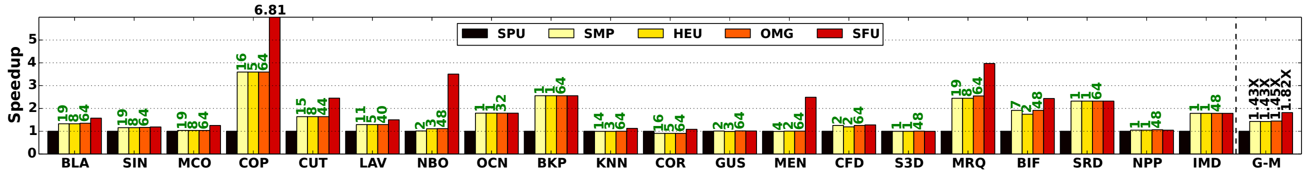

Figure 6.8: Speedup for QoS_ratio=0.8 on Kepler GPU in SP.

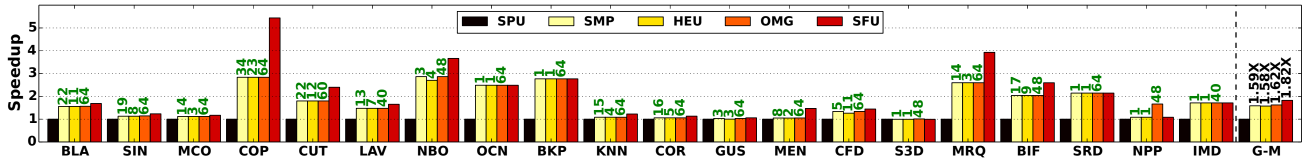

Figure 6.9: Speedup for QoS_ratio=0.8 on Maxwell GPU in SP.

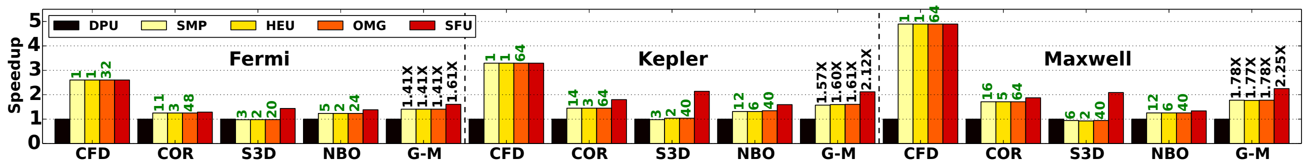

Figure 6.10: Speedup for QoS_ratio=0.8 on Fermi, Kepler and Maxwell GPUs in DP.

6.6 Validation

In this section, we validate our SFU-driven approximate acceleration design in the overall framework. We test 20 SP and 4 DP applications shown in Table 6.6 on the Fermi, Kepler and Maxwell platforms (*Platform 1,2,3* in Table 6.3). To be convenient, here we define **QoS_ratio** as the ratio of QoS with respect to the error-rate of the SFU version, which is supposed to be the highest based on the observations in Section 6.5.3.

Note that QoS_ratio is not QoS. For example, if the QoS of the pure SFU version regarding an application is 0.7, which means the error-rate of the SFU version is 1-0.7=0.3; then a QoS_ratio of 0.8 equals to a QoS of 1-0.3*0.8=0.76. We use QoS_ratio because the QoS values for the SFU-versions of different applications are distinct. The QoS_ratio offers a unified assessment criteria for comparison among applications. We also implement the naive (*SMP*), the heuristic (*HEU*) and the exhaustive search (*OMG*) methods described in Section 6.5.4 for searching efficiency comparison. Figure 6.7, 6.8 and 6.9 illustrate the results for applying our framework to locate the optimal approximate degree of the 20 SP applications on the three GPU platforms with the *QoS_ratio*[5]*=0.8*, respectively.

[5]We choose QoS=0.8 as an example for demonstration purposes. Users should determine the proper QoS metric and

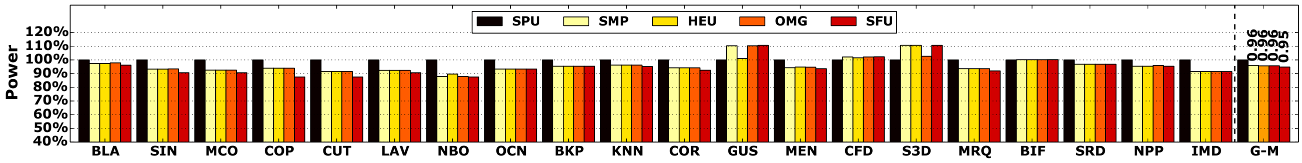

Figure 6.11: Normalized power reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in SP.

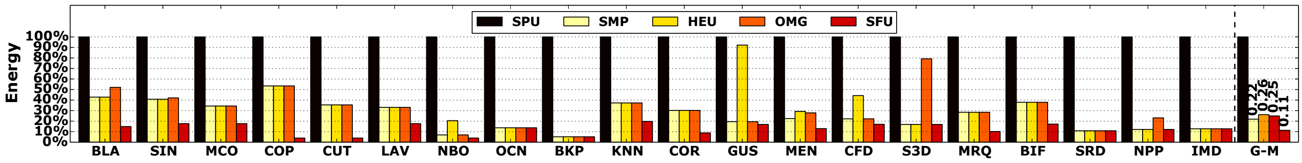

Figure 6.12: Normalized energy reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in SP.

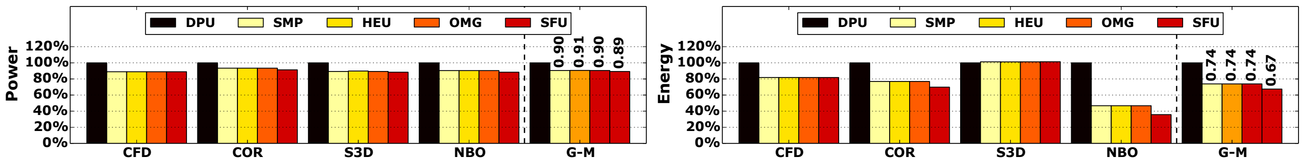

Figure 6.13: Normalized power and energy reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in DP.

Figure 6.10 shows the results for the 4 DP applications. In these four figures, **SPU/DPU** is the baseline with no approximation. **SFU** is the maximum attainable speedup via the proposed approach when all the transcendental functions are calculated by the SFUs. The green numbers marked on top of the bars indicate the total search rounds or steps, as described in Section 6.5.4. Such numbers indicate the numbers of executions during the search, or the searching overhead. We also show the geometric-mean of the performance speedup across the 20 SP and 4 DP applications to provide a general sense of acceleration under our framework. These figures demonstrate that given a specified QoS, *HEU* can achieve close to the best attainable performance with smaller searching iterations, compared to *SMP* and *OMG*.

Figure 6.11, 6.12 and 6.13 illustrate that the normalized power and energy reduction for SP and DP on the Maxwell Jetson-TX1 GPU (*Platform 5* in Table 6.3) for calculating the transcendental functions in the 20 SP and 4 DP applications via the proposed methods (*SMP, HEU* and *OMG*, which is the most optimal can be achieved at that QoS level) under the QoS_ratio=0.8. As can be seen, although the power reduction does not seem to be tremendous (around 5% for SP and 10% for DP), the energy reduction is quite significant – more than 75% and 25% for SP and DP respectively, which implies that our approximate method can also be quite effective for addressing power/energy constraining problems on GPUs.

level for their individual application.

6.7 Related Work

Approximate computing, which broadly refers to techniques that harvests substantial performance/energy benefits at the expense of modest accuracy loss, has prevailing at all levels of hardware and software designs. On one hand, the emerging big-data, multimedia and machine learning applications are much more insensitive to the computation accuracy. On the other hand, the low level hardware design faces ever-growing concerns on energy, resilience and sustainable scaling of performance. The majority of the existing research has been related to some traditional topics at both hardware level (e.g., fault-allowable storage [180], voltage overscaling [181], DRAM refresh [182], analog circuits [183], neural acceleration [184], descent fault recovery [185], remote memory data prediction [186], function memorization [163, 164], control/memory divergence [78]) and software level (e.g., loop perforation [176], task skipping [187], loop early termination [81, 188], program transformation [174], compilation [175], bitwidth reduction [182]). However, it is often not suitable to deploy the current approximate techniques directly to the scientific applications (e.g., weather simulation and molecular dynamics), which are usually numerically intensive and very sensitive to accuracy loss. This is especially true when future large-scale scientific applications are executed on thousands of heterogeneous HPC nodes (e.g., CPUs+GPUs) and a small inaccurate intermediate result can accumulate or propagate quickly to become significant [167, 168].

Recently, trading the accuracy of the results for better performance has been studied on GPUs [81, 163, 78, 164, 165, 166], as they become the essential computation units in both data centers and HPC systems. Samadi et al. [81] proposed three optimization techniques to automatically generate a series of GPU kernels with different aggressiveness of approximations. They also adopt an iterative sampling-calibration runtime tuning system to select the kernel in the series that is the most aggressive but complying to the specified QoS, provided that the same kernel is invoked repeatedly. Later, they found that for data-parallel applications, six commonly-used algorithm patterns could be approximated based on their specific properties [163]. Arnau et al. [164] proposed a look-up-table based task-level memorization approach to remove the redundant fragment computation when processing graphical applications in low-power mobile GPUs. Sartori and Kumar [78] applied the approximate concept to address the control and memory divergence on GPUs. They claimed that, for some error-tolerated applications, if the lockstep execution and memory coalescing are strictly enforced by approximating divergent paths to regular/coalesced paths, significant performance can be achieved with limited output quality degradation. Yazdanbakhsh et al. [165] focused on the long memory latency and limited memory bandwidth of GPUs, and predict the requested memory value without actually fetching it from the off-chip memory. Finally, Sutherland et al. [166] predicted the requested memory values using the GPU texture fetch units based on a thread's local history. However, the work above primarily exploits the spatial and/or temporal locality — the similarity among memory elements, computation lanes, historical memory loads, etc. They use hardware (e.g., look-up-table) or software (e.g., program transformation) approaches to approximate some of the requested data or computation with the predicted value based on locality. They often cannot provide accuracy assurance as locality is not always held, and if the crucial elements are approximated significantly inaccurate, catastrophic failures may occur. That is why most of the work above focused on applications that inherently have high tolerance for errors (e.g., machine learning or image applications), e.g., $\geq 10\%$ inaccuracy for approximation. Furthermore, the exact trade-off trends between the performance and accuracy are mostly nonlinear, sometimes even unknown beforehand. This is also why many of them require a profiling phase to test the kernel versions or train the look-up table. In addition, the performance-accuracy tuning space is relatively small and coarse-grained for most of the work above. In contrast, our SFU-centric approximation approach introduces nearly-linear performance-accuracy trade-off curves with a relatively large and fine-grained tuning space, for accuracy-sensitive scientific applications.

6.8 Limitations and Future Works

Limited by the situation that only 9 single-precision and 4 double-precision approximate numeric functions are implemented in the SFUs, the proposed design can only accelerate applications that contain these functions. Furthermore, limited by the fixed accuracy of the current SFU design (with errors less than 1E-6, see Table 6.4), we are unable to trade more accuracy with additional performance/energy gains. With regard to the future work, from hardware perspective, we can either design special-function accelerators that are faster but with higher error tolerance, or create accelerators that are more general-purpose such as the neural accelerator for GPUs [79]. From the software perspective, application developers can provide alternative approximate kernel implementations. For instance, in the *leukocyte* application from Rodinia benchmark [37], the *heaviside()* kernel has another "*simpler and faster*" approximate implementation which targets *actanf()*. Using a similar idea proposed in this work, we can co-schedule this user-defined approximate version with the accurate version without hardware involvement. [189] actually offers some software-based approximate functions, such as *sin, cos, exp* and *rcp*. Finally, it is also possible to apply the co-scheduling approach to approximate/accurate memory access of GPUs, such as guessing the data value when it is missed in the cache [165], or approximating a value based on the surrounding elements via interpolation in the texture cache [166].

6.9 Conclusion

In this chapter, we focused on a crucial GPU component which however, has long been ignored — the Special Function Units (SFUs), and show its outstanding role in performance acceleration and approximate computing for GPU applications. We exhaustively evaluated the 9 single-precision and 4 double-precision numeric transcendental functions that are accelerated by SFUs in terms of their latency, accuracy, power, energy, throughput, resource cost, etc. Based on these information, we proposed a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. It leverages the SIMT execution model of GPU to partition the initiated warps into a SPU/DPU-based slower but accurate path, and a SFU-based faster but approximated path, and then tune the relative partition ratio among the two to control the trade-offs between the performance and accuracy of the kernels. In this way, a fine-grained and almost linear tuning space for the trade-off between performance and accuracy can be created for a scientific application with approximate acceleration. With the linear tuning curve, we propose a simple yet effective heuristic method to search the optimal approximate degree that delivers the best performance subjecting to a user-predefined QoS level. The entire tuning process can be encapsulated as an automatic pure-software approximate-optimization framework, which is demonstrated to be effective for delivering immediate and substantial performance gains over a series of commodity GPU platforms.

CHAPTER 7

GPU Shared Memory Optimization: *Fine-Grained Synchronizations and Dataflow Programming*

The last decade has witnessed the blooming emergence of many-core platforms, especially the Graphic Processing Units (GPUs). With the exponential growth of cores in GPUs, utilizing them efficiently becomes a challenge. The data-parallel programming model assumes a single instruction stream for multiple concurrent threads (SIMT); therefore little support is offered to enforce thread ordering and fine-grained synchronization. This becomes an obstacle when migrating algorithms which exploit fine-grained parallelism, to GPUs, such as the dataflow algorithms. In this chapter, we propose a novel approach for **fine-grained inter-thread synchronization on the shared memory of modern GPUs**. We demonstrate its performance and compare it with other fine-grained and medium-grained synchronization approaches. Our method achieves 1.5x speedup over the warp-barrier based approach and 4.0x speedup over the atomic spin-lock based approach on average. To further explore the possibility of realizing fine-grained dataflow algorithms on GPUs, we apply the proposed synchronization scheme to *Needleman-Wunsch* — a 2D wavefront application involving massive cross-loop data dependencies. Our implementation achieves 3.56x speedup over the atomic spin-lock implementation and 1.15x speedup over the conventional data-parallel implementation for a basic sub-grid, which implies that the fine-grained, lock-based programming pattern could be an alternative choice for designing general-purpose GPU applications (GPGPU). This work has been presented at the 29th ACM International Conference on Supercomputing (ICS-15) [88].

7.1 Introduction

To harness the unprecedented computational capacity of modern multiprocessor architectures, a program must be partitioned and executed by multiple threads that communicate via shared memory or interconnection network. To ensure correctness, however, operations from various threads must obey certain order restrictions imposed by the program logic. Synchronization is the process referring to this coordination issue, during which information is exchanged among participant threads in certain order.

Synchronization can be further classified as *thread cooperation* and *thread contention* [190]. Thread cooperation enforces read-after-write data dependencies between cooperative threads, which is accomplished by producer-consumer primitives in general. Thread contention, on the other hand, ensures exclusive manipulation of the shared data so that program consistency is preserved. Atomic operations are provided for this purpose. The major difference between the two classifications is that thread cooperation emphasizes access order while thread contention stresses mutual exclusion. In this chapter, unless stated otherwise, the word *synchronization* is specially referred to thread cooperation.

Synchronization is not free. It can consume a significant fraction of the execution time due to parallelism degradation, as threads may stall at barriers or spin at locks [191, 192]. Furthermore, the synchronization process itself induces overhead, such as the communication delay and memory traffic for enquiring and releasing locks, the operation overhead for updating mutexes, the storage cost for synchronization variables, etc. Such overhead is particularly significant for algorithms that exploit *fine-grained parallelism* (e.g., many dataflow algorithms) as the occurrence of synchronization in these algorithms is much more frequent than in other applications [193]. As a result, numerous works have been proposed to alleviate the *fine-grained synchronization* overhead, from both architectural [194, 195, 196] and algorithmic perspectives [197, 198, 199].

Starting from the last decade, the graphics processing unit (GPU) has evolved to be applied on general purpose applications [44, 1]. However, traditional data-parallel programming models for GPUs assume a single instruction stream for all concurrent threads (SIMT) and little support is offered to enable elaborate thread cooperation. This becomes an obstacle when migrating dataflow applications which exploit fine-grained parallelism to GPUs.

GPU threads are organized in a hierarchy of three levels: *thread*, *warp* and *block*. Accordingly, three different granularities are addressed for GPU synchronization:

- *coarse-grained:* synchronization among thread blocks.
- *medium-grained:* synchronization among warps in thread blocks.
- *fine-grained:* synchronization among threads in thread blocks.

GPU currently provides hardware support for medium-grained warp barriers [53]. It also offers fine-grained atomic operations on global and shared memory [46]. However, the existing atomic operation based synchronization scheme, as will be seen, exhibits poor performance; using it incurs significant overhead. In this chapter, we propose a fine-grained, highly efficient thread synchronization mechanism on the shared memory of NVIDIA Fermi GPUs [46]. Instead of seeking to reduce the occurrence of synchronization, we look into an atomic instruction itself from a lower level point of view. By reassembling the *micro-instructions* that comprise an atomic operation, we develop an approach that can set up a *producer-consumer* communication channel between cooperative threads in a thread block with much less overhead than the atomic spin-lock based implementation. We validate the correctness and demonstrate the effectiveness of the proposed approach through comparisons with other fine-grained and medium-grained synchronization approaches. Further, to explore the possibility of realizing thread-level dataflow algorithms on GPUs, we apply the proposed synchronization scheme to *Needleman-Wunsch* – a 2D wavefront application that contains a large amount of cross-loop data dependencies. The performance we obtained proves that the fine-grained, lock-based programming pattern could be an alternative choice for designing GPGPU applications.

This chapter thus makes the following contributions:

- We show the inefficiency of the atomic spin-locks and propose a novel lock mechanism (called **tiny-lock**) that shows much better performance with no memory cost.
- We use the tiny-lock to build highly efficient producer-consumer primitives for fine-grained data synchronization between cooperative threads in a thread block.
- We address two architectural factors that can lead to deadlocks: one is the structural conflict between thread ordering and SIMD execution; the other is lock alias.
- We show how to realize lock-based dataflow computing on GPUs using a wavefront application. This is the first time, to the best of our knowledge, that a fine-grained dataflow model has been reported to be efficiently implemented at the lowest *thread level* of GPUs.

7.2 The Lock Unit on GPU Shared Memory

In this section, we briefly describe the architecture of the lock unit in GPU shared memory and the associated operations.

7.2.1 Shared Memory Lock Unit

The *shared memory* (i.e., *scratchpad memory*) in a GPU is a small on-chip storage shared among all processing units in a streaming multiprocessor (SM). It serves as a communication interface for fast data exchanging between different threads of a thread block. Being on-chip, the shared memory has much higher bandwidth and shorter access latency compared to the *global memory* (or *main memory*) of GPUs. Therefore, optimizations which can shift global memory access to shared memory access are highly advised by the CUDA programming guide [53].

The lock mechanism that enables fast atomic access is implemented in the shared memory, under the help of a module called "***lock unit***", in Fermi GPUs (see Figure 7.1). According to the associated patent [200], the lock bits are flags indicating the present lock status for the corresponding locations in the main storage (i.e., the Storage Resource in Figure 7.1). The lock bit is set so that other updating requests to that location are refused. For space concern, multiple locations in the main storage are *aliased* to a single lock bit. A hash function is implemented to perform the mapping, ensuring that successive *words* are mapped to distinct lock bits. For Fermi GPUs, a total of 1024 independent lock bits are provided for the 16KB (or 48KB, based on configuration) shared memory. Word addresses with a stride of 1024 are aliased to the same lock bit. When a memory request being delivered (to

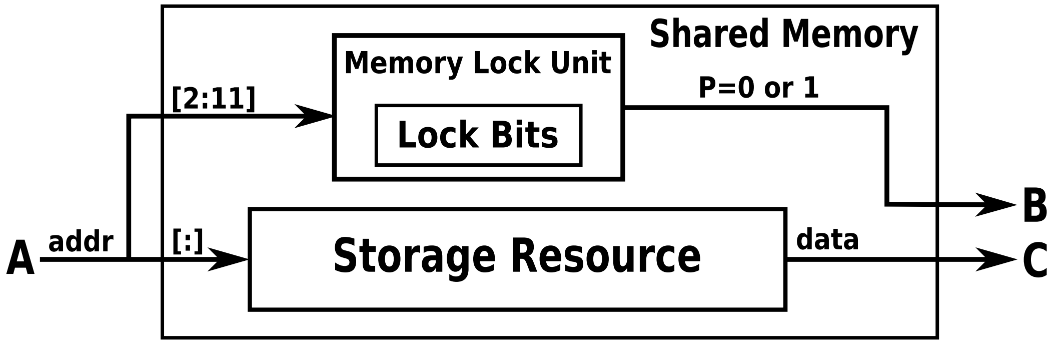

Figure 7.1: Shared memory lock unit. Terminal A reads the request memory address and looks it up in the storage resource. The fetched data is returned to terminal C that connects to a general register. Meanwhile, the 2-to-11 bits of the data address is used to retrieve the associated lock bit from the lock unit. The value of the lock bit is returned to terminal B, which connects to a predicate register.

terminal A in Figure 7.1), the 2-to-11 bits of the data address is labeled as the lock address and redirected to the lock unit. Gomez-Luna et al. discusses this mapping mechanism exhaustively in [201] and report a number of 1024 lock bits. We confirm this value experimentally when testing deadlocks (see Section 7.4). Regarding such a design, the following characteristics are highlighted for our proposal:

- **Efficiency:** Accessing the lock units does not require extra pipeline-stages or decision logic in the critical path because it is performed in parallel with the ordinary data access. So no extra delay is induced.
- **Flexibility:** The lock unit is not configured to track the ownership of the locks. It is the program's responsibility to honor the lock bits and to prevent illegal access to the locked locations in the main storage.

7.2.2 Shared Memory Atomic Operations

Listing 7.1 shows the low-level assembly sequence (SASS) generated by *cuobjdump* for the atomic instruction "*atomicAdd()*" to the shared memory of Fermi GPUs, in CUDA runtime (i.e., *atom.shared.add* instruction in PTX [155]). It indicates that the high-level *"atomic"* instruction is essentially comprised by a series of low-level SASS operations:

```
/*00a0*/  LDSLK P0, R1, [R0]; // try to lock
/*00a8*/  @P0 IADD R4, R1, 0x1; // if success, add 1
/*00b0*/  @P0 STSUL [R0], R4; // store and release lock
/*00b8*/  @!P0 BRA 0xa0; // if not success, retry
```

Listing 7.1: SASS code for atomicAdd()

- **LDSLK** loads data from address [R0] to a general register R1. It also reads the associated lock bit to a 1-bit predicate register P0. (In Figure 7.1, R0 is connected to A, R1 is connected to C, P0 is connected to B.) Therefore, P0 equals true implies that the target lock is successfully acquired by the current thread. Meanwhile, the lock bit in the lock unit toggles to 0, disabling subsequent locking requests. Here, "LDS" stands for loading from shared memory while "LK" means loading the lock bit simultaneously.

- Based on P0=1 (@P0), **IADD** adds 0x1 to R1 and stores the sum to R4. Note that threads in a warp may diverge here if some of them fail to acquire the locks in the present locking test (@!p0).
- Also with P0=1, **STSUL** stores R4 to [R0] and triggers the lock unit to reset the lock bit. "STS" stands for storing to shared memory while "UL" means unlocking simultaneously.
- **BRA** is the branch operation that jumps to instruction address 0xa0, which is the entry of the atomic procedure. In this way, the threads failed to obtain locks in the current test rotate back and redo the atomic process. Meanwhile, the finished threads have to wait beyond this BRA operation until all divergent threads in the warp have reached so as to continue lockstep execution.

Regarding these operations, it should be noted that:

- The default value of a lock bit is 1, indicating that it is free for fetching. LDSLK resets the lock bit to 0 while STSUL sets the lock bit to 1. It is infeasible to set the lock bit via LDSLK or reset the lock bit via STSUL. There is no alternative way to set or reset a lock bit.
- To release a lock bit, a thread **must** store a value to the corresponding memory location simultaneously. The store overwrites the original content.

7.3 Fine-Grained Synchronization

In this section, we present the fine-grained synchronization mechanism. We first describe our motivation and then propose the **tiny-lock**, based on which we show our fine-grained synchronization scheme.

7.3.1 Motivation

Our approach is motivated by the observation that an atomic instruction in the shared memory is comprised of multiple low-level SASS operations (Section 7.3.2). Therefore, we can *reassemble these SASS operations in a different way to build other more efficient synchronization procedures*.

7.3.2 Tiny-Lock

Fine-grained synchronization relies on fine-grained locks. Listing 7.2 illustrates a common implementation [147] of the fine-grained spin-locks based on atomic instructions.

```
__device__ inline void lock(int* p_mutex){
  while(atomicCAS(p_mutex,0,1)!=0);//compare and swap
}
__device__ inline void unlock(int* p_mutex){
  atomicExch(p_mutex,0);//exchange
}
```

Listing 7.2: Baseline implementation: atomic spin-locks [147]

In Listing 7.3, we show the SASS sequence of the baseline implementation of the lock/unlock primitives. To make it more clear, we draw the corresponding control-flow-graph (CFG) in Figure 7.2. There are two loops in the *Lock* routine: the small loop is spinning for a lock bit. It is embedded in *atomicCAS()*. The big loop, which corresponds to the *while* statement, is the actual iteration for the user-defined mutex variable stored in the main storage of the shared memory.

```
// ================================= Lock =================================
/*0060*/  SSY 0x98; //set convergence point
/*0068*/  LDSLK P0, R2, [R0];
/*0070*/  @P0 ISETP.EQ.U32.AND P1, pt, R2, RZ, pt;
/*0078*/  @P0 SEL R3, R2, 0x1, !P1;
/*0080*/  @P0 STSUL [R0], R3;
/*0088*/  @!P0 BRA 0x68; //atomicCAS loop
/*0090*/  ISETP.EQ.AND.S P2, pt, R2, RZ, pt;
/*0098*/  @!P2 BRA 0x60; //while loop
/*00a0*/  ... //converge to proceed lockstep execution
// ================================ Unlock ================================
/*00b0*/  LDSLK P0, RZ, [R0];
/*00b8*/  @P0 MOV32I R2, 0x1;
/*00c0*/  @P0 STSUL [R0], R2;
/*00c8*/  @!P0 BRA 0xb0;
```

Listing 7.3: SASS code of atomic spin-locks

This is a recursive design: the user-defined mutex acts as an intermediate layer to realize the required locking functionality (i.e., the big loop) whereas the lock bit of the mutex is leveraged to ensure atomic updates to the mutex (i.e., small loop). Such a design behaves quite well when the mutex serves as a semaphore, but is probably redundant when only a single-bit lock is required — *why not exploit the lock bit directly?*

We show the novel design in Listing 7.4. It is called **tiny-lock**. There are two primitives for locking: *Lock* simply fetches without verifying the locking result. It is used when the programmer guarantees the acquisition of the target locks (e.g., in an initialization scenario). Otherwise, *Waitlock* has to be applied, which repeatedly fetches the lock until it eventually succeeds. *Unlock* stores 1 to the lock bit for release.

```
// ================================= Lock =================================
/*0000*/  LDSLK P0, RZ, [R0];
// =============================== WaitLock ===============================
```

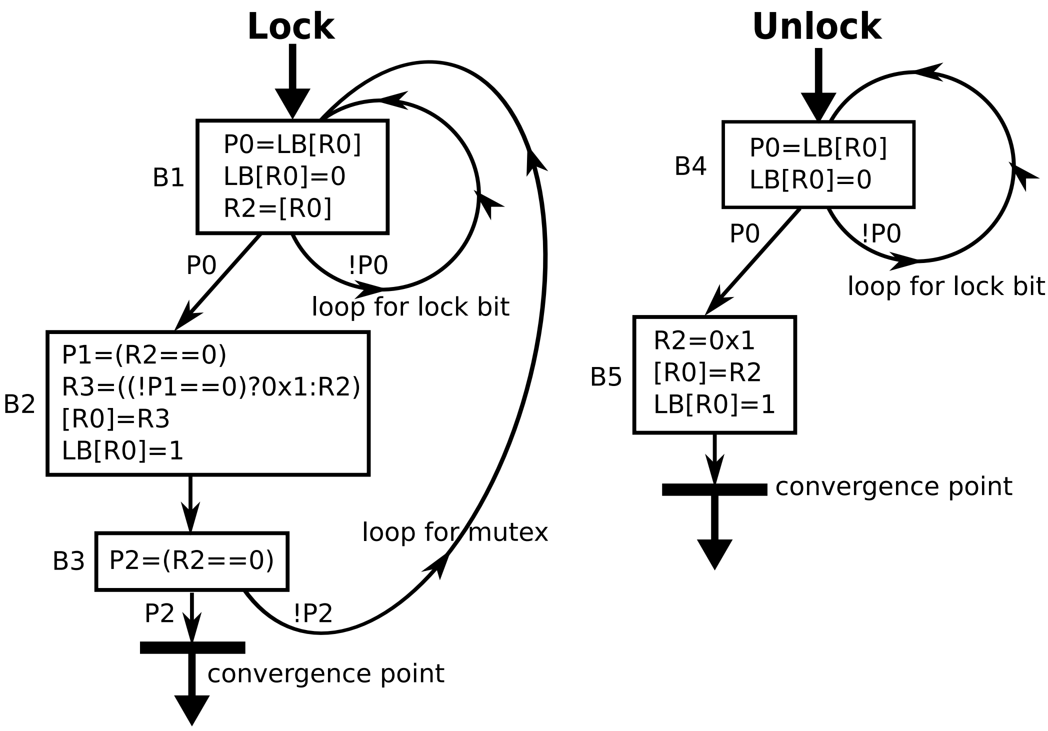

Figure 7.2: CFG of atomic spin-locks. LB stands for lock unit. Convergence point is the place where divergent threads of a warp rejoin to proceed lock-step execution. In the *Lock* routine, the big loop is for acquiring the user-defined mutex while the small loop is for acquiring the lock bit of the mutex. P2 being a replicate of P1 is the result of direct translation from two PTX instructions by the ptxas assembler. In the *Unlock* routine, the atomic update to the mutex (i.e., the small loop) is a must; otherwise, the updated result may be overwritten unexpectedly by another thread who acquires the lock bit but not the mutex. Since that thread needs to write a value to the mutex for releasing the lock bit, it uses a dated value obtained when fetching the lock bit as it is unaware of the latest update.

```
/*0010*/  LDSLK P0, RZ, [R0];
/*0018*/  @!P0 BRA 0x10;
// ================================ Unlock ================================
/*0020*/  STSUL [R0], RZ;
```

Listing 7.4: Proposed fine-grained lock

Such a design completely eliminates the space cost for the user-defined mutex. It also avoids the big loop in the *Lock* routine and the small loop in the *Unlock* routine. Compared to the baseline implementation, it has the following advantages:

- **Time Delay**: the proposed design reduces the static number of SASS operations by 75% for *Lock* and *Unlock*; and by 50% for one iteration of *Waitlock* (although the dynamic number of operations executed by waitlock depends on the waiting time experienced). Meanwhile, the lock unit is accessed in parallel with the shared storage (*Efficiency* in Section 7.1), so the maximum delay for accessing locks is equal to an ordinary memory read or write. Furthermore, this delay can be completely hidden in certain scenarios, e.g., the read-after-write data synchronization.
- **Storage Cost**: since the lock unit is isolated from the main storage, our scheme does not require any shared memory storage. In comparison, the baseline implementation has to explicitly

```
void producer(){
  lock(&mutex);//initialize
  ...
  shared_buffer=put;//store to channel
  unlock(&mutex);//signal consumer
}
```

```
void consumer(){
  ...
  lock(&mutex);//wait producer to store
  get=shared_buffer;//load from channel
  unlock(&mutex);//finalize
}
```

Listing 7.5: Fine-grained synchronization based on atomic spin-locks

```
//============= producer =============
/*0000*/ LDSLK P0,RZ,[R0];//initialize
...
//store to channel and unlock
/*0010*/ STSUL [R0],R4;
```

```
//============= consumer =============
//wait and load from channel
/*0100*/ LDSLK P0,R2,[R0];
/*0108*/ @!P0 BRA 0x100;//spinning
/*0200*/ STSUL [R0],R2;//finalize
```

Listing 7.6: Fine-grained synchronization based on atomic spin-locks

allocate a word as an intermediate mutex. Furthermore, since only the lock bit is of interest, in many cases (see Section 7.5 and Section 7.6) we can read the content of the memory location to the zero register in *Lock* or write the original value back in *Unlock* so that no register is used either.

- **Memory Traffic**: there is only one load transaction for *Lock* and one store transaction for *Unlock*. For *Waitlock*, unlike the baseline implementation that writes the original value back to the mutex if the lock is not obtained (B2 in Figure 7.2 when R2!=0), our approach does not produce any write traffic when locking. Furthermore, it does not produce computation traffic like the baseline implementation (e.g., operations 0x0070, 0x0078 and 0x00b8 in Listing 7.3).

7.3.3 Fine-Grained Synchronization

All concurrent programming models offer programmers the ability to control the order of dataflow from different threads. However, conventional SIMT programming model assumes weak inter-dependencies among threads that relies on barriers to enforce thread ordering. However, barriers are either coarse-grained or medium-grained in GPUs, which are too coarse for thread-to-thread synchronization. Therefore, fine-grained locks have to be used for such synchronization.

Listing 7.5 illustrates how an atomic spin-lock is used for read-after-write synchronization – *the producer thread acquires the mutex in advance and releases it after writing to the shared buffer so that when the consumer thread obtains the mutex, it can read safely.*

Here, a 1-bit lock is already sufficient to accomplish the job. However, as discussed earlier and will be seen in the experiments, the atomic spin-lock incurs significant time/space/traffic overhead which makes it too costly for frequent inter-thread synchronization. The proposed tiny-lock design significantly reduces such overheads and is therefore the ideal option upon which to construct the fine-grained synchronization scheme. Its implementation is shown in Listing 7.6.

This is the one-to-one synchronization scheme, which can be extended further to one-to-many and many-to-one conditions: the producer alternatively signals all its consumers or the consumer waits for all its producers.

7.3.4 Deadlock

Programmers must be careful when using fine-grained locks in GPUs because it is easy to generate deadlocks. Besides general causes from algorithmic aspects, there are two special scenarios that may lead to deadlocks for GPUs. We label them *SIMD Deadlock* and *Alias Deadlock*.

SIMD Deadlock

This kind of deadlock is due to a structural conflict between inter-thread synchronization and SIMD-lockstep execution. Consider the following scenario: what if the producer and consumer threads are from the same warp? The answer is — a *deadlock*. The general explanation is that lockstep stresses synchronous execution whereas thread cooperation enforces consumer-after-producer (i.e., read-after-write) order, which is essentially asynchronous. Therefore, if the synchronizing threads are from the same warp, we need a divergence mechanism to separate the producer and the consumer's execution paths. In addition, for the producer, the lock and unlock operations must be within the same divergent segment, or in other words, the unlock operation must be the post-dominator for the lock operation before the next convergence point. Otherwise, the producer will wait at that convergence point for the consumer to join in order to proceed to execute the unlock instruction, whereas the consumer is waiting to acquire the lock before it can step to the convergence point. Here the inter-waiting produces a deadlock.

In fact, such deadlocks occur more often than just for synchronization. Consider a warp executing the lock function in Listing 7.2. The convergence point is well beyond the while loop (see the black barrier in Figure 7.2). If two or more threads in the warp are contending for the same mutex (not lock bit), due to atomicity, only one of them can acquire it. However, this thread has to be blocked at the convergence point, waiting for other threads to join. Meanwhile, the remaining threads are adversely waiting for that thread to release the mutex (via calling the unlock function) before they can proceed. Here, the same reason leads to a deadlock: *the SIMD convergence point is earlier than unlock*. To circumvent this problem, a direct implementation for the baseline scheme is shown in Listing 7.7. In this way, the release of the mutex (i.e., *atomicExch(p_mutex,0)*) can be performed before the warp convergence point, which is right after the while loop.

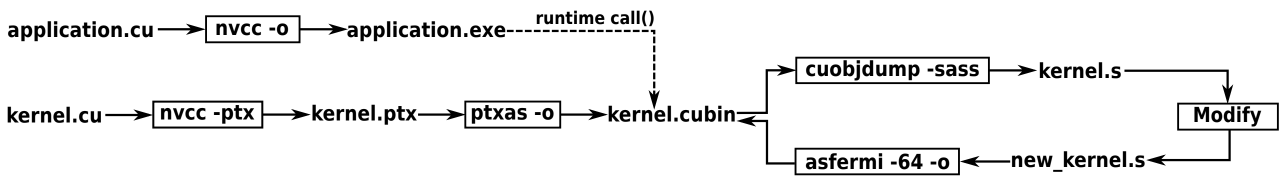

Figure 7.3: Experiment workflow. The application is written in CUDA driver-API that can load cubin object file at runtime. The kernel is first developed in CUDA-C and compiled to PTX code via *NVCC*. The PTX file is then assembled to a cubin binary via *ptxas* which is marked as the base binary. After that, the human-readable SASS routine is dumped from the base binary through *cuobjdump*. We modify this routine manually to insert the producer-consumer instructions, which is re-assembled to an updated cubin file for the driver-API to load.

```
__device__ void producer_consumer(int* p_mutex){
  bool finished = false;
  while(!finished){
    if(atomicCAS(p_mutex,0,1)==0){
      finished=true;
      ... // critical section
      atomicExch(p_mutex,0);
} } }
```

Listing 7.7: Intra-warp synchronization based on atomic spin-locks

And for our scheme in Listing 7.6, the predicate register can be manipulated to include the unlock operation into the same divergent path, as shown in Listing 7.8.

```
// producer-consumer
/*00a0*/   SSY 0x110;//set convergence point
/*00a8*/   LDSLK P0, R2, [R0];
...        @P0 ... //critical section
/*0100*/   @P0 STSUL [R0], R2;
/*0108*/   @!P0 BRA 0xa8;
```

Listing 7.8: Intra-warp synchronization based on lock bits

Although we successfully circumvent this deadlock at programming level, another problem still remains – performance degradation. As GPU adopts lane-masks to switch between divergent branches for a warp, the performance is impaired when each divergent branch has to be executed sequentially. Here, the producer lane has to wait until the consumer lane finishes. Even worse, if the consumer is in turn a producer of another synchronization also in the same warp, such as in a “scan” operation, then all the former producers have to be blocked until the final consumer finishes the synchronization. In the worst case, the performance drops by 32 folds (e.g., a propagation chain). Unless a perfect pipeline can be formed (i.e., producers start working on new data but execute in a lockstep with the consumers), some threads will be idle. The problem here is that the dispatch units only issue warp instructions, which is too coarse-grained for elaborate intra-warp coordination.

Summarizing, for synchronization between threads of different warps, we use the lock/unlock primitives in Listing 7.5 and 7.6. Both the producer and consumer can proceed immediately after the synchronization. But for synchronization involving threads from the same warp, the critical sections in Listing 7.7 and 7.8 are necessary. The producers have to wait until all their direct or indirect consumers accomplish their synchronization and arrive at the convergence point. Although performance suffers, the fine-grained scheme is still better than a medium-grained approach as consumers from other warps can be signaled as soon as the required data is produced, instead of waiting for the whole warp that contains the consumer to be finished.

Alias Deadlock

This kind of deadlock is due to lock bit aliasing. There are two conditions: First, suppose a thread already holds the lock bit of a memory location, say [M], but is trying to fetch from its aliased location (e.g., [M+1024], see Section 7.3.1). Then, the thread will trap in a circle because it is attempting to get a lock bit from itself. Based on our experiments, such a conduct immediately leads to a deadlock. However, the positive side is that such an experiment confirms the stride of lock bit alias is 1024 [201].

Second, we need to ensure the producer acquires the lock before its consumer (see Listing 7.5 and 7.6). As warps are not synchronously executed in an SM, this is achieved by placing a coarse-grained barrier (i.e., *__syncthreads()*) after the initialization phase for the whole thread block. Lock-bit aliasing generates deadlock because the warp obtaining the aliased lock waits at the block-wise barrier for other warps, including the failed warp, while the failed warp is waiting for the aliased lock before it can reach the barrier.

Although alias deadlock is easy to understand, it is one of the major restrictions for the proposed synchronization scheme: *to avoid alias deadlock, only 1024 locks can be utilized safely*. This number is smaller than the allocatable threads for an SM (i.e., 1536 threads) and much smaller than the entries of the shared memory (i.e., 4096 or 12,288). Given the fact that an SM can accommodate several thread blocks, the volume of usable lock bits can significantly limit the number of thread blocks an SM could support, hence degrading the performance for a large data size (see Section 7.6).

7.3.5 Warp-Shared Lock Bit

When fine-grained lock bits are exploited for the situations of medium-grained synchronization, it is possible to share a single lock bit for the whole warp, which reduces the demand for lock bits by a factor of 32. The idea is to exploit the warp-wise voting instructions [155]. Listing 7.9 provides the implementations for the lock and unlock routines, based on which the readers can further construct warp synchronization primitives.

```
// ================================ Lock ================================
//R0 is the same for all threads across the warp
/*0000*/  LDSLK P0, RZ, [R0];
// ============================== WaitLock ==============================
//If any thread acquires the lock bit, continue
/*0010*/  LDSLK P0, RZ, [R0];
/*0018*/  VOTE.ANY RZ, P1, P0;
/*0020*/  @!P1 BRA 0x10;
// =============================== Unlock ===============================
//Thread 0 in the warp releases the shared lock
/*0020*/  S2R R1, SR_LaneId; //Load lane_id
/*0028*/  ISETP.EQ P0, pt, R1, RZ, pt;//lane_id=0?
/*0030*/  @P0 STSUL [R0], RZ;
```

Listing 7.9: Warp-shared lock bit scheme

For *Lock*, any thread in the warp may acquire the lock eventually, but we know one of them must obtain it. For *Waitlock*, after acquiring, all threads are enforced to participate in a warp-wise vote. If any thread successfully acquires the target lock (i.e., P0=1), the voting result is true (i.e., P1=1). Then the whole warp quits the spinning loop and proceeds lockstep execution. Otherwise, the warp rotates back and tries again. For *Unlock*, it may be too expensive to let every thread perform the release operation since a 32-degree bank conflict and lock conflict can be generated [201]. Furthermore, if there are multiple threads waiting for the lock, releasing it 32 times (due to conflict) may potentially violate the consistency between the waiting threads. The method here is to find a representative. Here the ISETP instruction and predicate register P0 are used to select thread 0 for releasing. Note it is not feasible to let the representative thread acquire the lock for the whole warp because the remaining 31 threads may fail to make their writings observable by other warps due to the weekly-ordered memory model [53]. However, such a design is not a problem if an atomic-spin lock is shared for the whole warp, as it enforces the order in the memory.

7.4 Validation

In this section, we validate the correctness and demonstrate the effectiveness of our fine-grained synchronization scheme. We use a NVIDIA GTX-570 GPU as the test platform. It contains 15(SM)x32 CUDA cores with compute capacity 2.0 (Fermi). The CUDA toolkit version is 4.0. In terms of tools, *cuobjdump* is employed to generate the SASS code of the target kernel. We then modify the SASS code to insert our lock operations. However, to reproduce the *cubin* binary for the updated SASS code, an SASS assembler is necessary. Since *ptxas* only accepts PTX code, we use an open-source SASS assembly tool named *asfermi* [107] instead. This is also the reason why we restrict to Fermi – *asfermi does not support other architectures right now*. The detailed workflow is depicted in Figure 7.3.

```
for (i=0;i<32*N;i++) A[i+32]=A[i]+independent_computation(i);
```

Listing 7.10: Validation kernel (serial version)

The loop shown in Listing 7.10 is used for validation. It contains a parallel independent computation phase and a serial dependent reduction phase. It is derived from the kernel developed by Tullsen et. al. [195] that represents a common map-reduce pattern. In order to compare with the medium-grained synchronization approaches (see Section 7.2), we extend the dependency distance from 1 to the size of a warp (i.e., 32). Meanwhile, since only 16 warp barriers are available in a thread block (see Section 7.2), *N* is set to be 16. The whole loop is parallelized and mapped to 16 warps for concurrent execution. We compare the proposed tiny-lock implementation (i.e., *tiny_lock*, Section 7.4.3) with the atomic spin-lock implementation (i.e., *atom_lock*, Section 7.4.2), the medium-grained sync-arrive barrier implementation (i.e., *warp_barr*, Section 7.2), the shared lock-bit implementation (i.e., *warp_vote*, Section 7.4.5) as well as a shared spin-lock implementation (a warp shares a common spin-lock, i.e., *shrd_lock*). The core of the kernels for atomic spin-lock based, sync-arrive barrier based and tiny-lock based implementations are shown in Listings 7.11, 7.12 and 7.13 respectively.

```
__shared__ int A[32*N], mutex[32*N];
  lock(mutex[tid]);//producer initially locks
  __syncthreads();//ensure producer gets lock first
  /* ========================== Reduction Phase ==========================*/
  if (wid > 0) lock(mutex[tid-32]); //consumer waits
  A[tid]=A[tid-32]+independent_computation(tid);
  unlock(mutex[tid]);     //producer releases
  unlock(mutex[tid-32]); //finalize
```

Listing 7.11: Atomic spin-lock based version (CUDA code)

```
__shared__ int A[N*32];
  int tid = threadIdx.x; int wid = tid>>5; //log32=5
  /* ========================== Reduction Phase ==========================*/
  if (wid>0) asm("bar.sync␣%0,%1;"::"r"(wid-1),"r"(64));
  A[tid+32]=A[tid]+independent_computation(tid);
  asm("bar.arrive␣%0,%1;"::"r"(wid),"r"(64));
```

Listing 7.12: Warp barrier based version (PTX embedded CUDA code)

Scheme	Granularity	Performance	Memory Cost	Resource	Programmability
atom_lock	fine	x1.0	128 bytes/warp	4096/12,288 locations per SM	CUDA runtime
warp_barr	medium	x2.6	0	16 barriers per thread_block	PTX/embedded_PTX
shrd_lock	medium	x0.8	4 bytes/warp	4096/12,288 locations per SM	CUDA runtime
tiny_lock	fine	x4.0	0	1024 lock bits per SM	Assembly
warp_vote	medium	x2.0	4 bytes/warp	1024 lock bits per SM	Assembly

Table 7.2: Summary of synchronization schemes

```
/*0000*/ LDSLK P0,RZ,(A[tid]);//producer init locks
/*0008*/ BAR.RED.POPC RZ,RZ;  //block barrier
/* =========================== Reduction Phase ============================*/
/*0100*/ ISETP.EQ P0, pt, (wid), RZ, pt;
/*0108*/ @P0 BRA 0x120; // warp_0 breaks
/*0110*/ LDSLK P1,R1,(A[tid-32]); //consumer waits
/*0118*/ !@P1 BRA 0x110;
/*0120*/ IADD R2,R1,(independent_computation(tid));
/*0128*/ STSUL (A[tid]),R2; //producer releases
/*0130*/ @!P0 STSUL (A[tid-32]),R1; //finalize
```

Listing 7.13: Tiny-lock based version (SASS code)

To be simple, we set *independent_computation()* to immediately return its thread index. Therefore, if we measure the elapsed time for the reduction phase, it is the *raw delay for 16 times' synchronization and additions in sequence*. Figure 7.4 illustrates the measured execution time in cycles for the reduction phase for the 5 schemes. Table 7.1 lists the resource cost for each scheme.

As can be seen, our tiny-lock based approach is 4.0x times faster than the atomic spin-lock based scheme and is 1.5x times faster than the warp barrier scheme. Meanwhile, warp voting is shown to be an expensive operation (it actually induces thread divergence in a warp) although the sharing saves many lock bits. Finally, picking a warp-representative thread reduces space cost at the expense of performance loss. Table 7.2 summarizes the 5 schemes.

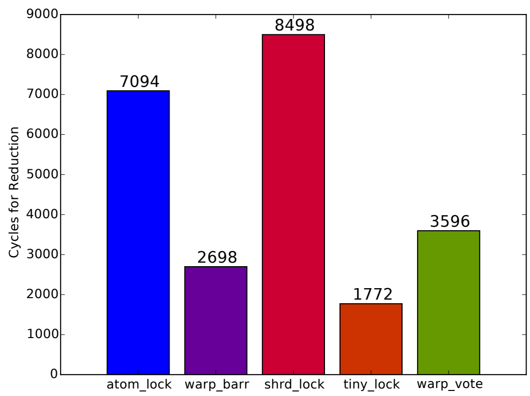

Figure 7.4: Execution time for the reduction phase in cycles

Scheme	Shared Memory Cost	Lock Bit Used
atom_lock	2048 bytes	512 (implicit)
warp_barr	0	0
shrd_lock	128 bytes	32 (implicit)
tiny_lock	0	512 (explicit)
warp_vote	0	32 (explicit)

Table 7.1: Resource Cost

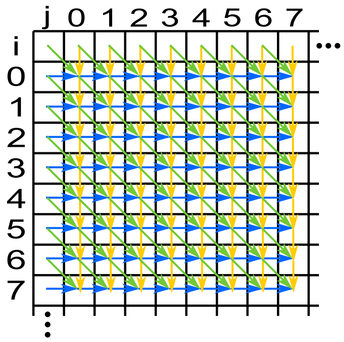

Figure 7.5: Dependence graph for the *Needleman-Wunsch* algorithm. The green arrows denote dependencies with the north-west neighbors. The yellow arrows refer to dependencies with north elements. The blue arrows indicate dependencies with the west grid-points. The first row and column of the grid are the initial values.

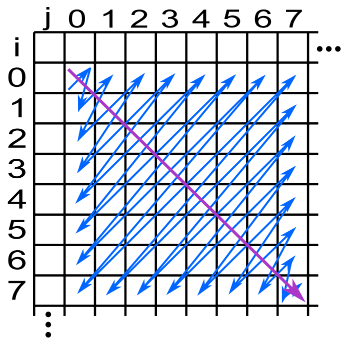

Figure 7.6: Working trace for wavefront parallel pattern. The wavefront direction coincides with the diagonal of the grid. In each wavefront step, the points along the anti-diagonal can be processed in parallel.

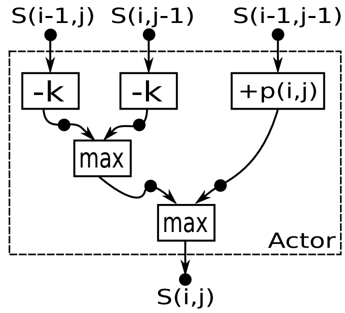

Figure 7.7: Dataflow graph. The actor computes Equation.7.1. When the required operands $S(i-1,j)$, $S(i,j-1)$ and $S(i-1,j-1)$ are ready, the actor can fire. The arcs across the dashed box denote the dependencies with other actors, which are also the places that require synchronization.

7.5 Wavefront Application

In this section, we use the *Needleman-Wunsch* algorithm [202, 203] from the *Rodinia* benchmark [37] as an example to describe how to efficiently implement a dataflow algorithm on GPUs using the proposed fine-grained, tiny-lock based synchronization schemes. The application is to find the best alignment between protein or nucleotide sequences in bioinformatics. Its core computation is:

$$S(i,j) = max \begin{cases} S(i,j-1)-k \\ S(i-1,j-1)+p(i,j) \\ S(i-1,j)-k \end{cases} \tag{7.1}$$

where S is 2D grid and $p(i,j)$ is a predefined reference field. As can be seen, the computation of each grid-point has true data dependencies on its north, west and north-west neighbors. The dependence graph is shown in Figure 7.5.

The data-parallel model relies on wavefront propagation to resolve such dependencies. In [204], Lamport et al. show that, for a multi-dimensional volume, given a value f, all points laid in the hyperplane satisfying $i+j+... = f$ can be processed in parallel while all their dependent points fulfill $i+j+... = f-1$. By stepping along the incremental direction of f and processing all elements associated, data dependencies can be respected. So far, all the existing implementations of wavefront applications on GPUs adopt this data-parallel pattern [205, 206, 207]. Figure 7.6 illustrates the processing trace of this pattern for the *Needleman-Wunsch* algorithm.

However, the data-parallel propagation approach confronts two problems: first, as the points that can be processed in parallel are along the line that is perpendicular to the diagonal, the computation workload for each propagation step is quite unbalanced, especially for SIMD processing. Second,

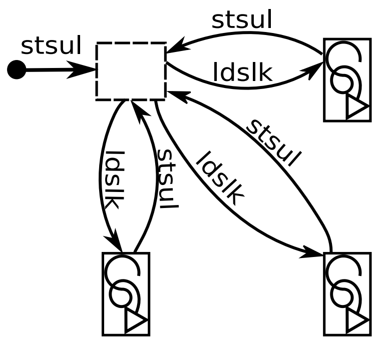

Figure 7.8: Using shared channel for synchronization.

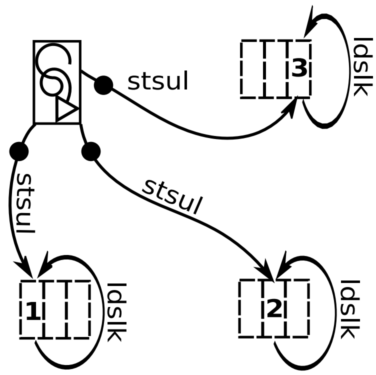

Figure 7.9: Using private channel for synchronization.

since the grid-points are normally sequentially stored along the axises of the gird in memory, data access in each step is cache unfriendly and cannot be coalesced for effective global memory fetch.

The major factor leading to the irregular computation and memory access is the rigorous 2D data-dependencies, which can be naturally and effectively resolved by a static dataflow model. A dataflow model describes the computation of each point as an *actor* which is executed by a GPU thread. The actor *fires* when all the operands it requires are available. Many actors may fire simultaneously, thus achieving high-level asynchronous concurrency. The dataflow graph for the application is shown in Figure 7.7. Since the computation of an actor is relatively simple, we concentrate on the communication part: *how to effectively synchronize between actors*.

There are two approaches: One is *resource-preferred*, which means a common synchronization channel is shared among the three consumers of a producer (Figure 7.8). Recall the synchronization process in Listing 7.5 and 7.6: the producer thread acquires the lock of the channel buffer first. Then, its three consumers (south, east and south-east neighbors) spin at the channel (it is also possible that they spin at other channels). When the producer fires, it releases a token to the channel. An arbitrary waiting consumer may acquire the token, but as other consumers may still wait for the token, it must restore the token back to the channel after usage. Since three consumers share one synchronization channel, a single lock is enough. However, due to the sharing of the token, a consumer may false-wait for other consumer(s) to restore the token before it can fire (In fact, it only has to wait for the producer, but there is no way for it to distinguish).

The other approach is *performance preferred*, meaning that each synchronization uses an isolated channel so that the consumers are independent of each other (Figure 7.9). So it is possible that the consumers can start firing earlier and they do not have to restore the token afterwards, which may benefit performance. The expense is three times the lock resource cost.

```
if (ty!=0 && tx!=0){
 lock(&mutex[ty][tx]); __syncthreads();
 while(!finished){
  if(!north_sync && atomicExch(&mutex[ty-1][tx],1)==0){
    north=s[ty-1][tx];//get north operand
    north_sync=true;
    atomicExch(&mutex[ty-1][tx],0);
  }
  if(!west_sync && atomicExch(&mutex[ty][tx-1],1)==0){
    west=s[ty][tx-1];//get west operand
    west_sync=true;
    atomicExch(&mutex[ty][tx-1],0);
  }
  finished=north_sync && west_sync;//ready?
  if(finished){//fire
    s[ty][tx]=MAX(s[ty-1][tx-1]+p[ty][tx],north-k,west-k);
    unlock(&mutex[ty][tx]);}//put self
  }
}
```

Listing 7.14: Atomic-based lock version

```
 /*00e8*/LDSLK P0,RZ,[R7];//lock self
 /*00f0*/BAR.RED.POPC RZ,RZ;
 /*00f8*/SSY 0x170;
 /*0100*/@!P1 LDSLK P1,R11,[R7+-0x4];//west
         //restore the token
 /*0108*/@P1 STSUL [R7+-0x4],R11;
 /*0110*/@!P3 LDSLK P3,R10,[R7+-0x80];//north
 /*0118*/@P3 STSUL [R7+-0x80],R10;
         //restore the token
 /*0120*/PSETP.AND.AND P2,pt,P3,P1,pt;//ready?
 /*0128*/@P2 LDS R12,[R7+-0x84];//fire
 /*0130*/@P2 ISETP.GE.AND P4,pt,R10,R11,pt;
 /*0138*/@P2 IADD R12,R12,R4;
 /*0140*/@P2 SEL R13,R10,R11,P4;
 /*0148*/@P2 IADD R13,R13,-R15;
 /*0150*/@P2 ISETP.GE.AND P5,pt,R13,R12,pt;
 /*0158*/@P2 SEL R8,R13,R12,P5;
 /*0160*/@P2 STSUL [R7],R8;//put self
 /*0168*/@!P2 BRA 0x100;
```

Listing 7.15: Fine-grained lock naive version

In our implementation, concerning the lock bits are limited and a shortage of locks may restrict the volume of actors, we adopt the resource-preferred approach. Meanwhile, for a point $S(i,j)$, it depends on $S(i-1,j-1)$. However, since $S(i-1,j)$ and $S(i,j-1)$ also depend on $S(i-1,j-1)$, if any token(s) from $S(i-1,j)$ or $S(i,j-1)$ is acquired, $S(i-1,j-1)$ can essentially be safely loaded. The core part of the implementations based on atomic spin-locks and tiny-locks are shown in Listing 7.14

Grid Size	Atomic-Lock	Data-Parallel	Tiny-Lock
31x31	175 μs	57 μs	49 μs
62x62	466 μs	58 μs	49 μs
124x124	1050 μs	58 μs	50 μs
248x248	2285 μs	59 μs	51 μs
496x496	5052 μs	72 μs	79 μs
992x992	14757 μs	79 μs	109 μs
1984x1984	48808 μs	80 μs	165 μs

Table 7.3: Execution time for atomic-lock, data-parallel and tiny-lock based Implementations.

and 7.15. To avoid intra-warp synchronization deadlocks (Section 7.4.4), the critical section scheme is used. Furthermore, the thread block configuration is set to be 32x32 to fully leverage the 1024 lock bits of an SM (also to avoid deadlocks due to alias, see Section 7.4.4).

We use the same outer framework as the original code and test the three implementations (data-parallel, atomic spin-lock dataflow, tiny-lock dataflow) on the GTX-570 platform. The execution time of the kernels are listed in Table 7.3. As can be seen, our tiny-lock based implementation is far more efficient than the atomic spin-lock approach, with as much as 296x speedup for the 1984x1984 data grid. Compared with the original data-parallel implementation, our tiny-lock method achieves more than 1.15x speedup on small size data grid (less than 248x248), but is slower for larger sizes. The scalability problem here is incurred by the restrictions on the number of threads and lock bits in an SM. In the data-parallel design, one warp is already sufficient to process a sub-grid, so one thread block contains only 32 threads. However, for the dataflow design, this number is 1024. Consequently, for a large grid size, more sub-grids can be processed simultaneously in the data-parallel approach, as an SM can sustain 8 thread blocks at a time for Fermi. For the dataflow approaches, however, an SM can only support one thread block (In fact, the maximum number of resident threads per SM is 1536 for Fermi, but there are only 1024 lock bits), which severely limits the exploitable parallelism at the thread block level. If the new generation GPUs integrate more lock bits and allow more threads for a SM, the data-flow scheme could achieve superior performance than the data-parallel scheme, even for large grid sizes.

7.6 Related Work about GPU Synchronizations

For **coarse-grained** synchronization on GPUs, Xiao et al. proposed three schemes [208]: a simple version, a tree-based version, and a lock-free version. The simple version leveraged a global-shared mutex via global memory atomic operations. The tree-based version improved the simple version by synchronizing progressively along the tree branches. The lock-free version allocated a monitor thread block to coordinate synchronization among working thread blocks. Their work was later extended by Stuart et al. to build a set of course-grained synchronization primitives [209].

In terms of **medium-grained** synchronization, although the block-wise barrier *__syncthreads()* is widely adopted, it was not until recently that a warp-to-warp synchronization approach has been developed. It relies on the *sync-arrive* barrier pair [155]: *bar.sync* is a blocking operation that suspends the current warp until all desired warps have arrived at the barrier. *bar.arrive* is a non-blocking operation that signals the arrival of the current warp to the barrier. In [210], Bauer et al. proposed a producer-consumer communication model based on this barrier-pair that could coordinate data movement from a producer warp to a consumer warp via shared memory buffers. They further applied this medium-grained synchronization approach to a chemical application [94]. The performance was demonstrated and the implementation was straightforward using the PTX embedding technique. However, for this approach, although the number of synchronization threads is parameterizable, it has to be a multiple of the warp size [155] (32 for all present CUDA GPUs), meaning that the granularity is warp, not thread. Furthermore, only 16 barrier instances are available per thread block [155], making these barriers very precious and limited for frequent usage, such as in a context of dataflow programming.

Regarding **fine-grained** synchronization, the only approach till now, to the best of our knowledge, is through the spin-locks, which are constructed using the atomic operations in global memory and shared memory. However, the performance of such atomic spin-locks is poor and their utilization is highly discouraged [211]. In fact, the lack of highly efficient, fine-grained synchronization mechanisms has already become an obstacle that disturbs the broad adoption of GPUs for general purpose applications [209, 205].

7.7 Limitations

Here we evaluate the limitations of the proposed synchronization scheme. First, in order to use it, one has to do low-level **SASS assembly programming**, which requires significant efforts. The coding process is error-prone and can easily lead to deadlocks, while debugging is almost impossible. However, this situation can be significantly improved if NVIDIA provides specific PTX instructions or CUDA functions to manipulate lock bits. This can also resolve the second limitation – **portability**. As no official SASS assembler is available, although the idea is general, our real hardware testing has to rely on the open-source *asfermi* that only functions smoothly for a portion of instructions for Fermi architecture. Since Kepler has dramatically improved the atomic functionality, we expect the proposed scheme can work more efficiently on the Kepler architecture. The third limitation is the **number of usable lock bits**, which restricts the parallelism and scalability that can be achieved on GPUs.

7.8 Conclusion

In this chapter we proposed a highly efficient lock mechanism on the shared memory of NVIDIA Fermi GPUs. By reassembling the SASS micro-operations that comprise an atomic instruction, we developed a highly efficient, low cost lock approach that can be leveraged to set up a fine-grained producer-consumer synchronization channel between cooperative threads in a thread block. This is the first time that the SASS instructions comprising an atomic operation are used independently to form new synchronization primitives. Furthermore, we showed how to implement a dataflow algorithm on GPUs using a real 2D-wavefront application. This is the first work that explores the possibility of applying lock-based dataflow-style programming model on GPUs.

Although programming with locks for the current platform/assembler is low-level and deadlock-prone, our work is already sufficient to show the possibility and potential of lock-based dataflow programming for GPUs. We expect more developers, especially architects and library writers to see such potential and participate in exploring and simplifying the programmability of this new design pattern.

CHAPTER 8

Conclusion and Future Work

The past decade has seen the miraculous boosting of many-core processors, especially the general-purpose GPUs. With the extraordinary growth of cores and threads in these highly-parallel platforms, well-understanding and effectively tuning the performance is becoming an ever-growing challenge, especially when concerning the sharing of various execution resources, such as the registers, caches, function-units, on-chip memories, etc, among thousands of cores and tens of thousands of threads in parallel. In this chapter, we summarize the thesis and propose possible extensions to motivate possible future work.

8.1 Conclusion

In the first part of this thesis, we first reviewed the development of GPGPU in Chapter 1, in particular its history, performance scaling and major research topics. Then in Chapter 2, we briefly discussed GPGPU itself, including its machine model, execution model, programming model and evaluation model. In Chapter 3, we proposed an analytic model called X to track the typical features of a parallel machine and its running workload, while visualizing their joint-effects (e.g., the entanglement of ILP, TLP, DLP and MLP) as the machine's spatial-state in an intuitive and tractable figure — the X-graph. With the X-graph, the X-model is able to comprehensively investigate the combined effects of various types of parallelism and the complex cache effects. Developers and architects can thus easily draw an X-graph to identify performance bottlenecks, discern potential optimizations and derive novel intuitions. We demonstrated the machine portability and workload portability of the X-model and showed its unique utilization in various optimizing scenarios (e.g., reducing ILP for cache thrashing). Later in Chapter 4 and Chapter 5, we leveraged the X-model to exploit the underlying tradeoffs for concurrency & registers and MLP & cache-performance.

In the second part of the thesis, we focused on each different on-chip module inside a GPU, in particular the register, the L1/L2/RO caches, the SPU/DPU/SFU, the scratchpad memory and proposed different optimization designs respectively. In Chapter 4, we proposed an autotuning approach to resolve the conflict between concurrency and register usage for GPUs. We discovered that the performance impact from register is continuous but from concurrency is discrete. The tradeoff between the two factors forms a special relationship such that a series of critical-points can be precomputed. These CPs denote the best performance of each concurrency level, and the global optimum is then selected among them. Our method reduces the search space for the optimal register usage by up to 20x and enhances the overall GPU performance by up to 1.5x. In Chapter 5, we proposed an adaptive cache bypassing framework for GPUs. It used a straightforward approach to throttle the number of warps that could access the three types of GPU caches – L1, L2 and read-only caches, thereby avoiding the fierce cache thrashing of GPUs. We validated the framework on seven GPU platforms that covered all GPU generations. Results showed that adaptive bypassing could bring significant speedup (on average 2.16x) over the general cache-all and bypass-all schemes. In Chapter 6, we focused on a crucial GPU component which however, has long been ignored — the Special Function Units (SFUs), and show its outstanding role in performance acceleration and approximate computing for GPU applications. We exhaustively evaluated the 9 single-precision and 4 double-precision numeric transcendental functions that are accelerated by SFUs in terms of their latency, accuracy, power, energy, throughput, resource cost, etc. Based on these information, we proposed a design framework for SFU-driven approximate acceleration on GPUs. It leveraged the SIMT execution model of GPU to partition the initiated warps into a SPU/DPU-based slower but accurate path, and a SFU-based faster but approximated path, and then tune the relative partition ratio among the two to control the trade-offs between the performance and accuracy of the kernels. Our design achieved 1.89x speedup with an accuracy loss of 0.15 for the results (QoS=0.8). In Chapter 7, we proposed a highly efficient lock mechanism on the shared memory of GPUs. By reassembling the SASS micro-operations that comprise an atomic instruction, we developed a highly efficient, low cost lock approach that can be leveraged to set up a fine-grained producer-consumer synchronization channel between cooperative threads in a thread block. Furthermore, we showed how to implement a dataflow algorithm on GPUs using a real 2D-wavefront application. This is the first work that explores the possibility of applying lock-based dataflow-style programming model on GPUs. Our method achieves 1.15x performance improvements over the baseline design from the benchmark. There are three common features for these design approaches:

- Transparent: all the designs are purely software-based. Therefore, they require no modifications or extensions to the underlying hardware or the user applications. They are immediately deployable and lead to very achievable performance benefits.
- Tractable: all the designs are intuitive to understand while straightforward to implement (probably excluding Chapter 7). Mostly they serve as a fully-automatic compile-time/runtime framework that can be integrated as part of the compiler/profiler toolchain.
- Portable: all the designs are validated on the three existing NVIDIA GPU generations: Fermi, Kepler and Maxwell (except Chapter 7 which is infeasible). They boost performance on all of them thus proving great inter-platform portability.

In this thesis, we strongly highlighted the most significant divergence for modern GPU architecture/software design, when compared with the traditional CPU family — the **spatial property of the massive SIMT execution model** (which is addressed by the X-model). It introduced a novel and fine-grained dimension (i.e., the thread dimension) into the design space, thus enabling the proposed horizontal design paradigm: *instead of tuning upon instructions/functions in the program context* (e.g., all the CPU-based optimization techniques), *we tuned the massive identical fine-grained threads among different compute/data-paths* (e.g.,partitioned warps among cache/bypass data-paths in Chapter 5, partitioned warps among SFU/SPU compute-paths in Chapter 6). We believe this is one of the most crucial and exciting research opportunities for GPUs.

8.2 Future Work

Of course, the contents in the thesis can always be extended. We have already discussed possible extensions for the works in each separate chapter. Specially for the X-model, it can be extended to address other important features of GPU, such as the memory access coalescing, the shared-memory bank-conflicts, the atomic operations, etc. Besides the performance model, a power model can be integrated into the X-model to indicate the power variation when a parameter is changed. Furthermore, the mathematic basis of the model can be improved while the model itself can be validated on other multithreaded platforms, such as Intel Xeon-Phi, AMD GPUs, the normal multicore processors and possibly grid or cloud.

References

[1] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), 2008.

[2] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[3] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[4] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[5] Qianqian Fang and David A Boas. Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Optics express*, 17(22):20178–20190, 2009.

[6] Jacobus Antoon Van Meel, Axel Arnold, Daan Frenkel, SF Portegies Zwart, and Robert G Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[7] Peter J Lu, Hidekazu Oki, Catherine A Frey, Gregory E Chamitoff, Leroy Chiao, Edward M Fincke, C Michael Foale, Sandra H Magnus, William S McArthur Jr, Daniel M Tani, et al. Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 5(3):179–193, 2010.

[8] Nachiket Kapre and André DeHon. Performance comparison of single-precision spice model-evaluation on fpga, gpu, cell, and multi-core processors. In *2009 International Conference on Field Programmable Logic and Applications*, pages 65–72. IEEE, 2009.

[9] Wenjing Gao, Nguyen Thi Thanh Huyen, Ho Sy Loi, and Qian Kemao. Real-time 2d parallel windowed fourier transform for fringe pattern analysis using graphics processing unit. *Optics express*, 17(25):23147–23152, 2009.

[10] CUDA NVIDIA. CUDA Programming Guide. 2007.

[11] Mark JP Wolf. *Before the crash: Early video game history*. Wayne State University Press, 2012.

[12] Mason Woo, Jackie Neider, Tom Davis, et al. Opengl programming guide. 1997.

[13] Kris Gray. *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[14] NVIDIA Corporation. Geforce 256.

[15] Randi J Rost, Bill Licea-Kane, Dan Ginsburg, John M Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language*. Pearson Education, 2009.

[16] Craig Peeper and Jason L Mitchell. Introduction to the directx® 9 high level shading language. *ShaderX2: Introduction and Tutorials with DirectX*, 9, 2003.

[17] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[18] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[19] Matt Pharr and Randima Fernando. *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[20] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. *Language version*, 1, 2004.

[21] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[22] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[23] Daniel Horn. Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation, chapter stream reduction operations for gpgpu applications, 2005.

[24] M Houston and N Govindaraju. Gpgpu: general-purpose computation on graphics hardware. *Course at SIGGRAPH*, 2007.

[25] Dominik Göddeke. *Gpgpu-basic math tutorial*. Univ. Dortmund, Fachbereich Mathematik, 2005.

[26] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[27] Intel. Developer Guide for Intel SDK for OpenCL, 2016.

[28] NVIDIA. Opencl programming guide for the cuda architecture, 2009.

[29] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC*. IEEE, 2008.

[30] Ang Li, Akash Kumar, Yajun Ha, and Henk Corporaal. Correlation ratio based volume image registration on gpus. *Microprocessors and Microsystems*, 39(8):998–1011, 2015.

[31] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[32] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[33] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[34] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(2):1, 2008.

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[36] Bowen Zhang and Cornelis W Oosterlee. Option pricing with cos method on graphics processing units. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[37] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[38] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[39] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 63–74. ACM, 2010.

[40] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[41] Molly A O'Neil and Martin Burtscher. Microarchitectural performance characterization of irregular gpu kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 130–139. IEEE, 2014.

[42] CUDA NVIDIA. SDK Code Samples, 2015.

[43] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software. IEEE International Symposium on*. IEEE, 2009.

[44] W Hwu Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[45] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, (2):39–55, 2008.

[46] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Fermi architecture white paper.

[47] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, (2):50–59, 2011.

[48] NVIDIA. NVIDIA GeForce GTX680 Whitepaper.

[49] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2015.

[50] NVIDIA. NVIDIA GeForce GTX750Ti Whitepaper, 2015.

[51] NVIDIA. NVIDIA GeForce GTX980 Whitepaper, 2015.

[52] NVIDIA. NVIDIA Tesla P100 Whitepaper, 2016.

[53] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[54] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 2010.

[55] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2010.

[56] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[57] B Dally. Gpu computing to exascale and beyond. *Lecture slides–http://www. nvidia. com/content/PDF/sc*, 2010.

[58] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. Low-power cmos digital design. *IEICE Transactions on Electronics*, 75(4):371–382, 1992.

[59] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3):13, 2011.

[60] Anantha P Chandrakasan, William J Bowhill, and Frank Fox. *Design of high-performance microprocessor circuits*. Wiley-IEEE press, 2000.

[61] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. Gated-v dd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 90–95. ACM, 2000.

[62] Yue Wang, Soumyaroop Roy, and Nagarajan Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 300–303. EDA Consortium, 2012.

[63] Daecheol You and Ki-Seok Chung. Dynamic voltage and frequency scaling framework for low-power embedded gpus. *Electronics letters*, 48(21):1333–1334, 2012.

[64] Rong Ge, Ryszard Vogt, Jahangir Majumder, Ahmad Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 826–833. IEEE, 2013.

[65] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[66] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.

[67] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 50, pages 265–266. ACM, 2015.

[68] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. Optimization of asynchronous graph processing on gpu with hybrid coloring model. In *ACM SIGPLAN Notices*, volume 50, pages 271–272. ACM, 2015.

[69] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[70] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[71] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[72] David J Palframan, Nam Sung Kim, and Mikko H Lipasti. Precision-aware soft error protection for gpus. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 49–59. IEEE, 2014.

[73] Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[74] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 287–300. IEEE, 2011.

[75] Sotiris Tselonis, Vasilis Dimitsas, and Dimitris Gizopoulos. The functional and performance tolerance of gpus to permanent faults in registers. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 236–239. IEEE, 2013.

[76] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104. ACM, 2009.

[77] Imran S Haque and Vijay S Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 691–696. IEEE, 2010.

[78] John Sartori and Ravindra Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia*, 15(2):279–290, 2013.

[79] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kerman, and Hadi Esmaeilzadeh. Neural Acceleration for GPU Throughput Processors. 2015.

[80] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 35–50. ACM, 2014.

[81] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24. ACM, 2013.

[82] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. Sfu-driven transparent approximation acceleration on gpus. In *Proceedings of the 30th ACM on International Conference on Supercomputing (ICS)*. ACM, 2016.

[83] Daniel Wong, Nam Sung Kim, and Murali Annavaram. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 176–187. IEEE, 2016.

[84] Ang Li, Y. C. Tay, Akash Kumar, and Henk Corporaal. Transit: A visual analytical model for multithreaded machines. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2015.

[85] Ang Li, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda, and Henk Corporaal. X: A comprehensive analytic model for parallel machines. In *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[86] Ang Li, Leon Shuaiwen Song, Akash Kumar, Eddy Z. Zhang, Daniel Chavarria, and Henk Corporaal. Critical Points Based Register-Concurrency Autotuning for GPUs. In *DATE*. IEEE, 2016.

[87] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 17. ACM, 2015.

[88] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-grained synchronizations and dataflow programming on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pages 109–118. ACM, 2015.

[89] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[90] John McCalpin. Stream benchmark, 1995.

[91] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[92] NVIDIA. CUDA Programming Guide, 2015.

[93] NVIDIA. CUDA Best Practice Guide, 2015.

[94] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: leveraging warp specialization for high performance on gpus. *ACM SIGPLAN Notices*, 49(8), 2014.

[95] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 408–419. IEEE, 2005.

[96] Inderjit Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 578–590. IEEE, 2013.

[97] Laxmi N Bhuyan, Qing Yang, and Dharma P Agrawal. Performance of multiprocessor interconnection networks. *Computer*, (2):25–37, 1989.

[98] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 407–420. IEEE Computer Society, 2007.

[99] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *MICRO*. IEEE Computer Society, 2012.

[100] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Divergence-aware warp scheduling. In *MICRO*. ACM, 2013.

[101] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO*. ACM, 2011.

[102] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. *ACM SIGARCH Computer Architecture News*, 41(1), 2013.

[103] Vasily Volkov. Better performance at lower occupancy. In *GTC*, 2010.

[104] Nicholas Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[105] NVIDIA. CUDA Compiler Driver NVCC, 2015.

[106] NVIDIA. CUDA Binary Utilities, 2015.

[107] Yunqing Hou. Asfermi: An assembler for the NVIDIA Fermi instruction set. `http://code.google.com/p/asfermi/`, 2011.

[108] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Barra: A parallel functional simulator for gpgpu. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351–360. IEEE, 2010.

[109] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[110] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 9. ACM, 2011.

[111] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A detailed GPU cache model based on reuse distance theory. In *HPCA*. IEEE, 2014.

[112] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. Miaow-an open source rtl implementation of a gpgpu. In *Low-Power and High-Speed Chips (COOL CHIPS XVIII), 2015 IEEE Symposium in*, pages 1–3. IEEE, 2015.

[113] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. Enabling gpgpu low-level hardware explorations with miaow: an open-source rtl implementation of a gpgpu. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):21, 2015.

[114] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[115] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[116] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[117] Tao Zhang, Guangshuo Chen, Wei Shu, and Min-You Wu. Microarchitectural characterization of irregular applications on gpgpus. *ACM SIGMETRICS Performance Evaluation Review*, 42(2):27–29, 2014.

[118] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[119] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12. IEEE, 2009.

[120] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in gpu applications. In *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[121] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60. IEEE, 2014.

[122] NVIDIA. Profiler User's Guide, 2015.

[123] NVIDIA. Kepler Tuning Guide, 2015.

[124] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[125] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for GPGPUs. In *PACT*. IEEE, 2013.

[126] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[127] Y. C. Tay. Analytical performance modeling for computer systems. *Synthesis Lectures on Computer Science*, 2013.

[128] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[129] Bruce L Jacob, Peter M Chen, Seth R Silverman, and Trevor N Mudge. An analytical model for designing memory hierarchies. *TC*, 45(10), 1996.

[130] NVIDIA. CUDA port of the stream benchmark.

[131] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[132] Jack Dongarra. Toward a new metric for ranking high performance computing systems. *Sandia Report*, (SAND2013-4744 312), 2013.

[133] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[134] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *MICRO*. IEEE, 2014.

[135] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *HPCA*. IEEE, 2014.

[136] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[137] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1), 2009.

[138] Zvika Guz, Oved Itzhak, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C Weiser. Threads vs. caches: modeling the behavior of parallel workloads. In *Proc. ICCD*. IEEE, 2010.

[139] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. ISCA*. ACM, 2009.

[140] Mike Murphy. NVIDIA's Experience with Open64. In *Open64 Workshop at CGO*, 2008.

[141] Ari B Hayes and Eddy Z Zhang. Unified on-chip memory allocation for SIMT architecture. In *ICS*. ACM, 2014.

[142] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *ISCA*. ACM, 2011.

[143] Wing-Kei S Yu, Ruirui Huang, Sarah Q Xu, Sung-En Wang, Edwin Kan, and G Edward Suh. SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading. In *ISCA*. ACM, 2011.

[144] Mark Gebhart, Stephen W Keckler, Brucek Khailany, Ronny Krashinsky, and William J Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *MICRO*. IEEE, 2012.

[145] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-compression: enabling power efficient GPUs through register compression. In *ISCA*. ACM, 2015.

[146] 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 *Computer graphics forum*, volume 26. Wiley Online Library, 2007.

[147] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[148] Peter N Glaskowsky. NVIDIA's Fermi: the first complete GPU computing architecture, 2009.

[149] John Nickolls and William J Dally. The GPU computing era. *IEEE Micro*, 30(2), 2010.

[150] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. In *ICCAD*. IEEE, 2013.

[151] Yao Zhang and John D Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*. IEEE, 2011.

[152] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Adaptive Cache and Concurrency Allocation on GPGPUs. 2013.

[153] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *MICRO*. IEEE, 2010.

[154] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for GPGPUs. *ACM SIGARCH Computer Architecture News*, 41(3), 2013.

[155] NVIDIA. PTX: Parallel Thread Execution ISA Version 4.0. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[156] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic gpu cache bypassing. In *ICS*. ACM, 2015.

[157] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *ICS*. ACM, 2012.

[158] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *PACT*. IEEE Press, 2013.

[159] Dong Li, Minsoo Rhu, Daniel R Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S Fussell, and Stephen W Redder. Priority-based cache allocation in throughput processors. In *HPCA*. IEEE, 2015.

[160] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498. ACM, 2013.

[161] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate Computing and the Quest for Computing Efficiency. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC'15, pages 120:1–120:6. ACM, 2015.

[162] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of Service Profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 25–34. ACM, 2010.

[163] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 35–50. ACM, 2014.

[164] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 529–540. IEEE, 2014.

[165] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Taesoo Kim, Onur Mutlu, and Todd C Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. In *Proceedings of the 11th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*. ACM, 2016.

[166] Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger. Texture Cache Approximation on GPUs. 2015.

[167] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'12, pages 78:1–78:12. IEEE Computer Society Press, 2012.

[168] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the Propagation of Transient Errors in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, pages 72:1–72:12. ACM, 2015.

[169] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/-software interface*. Newnes, 2013.

[170] Stuart F Oberman and Michael Y Siu. A high-performance area-efficient multifunction interpolator. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 272–279. IEEE, 2005.

[171] Davide De Caro, Nicola Petra, and Antonio GM Strollo. High-performance special function unit for programmable 3-D graphics processors. *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, 56(9):1968–1978, 2009.

[172] NVIDIA. CUDA Math API, 2015.

[173] NVIDIA. NVIDIA system management interface, 2015.

[174] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, volume 47, pages 441–454. ACM, 2012.

[175] Pooja Roy, Jianxing Wang, and Weng Fai Wong. PAC: program analysis for approximation-aware compilation. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 69–78. IEEE Press, 2015.

[176] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[177] NVIDIA. Inline PTX Assembly in CUDA, 2015.

[178] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174. IEEE, 2009.

[179] Ismail Akturk, Karen Khatamifard, and Ulya R Karpuzcu. On quantification of accuracy loss in approximate computing. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, page 15, 2015.

[180] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[181] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, 2012.

[182] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312. ACM, 2012.

[183] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose Code Acceleration with Limited-precision Analog Computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 505–516. IEEE Press, 2014.

[184] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460. IEEE Computer Society, 2012.

[185] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA'10, pages 497–508. ACM, 2010.

[186] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 127–139. IEEE Computer Society, 2014.

[187] Martin Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS'06, pages 324–334. ACM, 2006.

[188] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[189] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand. Sparse-coefficient polynomial approximations for hardware implementations. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 1, pages 532–535. IEEE, 2004.

[190] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

[191] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1), 1990.

[192] Thomas E Anderson, Edward D Lazowska, and Henry M Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Computers, IEEE Transactions on*, 38(12), 1989.

[193] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. *The impact of synchronization and granularity on parallel systems*, volume 18. ACM, 1990.

[194] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 35–45. ACM, 2007.

[195] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture. Proceedings. Fifth International Symposium On*, pages 54–58. IEEE, 1999.

[196] William E Cohen, Henry G Dietz, and JB Sponaugle. Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors. In *Parallel Processing. International Conference on*, volume 1. IEEE, 1994.

[197] Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for efficient placement of synchronization primitives. In *ACM Sigplan Notices*, volume 44, pages 199–208. ACM, 2009.

[198] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *Computers, IEEE Transactions on*, 36(12), 1987.

[199] Martin C Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4), 1999.

[200] Brett W Coon, Peter C Mills, John R Nickolls, and Lars Nyland. Lock mechanism to enable atomic updates to shared memory, November 8 2011. US Patent 8,055,856.

[201] Juan Gomez-Luna, José Marıa González-Linares, Jose Ignacio Benavides Benitez, and Nicolas Guil Mata. Performance modeling of atomic additions on GPU scratchpad memory. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11), 2013.

[202] Carlos ER Alves, Edson Norberto Cáceres, Frank Dehne, and Siang W Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Computational Science and Its Applications*, pages 249–258. Springer, 2003.

[203] Hsien-Yu Liao, Meng-Lai Yin, and Yi Cheng. A parallel implementation of the smith-waterman algorithm for massive sequences searching. In *Engineering in Medicine and Biology Society. 26th Annual International Conference of the IEEE*, volume 2, pages 2817–2820. IEEE, 2004.

[204] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2), 1974.

[205] Ashwin M Aji and Wu-Chun Feng. Accelerating data-serial applications on data-parallel GPGPUs: a systems approach. Technical report, TR-08-24, Computer Science, Virginia Tech, 2008.

[206] Simon J Pennycook, Gihan R Mudalige, Simon D Hammond, and Stephen A Jarvis. Parallelising wavefront applications on general-purpose GPU devices, 2010.

[207] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, and Joel H Saltz. Efficient irregular wavefront propagation algorithms on hybrid CPU–GPU machines. *Parallel computing*, 39(4), 2013.

[208] Shucai Xiao and Wu-Chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel and Distributed Processing, IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[209] Jeff A Stuart and John D Owens. Efficient synchronization primitives for GPUs. *arXiv preprint arXiv:1110.4623*, 2011.

[210] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 12. ACM, 2011.

[211] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013.

About the Author:

Ang Li was born in August, 1987 in Yongji, Shanxi, China. After finishing his Bachelor in Software Engineering from the Computer Science department of Zhejiang University (ZJU), Hangzhou, China in 2010, he was employed as a Software Engineer in Mintel Consulting, Shanghai, China, from May, 2010 to January, 2011. Then, he was employed as a Computer Science Engineer in CAPS Entreprise, Shanghai, China, from February, 2011 to April, 2012. In August, 2012, he started to pursue a joint-PhD degree from the Electrical and Computer Engineering department of National University of Singapore (NUS), Singapore, and the Electrical Engineering department of Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands, on the topic of "GPU Performance Modeling and Optimization". The research results are presented in this dissertation.

List of Publications:

[1] **Ang Li**, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar and Henk Corporaal. SFU-Driven Transparent Approximation Acceleration on GPUs. In *27th International Conference on Supercomputing (ICS)*. ACM, 2016.

[2] Weifeng Liu, **Ang Li**, Jonathan Hogg, Iain Duff and Brian Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *22nd International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2016.

[3] **Ang Li**, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda and Henk Corporaal. X: A Comprehensive Analytic Model for Parallel Machines. In *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[4] **Ang Li**, Shuaiwen Leon Song, Akash Kumar, Eddy Z. Zhang, Daniel Chavarria and Henk Corporaal. Critical Points Based Register-Concurrency Autotuning for GPUs. In *Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2016.

[5] **Ang Li**, Gert-Jan Van Den Braak, Akash Kumar and Henk Corporaal. Adaptive and Transparent Cache Bypassing on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2015. **Nominated for Best Paper Award and Best Student Paper Award**

[6] **Ang Li**, Akash Kumar, Yajun Ha and Henk Corporaal. Correlation Ratio Based Volume Image Registration on GPUs. In *Microprocssors and Microsystems Journal (MICPRO)*, vol. 39, no. 8, pp. 998–1011. Elsevier, 2015.

[7] Runbin Shi, Zheng Xu, Zhihao Sun, Maurice Peemen, **Ang Li**, Henk Corporaal, Di Wu. A Locality Aware Convolutional Neural Networks Accelerator. In *18th International Conference on Digital Systems Design (DSD)*. IEEE, 2015.

[8] **Ang Li**, Akash Kumar, Y.C. Tay and Henk Corporaal. Transit: A Visual Analytical Model for Multithreaded Machine. In *24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2015.

[9] **Ang Li**, Gert-Jan Van Den Braak, Akash Kumar and Henk Corporaal. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *26th International Conference on Supercomputing (ICS)*. ACM, 2015.

[10] Mohammad Shihabul Haque, **Ang Li**, Akash Kumar, Qingsong Wei. Accelerating non-volatile/hybrid processor cache design space exploration for application specific

embedded systems. In *20th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2015.

[11] **Ang Li** and Akash Kumar. Accelerating Volume Image Registration through Correlation Ratio based Methods on GPUs. In *17th International Conference on Digital Systems Design (DSD)*. IEEE, 2014.