Journal of Parallel and Distributed Computing 127 (2019) 65–88

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity

Mahmoud Khairy [a,b,*], Amr G. Wassal [a], Mohamed Zahran [c]

[a] Computer Engineering Department, Cairo University, Egypt
[b] Electrical and Computer Engineering Department, Purdue University, IN, USA
[c] Computer Science Department, New York University, NY, USA

HIGHLIGHTS

- Recent years have been witnessing the emergence of using GPUs for general purpose computing due to their efficient performance/power ratio.
- Various issues need be addressed in order to rely on GPGPUs as a compelling general purpose accelerator for the next power-limited big-data era.
- Control Divergence, Memory Bandwidth and Limited Parallelism are the three main bottlenecks that limit GPGPU performance.
- Enhancing GPGPU programmability is an important feature for future GPUs to simplify GPGPU programming.
- The aim of this paper is to provide a survey of architectural advances to improve performance and programmability of GPUs.

ARTICLE INFO

Article history:
Received 16 February 2017
Received in revised form 1 November 2018
Accepted 22 November 2018
Available online 14 January 2019

Keywords:
GPGPU
Memory systems
Control divergence
Heterogeneous architecture

ABSTRACT

With the skyrocketing advances of process technology, the increased need to process huge amount of data, and the pivotal need for power efficiency, the usage of Graphics Processing Units (GPUs) for General Purpose Computing becomes a trend and natural. GPUs have high computational power and excellent performance per watt, for data parallel applications, relative to traditional multicore processors. GPUs appear as discrete or embedded with Central Processing Units (CPUs), leading to a scheme of heterogeneous computing. Heterogeneous computing brings as many challenges as it brings opportunities. To get the most of such systems, we need to guarantee high GPU utilization, deal with irregular control flow of some workloads, and struggle with far-friendly-programming models. The aim of this paper is to provide a survey about GPUs from two perspectives: (1) architectural advances to improve performance and programmability and (2) advances to enhance CPU–GPU integration in heterogeneous systems. This will help researchers see the opportunities and challenges of using GPUs for general purpose computing, especially in the era of big data and the continuous need of high-performance computing.

© 2018 Elsevier Inc. All rights reserved.

1. Introduction

Graphics Processing Units (GPUs) have been used for several years as a fixed-function hardware accelerator for 3D Graphics applications. Earlier generations of GPUs were designed to implement the conventional 3D rendering pipeline [103,143]. However, the high computational power, the GPUs can achieve compared with traditional multicore Central Processor Units (CPUs), encourage the developers to use GPUs for compute-intensive non-graphics workloads [222]. At that time, the term General Purpose computing using Graphics Processing Units (GPGPU) has emerged widely. The programmers used graphics APIs (e.g. Direct3D or OpenGL) to access shader cores. The programmers had to map program data appropriately to the available shader buffers and manage the data accurately through the graphics pipeline. Obviously, using graphics APIs for non-graphics general purpose programming was a very difficult task. However, with some heroic efforts, a considerable speedups were achieved [173]. This trend prompted the GPUs vendor to build a more programmable GPU architecture, known as unified shader architecture (e.g. NVIDIA's Tesla [144], NVIDIA's Fermi [161] and AMD Evegreen [13]) and release more-friendly high level abstraction APIs to facilitate GPGPU programming (e.g. NVIDIA's CUDA [166], AMD's CTM [71] and OpenCL [102]). Since then, a new era of GPGPU architecture and programming was unleashed and is still evolving to this day [38,99,161].

GPU acceleration has been widely adopted in high-performance computing (HPC) applications, such as computer vision, graph

* Corresponding author at: Computer Engineering Department, Cairo University, Egypt.
E-mail addresses: makhairy@eng.cu.edu.eg (M. Khairy), wassal@eng.cu.edu.eg (A.G. Wassal), mzahran@cs.nyu.edu (M. Zahran).

https://doi.org/10.1016/j.jpdc.2018.11.012
0743-7315/© 2018 Elsevier Inc. All rights reserved.

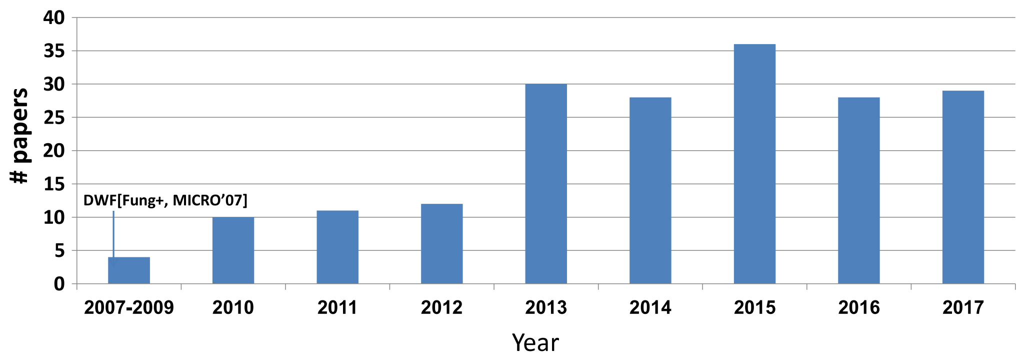

Fig. 1. Number of research papers related to GPGPU published during the last decade at the top-tier computer architecture conferences (MICRO, ISCA, ASPLOS, and HPCA).

processing, biomedical, financial analysis, and physical simulation [80,81]. This is due to the fact that GPUs are able to achieve tremendous computational power and efficient performance-per-watt compared to conventional multicore CPUs. Thus, there is no wonder that a large portion of supercomputers found in Top500 list rely on GPUs [224]. Moreover, the scope of applications that benefit from GPU acceleration has been expanded rapidly during the last decade to include server and cloud workloads [66,74], database processing [24,248] and deep machine learning [35]. However, because GPUs were initially designed to execute regular streaming applications, like graphics workloads, they are still not effective to accelerate some emerging data intensive workloads, due to the lack of irregular execution support, the memory bandwidth bottleneck and the GPGPU programming complexity.

In order to improve the performance, energy efficiency and programmability of GPUs for emerging data intensive workloads, researchers have been diligently working on enhancing GPU architecture for general purpose computing. Fig. 1 depicts the number of research papers related to GPGPUs that were published during the last decade at the top-tier computer architecture conferences. As shown in figure, there has been a growing interest in improving GPGPUs architecture during the last five years. Up to 28 and 29 research papers were published in 2016 and 2017 respectively which represents nearly 16% of the total number of papers.

Fig. 2 characterizes and divides these works into different categories. As we can see, there has been a noticeable interest in improving the performance of GPGPUs by mitigating the impact of control flow divergence [46,48,194,216,221], alleviating on-chip resource contention [97,160,195,274] and improving memory hierarchy performance [22,111,126,193,221,269]. Since GPGPU programming is complex, researchers worked on enhancing the GPU programmability by equipping GPUs with architectural support to improve data sharing and synchronization (e.g. cache coherence [209] and transactional memory [68,208,213]). They also investigated new techniques to boost GPGPU concurrency and multitasking [6,218,255], leading to an increase in available thread level parallelism (TLP) and efficiently utilizing the execution resources. Besides, to amortize the increasing chip area, there have been some efforts to integrate CPU with GPU on the same die chip [11,82,162]. Such designs need to be carefully studied because GPUs execute hundreds of threads that can monopolize on-chip shared resources (e.g. memory controller [17] , on-chip network [128] and last level cache [123]) and this leads CPU applications to be starved. To address this problem, researches have worked on efficiently and fairly managing shared resources between CPU and GPU. Furthermore, they worked on augmenting CPU–GPU architecture with more powerful communication mechanisms and fine-grained data sharing (e.g. unified virtual memory space [183] and CPU–GPU cache coherence [184]). In addition to that, some works have also investigated novel techniques to improve energy and power efficiency [156], define accurate model for performance and power [77,133], create software frameworks to ease GPGPU programming, develop fault tolerance capability and improve the 3D rendering pipeline for graphics workloads. Recently, researchers started looking into novel architecture techniques to build large scalable GPUs that are easy to manufacture [16,154], investigating security breaches on modern GPU [89,159] as well as building software frameworks and designing novel hardware to customize GPUs for deep learning acceleration [75,210].

In this paper, we present a survey of research works that aim to improve GPGPU performance, programmability and heterogeneity (i.e., CPU–GPU integration). Further, we introduce a classification of these works on the basis of their technical approach and key idea. Since it is not possible to review all the research works that are related to GPGPUs, we mainly focus on the following areas to limit the scope of the survey. We only discuss techniques proposed for improving GPGPU performance including (1) control flow divergence mitigation, (2) alleviating resource contention and efficient utilization of memory bandwidth across the entire memory hierarchy, including caches, interconnection and main memory, (3) increasing the available parallelism and concurrency, and (4) improving pipeline execution and exploiting scalarization opportunities. We also include architectural-based techniques that aim to improve GPGPU programmability, e.g. cache coherence, memory consistency, transactional memory, synchronization, debugging and memory management. We also provide a survey on research works which aim to enhance the on-chip integration of CPU–GPU heterogeneous architecture, including on-chip shared resource management and improving CPU–GPU programmability. While our main focus in this work is to discuss micro-architectural approaches, we may also refer to some prominent software- and compiler-based techniques related to our scope. On the other hand, we do not include studies related to performance and energy modeling, employing emerging memory technologies (e.g. non-volatile memory), register file, fault tolerance, works that only focus on improving GPU energy and power efficiency or CPU–GPU power management.[1] Additionally, we only adopt works that are related to many-thread GPU-like accelerator, while works that are concerned with other types of accelerators, such as many-core accelerator [62,201], are not covered in this survey. Further, we only focus on ideas related to general purpose computing, whereas techniques which aim to improve GPUs for graphics workloads are out of scope in this work.

The remainder of this paper is organized as follows, Section 2 presents a brief overview on GPGPUs programming model and architecture, Sections 3–5 review the techniques on improving GPGPU performance by alleviating control flow divergence, efficiently utilizing memory bandwidth and increasing parallelism

[1] For more information about works that aim to improve GPU power and energy efficiency, we refer the reader to [156].

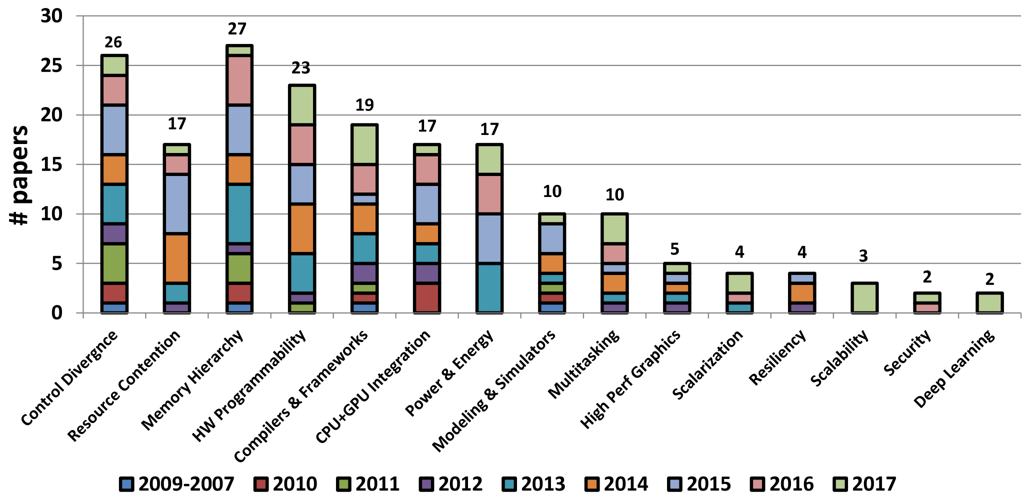

Fig. 2. Characterization of research papers related to GPGPU shown in Fig. 1.

respectively, Section 6 reviews the studies on enhancing GPGPU programmability, Section 7 reviews the works that aim to enhance CPU–GPU integration, Section 8 suggests future research directions and Section 9 concludes.

2. Background

In this section, we give a brief overview on GPGPU programming model and architecture. For more details, we kindly refer the reader to [70,114,166].

2.1. GPGPU programming model

The CUDA [166] or OpenCL [102] programming model allows the programmers to express the available data level parallelism in terms of fine-grain scalar threads. There are two important aspects of GPGPU programming model. First. it follows a CPU off-loading model, which means that the programmer is responsible for moving the data from CPU memory to GPU memory, and when the GPU program execution is finished, he has to move the result from GPU memory back to CPU memory. Second, the GPGPU programming model is a Single Program Multiple Data (SPMD) model, which means that all threads run the same program, however each thread can be at different states (i.e., execute different instructions within the same program).

A typical GPGPU application consists of multiple kernels (or grids). Each kernel contains a group of 2- or 3-dimensional of thread blocks (a.k.a. cooperative thread array or CTA) and each thread block is composed of 3-dimensional fine-grain scalar threads. Threads within the same thread block communicate with each other through a shared on-chip scratchpad memory and synchronization primitives.

During run-time, each of consecutive 32 threads[2] is grouped together to formulate a warp (a.k.a. wavefront). Warps are executed in a Single Instruction Multiple-Threads (SIMT) model. In SIMT execution model: (1) all threads within the same warp execute the same program counter (PC), i.e. execute in a lock-step, to amortize instruction fetch and decode cost efficiency. (2) Threads within the same warp are allowed to follow different control flow paths. (3) A long memory latency is tolerated by a fast warp context switching.

[2] NVIDIA's architecture warp is composed of 32 threads while AMD's architecture wavefront is composed of 64 threads.

2.2. GPGPU architecture

A typical GPGPU,[3] shown in Fig. 3, is composed of multiple GPU cores, named Streaming Multiprocessors (SMs), and a group of memory partitions. Each SM has its own register file, private L1 data cache, constant cache, read-only texture cache and software-managed scratchpad memory, named shared memory. They also contain a group of execution units, such as single instruction multiple data units (SIMDs), special function units (SFUs) and memory units. Each memory partition is attached with a L2 cache slice and a GDDR5 memory controller (or High Bandwidth memory in the modern GPUs [12]). The memory partitions and the SMs are connected together via a high-bandwidth interconnection network.

The warps in GPGPUs are executed on a two-level hierarchical scheduling (thread block scheduling and warp scheduling). A thread block scheduler, as shown in Fig. 3, distributes the thread blocks among SMs in a load-balanced fashion [23]. Thread block is dispatched to a SM only if the required resources of the thread block are available on this SM (e.g. register file, shared memory, warp scheduler entries, etc.). Thread blocks are subdivided by hardware into warps (each warp is composed of a consecutive 32 threads). Each SM contains a number of warp schedulers. The warp scheduler employs a specific policy to schedule the available warps over the execution units.

Each SM contains a memory-coalescing unit that attempts to coalesce memory requests generated by threads within each warp into the fewest possible cache line-sized memory requests. A *memory divergence* occurs when threads in the same warp access different regions of memory in the same SIMT instruction and thus the memory-coalescing unit fails to reduce memory requests. In the worst case scenario, up to 32 independent memory transactions (corresponding to 32 threads per warp) can be generated from a single memory instruction. In addition, as stated earlier, GPGPU hardware implementations employ SIMT execution model in which only one instruction is fetched for all the threads within the same warp. However, a *control flow divergence* occurs when threads in the same warp execute different control flow paths due to branch/loop statement. In this case, GPGPU requires a mechanism to allow each thread to follow its own thread of control. Typically, GPGPUs handle control flow divergence and re-convergence with a hardware-based stack reconvergence [48]. In stack-based

[3] NVIDIA and AMD have different terminology to describe GPU architecture. In this paper, we use NVIDIA's terminology. For more information on different GPU terminology used by other vendors, see Section 2.4.

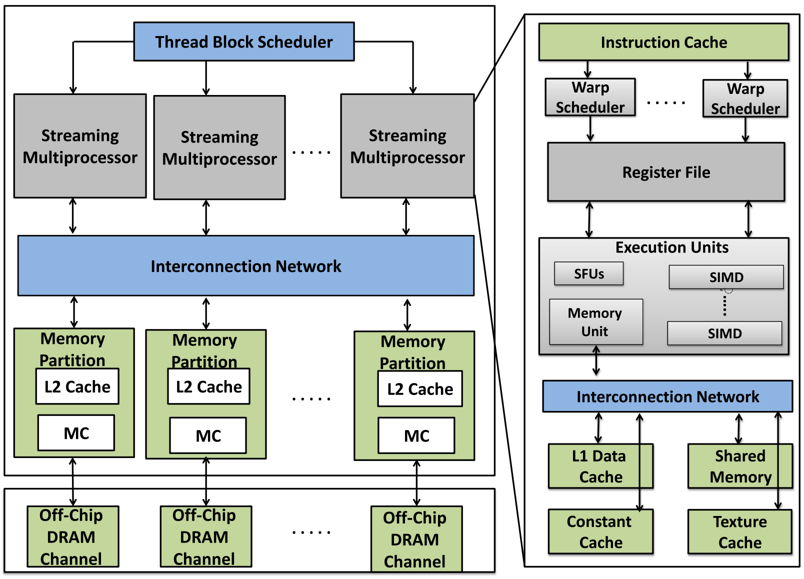

Fig. 3. A Typical GPGPU Architecture.

Table 1
CPU vs GPU design philosophy.

	CPU	GPGPU
Architecture design philosophy	Latency-oriented architecture	Throughput-oriented architecture
Microarchitecture attributes	Large caches, Sophisticated control, Branch prediction , Data forwarding	Small caches, simple control, No branch prediction, No data forwarding
Threads per core	2–4 Threads per core	1024–2048 threads (32–64 warps) per core
Hiding memory latency strategy	Hide memory latency through large Cache Hierarchy	Hide memory latency through massive multithreading zero-overhead switching
Preferable code execution	Sequential Code (if statement, function calling, etc.)	Data-parallel Code (loops, streams, etc.)

scheme, the execution of divergent paths is serialized (i.e., executing the taken path, then the non-taken path) and it ensures reconvergence occurs at or before the immediate postdominator (IPDOM) of the divergent branch. More details and examples of PDOM stack-based reconvergence can be found in [48]. To achieve the maximum performance of the underlying GPGPU hardware, the running program should minimize the occurrence of control flow divergence and memory divergence [114,166].

2.3. *GPU vs CPU design philosophy*

CPU and GPU are two different architectures. Each has its own design philosophy. No one is better than other, since each was designed to execute a specific task in an efficient manner. Table 1 summarizes the differences between both architectures. CPUs are well designed to execute sequential code. They are called latency-oriented architecture, because they are optimized to tolerate long latency instruction and memory access. They devote more transistor area for sophisticated control (e.g. branch predication and data forwarding) to tolerate long latency instruction. They contain larger last level caches (up to 8 MB in current CPUs) in order to hide long memory latency. On the other hand, GPUs are throughput-oriented architecture. They are well designed to execute some parts of applications that contain data-level parallelism (e.g. for/while loops) which requires high arithmetic intensity and large memory throughput. GPUs devote most of the chip area for processing elements that are deeply pipelined to increase the arithmetic throughput. They employ a simpler logic control and small caches. GPGPUs rely on massive number of threads/warps that interleaved with each other with fast context switching in order to hide long memory latency.

Table 2 compares between recent generation of high-end Intel CPU (Skylake core i7 [83]) and NVIDIA Pascal GPU (TitanX [167]) in terms of threads per core, computational intensity (GFLOPs), memory bandwidth (GB/sec) and cache hierarchy size. As shown in table, GPUs achieve a high computational power and memory bandwidth compared to CPUs. The recent NVIDIA Pascal TITANX outperforms Intel Skylake core i7 in terms of GFLOPs and memory bandwidth by 20X and 10X respectively. However, as we stated earlier, this does not mean that GPUs are better than CPUs. GPUs are more efficient when it comes to data-parallel high-computational code (e.g. loops). On the other hand, CPUs are more powerful when it comes to sequential latency-sensitive code (e.g. heavily sequential branches). To take advantage of both CPUs and GPUs, and efficiently utilize the available hardware resources, programmers need to execute the serial parts of their code on CPUs and launch data-parallel parts on GPUs. By this way, they are able to achieve the most out of the underlying hardware. CPU–GPU programming is well known as heterogeneous computing [114,166].

Table 2
Recent CPU vs GPU.

	Threads per core	GFLOPS (FMA)	Memory BW GB/sec	L1 cache	Last level cache	TDP (watts)
Intel Skylake Core i7 [83]	2	512 @4 GHZ	50	64 KB	8 MB	90
NVIDIA TITANX [167]	2048	10,000 @1.4 GHZ	480	48 KB	4 MB	250

Table 3
GPU terminology.

Literature [70]	NVIDIA [164,165]	AMD [10]	Intel [226]
Thread Block Scheduler (CTA Scheduler)	Giga-thread Engine	Ultra-threaded dispatcher	Thread Dispatcher
Multi-threaded SIMD Core	Streaming Multiprocessor (SM)	Compute Unit	Compute Slice
Hardware SIMDs per core	n x 32-wide Vector SIMD	n x 16-wide Vector SIMD + Scalar Unit	n Execution Units (EU), each EU has 2 x 4-wide Vector SIMD
SIMD lane	Stream Processor (or CUDA core)	Stream Core	SIMD lane
Thread Execution model	Single Instruction Multiple Thread (SIMT)	SIMD + Scalar	SIMD
SIMD Thread	Warp	Wavefront	EU Thread
SIMD Thread Scheduler	Warp Scheduler	Wavefront Scheduler	Thread Scheduler
SIMD Thread width (warp width)	=32	=64	variable (=8, 16, 32)
Local Scratchpad Memory	Shared Memory	Local Data Share (LDS)	Shared Local Memory (SLM)
Microarchitecture	Tesla, Fermi, Kepler, Maxwell, Pascal, Volta	Terascale (VLIW), Graphics Core Next (GCN)	Ivy Bridge, Haswell, Skylake

2.4. GPU terminology

GPUs from different vendors have very similar architectures, like the one shown in Fig. 3. Yet, GPU architecture terminology is complex to be tracked. This is due to the fact that each GPU vendor has its own technical terminology and design parameters. For instance, GPU core is named as Streaming Multiprocessor in NVIDIA terminology, while it is named Compute unit in AMD terminology. Further, what NVIDIA defines as *Warp*, is the same as AMD's *Wavefront*. Also, there are some variances in the architecture design parameters. For example, NVIDIA employs 32 threads per warp, whereas AMD employs 64 threads per wavefront. To address this issue, Table 3 summarizes the different GPU terminology and design parameters used by the main GPU vendors.

3. Control flow divergence

Control flow divergence occurs when threads in the same warp execute different control flow paths. Control flow divergence causes significant performance reduction for irregular workloads. The drawbacks of control divergence and irregular execution are four-fold. First, GPUs employ PDOM stack-based mechanism that serializes the execution of divergent paths. This serialization of divergent paths reduces the available thread level parallelism (i.e., the number of active warps at a time) which limits the ability of GPUs to hide long memory instruction latency. Second, control divergence limits the number of active threads in the running warps. As a result, SIMD execution units are not efficiently utilized when a diverged warp is executed. It has been shown that highly-divergent applications cause SIMD execution units to be underutilized for 50% of the time [48]. Third, control divergence may also lead memory divergence wherein threads in the same warp access different regions of memory and thus the memory-coalescing unit fails to reduce memory requests. Memory divergence causes huge pressure on memory resources and leads long memory latency and performance degradation [195]. Fourth, irregular applications tend to cause workload imbalance in such a way that assigned work (i.e., active threads per CTAs) to some GPU cores are larger than others. This typically happens in graph processing, and results in inefficient utilization of execution resources and longer execution time. To this end, different approaches have been proposed in literature to alleviate control flow divergence. We summarize these approaches in Table 4 along with the advantages and disadvantages of each aprroach. In the following paragraphs, we discuss these approaches in more detail. Since it is not possible to review all the related works in each approach, we only discuss one or two works from each approach.

3.1. Regrouping divergent warps

Fung et al. [48] proposed dynamic warp formation (DWF) that is not restricted to the conventional stack-based PDOM reconvergence mechanism. Instead, DWF dynamically re-forms divergent warps into new non-divergent warps on the fly. That is, at runtime, when divergence occurs, the hardware forms new denser warps by combining consistent threads from different warps that follow same control flow path. However, DWF performance largely depends on the warp scheduling policy to increase the opportunity of forming denser warps. Moreover, DWF does not reconverge diverged warp at IPDOM in order to amortize coalesced memory

Table 4
Alleviating control flow divergence approaches.

Approach	Works	Pros (+) and Cons (−)
Dynamically regrouping divergent warps	[25,48,120,148]	+ improves SIMD utilization. − leads to starvation eddies. − increases memory divergence and inefficient memory utilization.
Large Warp/CTA compaction	• Large warp compaction [46,160,191] • Thread remapping/ rearranging [189,226,257]	+ improves SIMD utilization. − CTA coarse-grain synch. − reduces Thread Level Parallelism (TLP).
Multi-path execution	[44,152,190,221]	+ improves TLP. − still low SIMD utilization.
MIMD-like architecture	• Variable warp sizing [119,151,194] • Stack-less architecture [216] • Multiple SIMD, Multiple Data (MSMD) [235] • Temporal-SIMT [99,125]	+ improves both SIMD utilization and TLP. − significant changes to baseline. − In [125,235] , it requires explicit synch at IPDOM.
Dynamic kernels/threads creation and aggregation	• Dynamic threads [53,105,171,215] • Dynamic CTAs [1,236,237] • Dynamic kernels [220] • Software control dynamic kernels [32,63]	+ improves SIMD utilization and mitigates memory divergence. − kernel launch overhead. − workload imbalance and low TLP.
In-core special unit accelerator	• Dataflow accelerator [233,234] • Neural accelerator [260] • Scalar unit [257]	+ improves utilization and energy efficiency. − requires changes to SW stack. − works only for specific categories of workloads.
Compiler- and software-based approaches	• Software optimization [272] • Compiler-based optimization [40,84,101,122,125] • Approximate/neural acceleration [61,199] • Warp-Consolidation [136].	+ no HW changes - still not as efficient as hardware approaches, especially in highly divergent workloads (e.g. graph processing)

address of converged warps. This results in starvation eddies and inefficient utilization of memory bandwidth [49].

3.2. Large Warp/CTA compaction

Fung and Aamodt [46] proposed thread block compaction (TBC) that allows a group of warps, that belong to the same thread block, to share the same PDOM stack. Hence, at a divergent branch, threads from the same thread block are compacted into new more dense warps to better fill the SIMD lane; thereby achieve better performance. However, TBC stalls all warps within a CTA on any potentially divergent branch until all warps reach the branch point. This coarse-grained barrier synchronization, which is not required for correctness, reduces thread level parallelism and decreases performance in some cases.

Rhu and Erez [191] proposed compaction-adequacy predictor (CAPRI), a fundamentally new approach to branch compaction that avoids the unnecessary synchronization required by previous technique (TBC). CAPRI dynamically identifies the compaction-effectiveness of a branch and only stalls threads that are predicted to benefit from compaction.

Rhu and Erez [189] observed that control frequently diverges in a manner that prevents compaction because of the way in which the alignment of a thread to a SIMD lane is fixed. Therefore, they proposed SIMD lane permutation (SLP) as an optimization to expand the applicability of compaction in case of conventional compaction technique is ineffective. Further, SLP seeks to rearrange how threads are mapped to lanes such that it allows even programmatic branches to be compacted effectively; thereby improving SIMD utilization.

3.3. Multi-path execution

Rhu and Erez [190] proposed dual-path stack (DPS) that executes the two divergent paths (i.e., the taken and non-taken paths) in parallel, instead of stacking the paths one after the other in PDOM stack. DPS ensures reconvergence of the two divergent paths at immediate post-dominators. To avoid false dependencies between independent splits, DPS maintains separate scoreboard units for each path and checks both units to make sure there are no pending dependencies across divergence and reconvergence points.

ElTantawy et al. [44] proposed a branch divergence handling mechanism which enables efficient multi-path execution and allows divergent paths to interleave with each other. Compared to the previous method (DPS) that enables interleaving only two divergent paths, this proposed method allows arbitrary number of divergent control flow paths to interleave; thereby increasing the available thread-level parallelism and improves the performance. Applications with nested divergent branches significantly benefit from multi-path execution.

3.4. MIMD-like architecture

Rogers et al. [194] observed that regular applications perform better with a wider warp size, whereas divergent applications achieve better performance with a smaller warp size. In order to take the advantage of both mechanisms, they proposed Variable Warp Sizing (VWS) which improves the performance of divergent applications by using a small base warp size in the presence of control flow and memory divergence. On the other hand, for regular applications, VWS groups sets of these smaller warps together by ganging their execution in the warp scheduler and thus amortizing the energy consumed by fetch, decode, and warp scheduling across more threads. In short, VWS makes use of a hierarchical warp scheduler, enabling divergent applications to execute multiple control flow paths while forcing convergent ones to operate in lock-step.

Lee et al. [125], Keckler et al. [99] proposed Temporal-SIMT (T-SIMT), a new microarchitecture where each warp is mapped to a single lane, and the threads within a converged warp dispatch an instruction one after the other over successive cycles. Upon divergence, threads progress independently and instructions simply dispatch for a single cycle. The independent lanes essentially operate as a traditional multithreaded MIMD processor; and hence divergence does not reduce the SIMD units utilization.

Wang et al. [235] proposed Multiple SIMD Multiple Data (MSMD), a flexible SIMD datapaths that can be repartitioned among multiple control flow paths upon divergence. That is, multiple paths can be issued to the same SIMD vector and execute in parallel as long as the data fragments generated from multiple paths are equal to the SIMD width.

The main advantage of MIMD-like approach compared to previous techniques is that it improves both SIMD efficiency and thread level parallelism for divergent applications, while previous works only improve either of them. However, it requires quite large changes to the baseline architecture. Further, T-SIMT and MSMD need a sophisticated hardware mechanism to ensure load balancing of divergent threads over SIMD lanes, especially in highly divergent data-dependent applications, such as graph processing. They also lack a hardware mechanism to track reconvergence of diverged warp in order to perform memory address coalescing, therefore, it is required to insert special synchronization instructions to regroup divergent warps at the IPDOM.

3.5. Dynamic kernels/threads

Steffen and Zambreno [215] proposed a SIMT architecture in which runtime threads are able to create kernel, called microkernel, in order to improve SIMT efficiency for ray tracing algorithms. In the proposed scheme, new threads are created to replace branching statements that cause low SIMD efficiency. The proposed architecture is composed of two components. The first part allows the original threads to create new processing threads. In the second part, the new threads are grouped together to formulate new warps. These new warps execute the same microkernel without branching statements; thereby improving the SIMD efficiency.

Kim and Batten [105] observed that one of the common software optimization techniques to improve workload balancing among threads in irregular applications, is to employ a shared software worklist (SWWL). That is, a group of persistent threads continuously access the SWWL to determine the workload to operate on. However, one drawback of SWWL is that hundreds of threads simultaneously access the SWWL resulting in high memory contention and sub-optimal load balancing. Thus, they proposed equipping GPGPUs with a hardware work-list (HWWL) that can be accessed via a software layer similar to SWWL. To avoid memory contention, HWWL is implemented as distributed hardware queues attached with the GPGPU execution units. In order to achieve optimal intra-core and inter-core workload balancing, they employ a local and global work redistribution unit that achieves dynamic, fine-grain load balancing among the running threads.

In order to improve the efficiency of irregular GPGPU workloads, modern GPUs [164,165] offer a new execution model, named CUDA Dynamic Parallelism (CDP) [163], that allows kernels to be launched from GPU device without going back to the CPU host. This new functionality can be used to dynamically form structured parallelism (DFP) from unstructured irregular applications (i.e., convert irregular divergent behavior to regular convergent behavior). Wang and Yalamanchili [239] observed that CDP improves the performance; however, the huge kernel launching overhead could negate the performance benefit of DFP. The overhead is due to the large number of launched kernels, the associated memory footprint and the low number of running warps per core. Thus, Wang et al. [236] proposed new mechanism, called Dynamic Thread Block Launch (DTBL), that employs light-weight thread block rather than heavy-weight device kernel for DFP. The dynamic creation of thread blocks can effectively increase the number of running warps per core. Also, DTBL overhead and the associated memory footprint are significantly lower than that of kernel launch. As a result, DTBL enables more efficient support of DFP.

Sartori and Kumar [199] proposed free launch, a new approach to overcoming the shortcomings of dynamic kernel launching. The proposed approach reuses the persistent parent threads to process the child kernel tasks, which is able to eliminate the launching overhead of dynamic kernels. It adaptively assigns tasks to parent threads in order to achieve a better load balance over GPU cores. The new technique does not require any hardware changes and shows significant improvement on a real GPU silicon.

3.6. Special unit accelerator

Voitsechov and Etsion [233,234] proposed an energy-efficient hybrid Dataflow/von Neumann architecture which aims to achieve the energy efficiency of dataflow accelerator and the generality of the von Neumann model for partitioning and executing large kernels. The proposed architecture concurrently executes vector of threads on dataflow accelerator instead of the conventional instruction-fetch global pipeline manner, while it still coalesces different threads that need to execute the same basic block based on von Neumann control flow semantic. In their approach, active threads are coalesced according to their control flow and are executed concurrently, therefore improving execution units utilization for irregular workloads.

3.7. Compiler- and software-based approaches

For compiler and software optimizations, Zhang et al. [272] proposed an optimized runtime thread-to-data remapping that removes non-coalesced memory accesses and thread divergences. Diamos et al. [40] suggested thread frontiers as an alternative mechanism to the immediate post-dominator reconvergence algorithm, which does not guarantee the earliest reconvergence point in an unstructured control flow. Sartori and Kumar [199] observed that many GPU applications exhibit error tolerance to propose branch and data herding. Thus, they proposed a technique that alleviates branch divergence by forcing the divergent threads to execute the most popular path. A profiling framework was proposed to maximize performance while maintaining acceptable output quality.

4. Efficient utilization of memory bandwidth

GPGPU caches and memory hierarchy suffer from severe resource contention which may degrade the performance due to the massive multithreading. Memory divergence is the main source of GPU resource contention, especially caches contention [195]. Memory divergence occurs when threads in the same warp access different regions of memory in the same SIMT instruction. Moreover, as we discussed earlier, GPUs are throughput-oriented architecture that run hundreds of threads simultaneously, thus many

GPGPUs are sensitive to memory bandwidth. Designing a high-bandwidth many-thread aware memory hierarchy is required to sustain the high memory bandwidth demands of GPGPU applications. In Section 4.1, we discuss proposed techniques to alleviate resource contention, and in Section 4.2, we present works that aim to design a throughput-oriented high-bandwidth memory hierarchy.

4.1. Alleviating cache thrashing, and resource contention

Due to the massive multithreading and the limited capacity of L1 cache, irregular GPGPU applications cause severe cache contention throughout memory hierarchy. Different methods have been proposed in literature to alleviate the problems associated with GPU caches and resource contention. We summarize these methods in Table 5. In the following paragraphs, we discuss these methods in more detail.

4.1.1. Two-level warp scheduling

Narasiman et al. [160] observed that, in commonly-used round-robin warp scheduling policy (RR), all warps arrive the same long memory latency instruction within the same time; thereby reducing the available warps ready to execute and significantly hindering the ability to hide long memory latency. Further, running all the warps at the same time may destroy intra-warp data locality and cause cache contention. To address these issues, they proposed two-level round-robin warp scheduling (TL-RR), in which the warps are split into fetch groups. TL-RR executes only one fetch group at a time and it schedules warps from the same fetch group in a round-robin fashion. When the running warps reach a long latency operation, then the next fetch group is prioritized. Hence, warps reach long latency instructions at different times, leading to better memory latency hiding capability. Also, this reduces cache contention by running lower number of warps that access the L1 cache.

Jog et al. [94] proposed CTA-aware locality-aware warp scheduling policy (OWL) that improves the previously proposed TL-RR warp scheduling. OWL augments the TL-RR with CTA-awareness, such that warps are split into groups of CTAs basis rather than warps basis, resulting in increased intra-CTA locality. Further, in contrast to the strict TL-RR policy that prioritizes a group of CTAs until they stall, OWL gives a group of CTAs higher priority when their data exist at the L1 cache such that they get the opportunity to reuse it, therefore improving L1 hit rates and alleviating cache contention. They also proposed Bank-Level-Parallelism-aware (BLP-aware) warp scheduling and memory-side data prefetcher in order to improve DRAM bandwidth utilization via exploiting memory bank-level parallelism and increasing row-buffer locality.

4.1.2. Coarse-grained CTA throttling

Kayıran et al. [97] demonstrated that executing the maximum possible number of CTAs on a GPU core (i.e., increasing TLP to the maximum) does not always lead to better performance. In fact, they found out that high number of concurrently running warps might generate massive number of pending memory requests, thereby create contention in cache, interconnection and main memory. This results in longer memory latency and performance slowdown. To alleviate resource contention, they proposed dynamic CTA scheduling mechanism (DYNCTA), which aims to allocate the optimal number of CTAs per GPU core that alleviate memory contention according to an application characteristics. DYNCTA dynamically adjusts over sampling periods the number of active CTAs per GPU core that reduces the memory latency without sacrificing the available TLP. In other words, it tries to find a trade-off number of lower active CTAs that reduces resource contention while maintaining a reasonable number of active CTAs that are able to tolerate long memory latency and keep the SIMD execution units utilized.

Lee et al. [130] proposed two orthogonal thread block scheduling schemes that aim to reduce resource contention and exploit inter-CTA locality. First, similar to previously work DYNCTA, Lazy CTA scheduling (LCS) was proposed to determine the optimal number of CTAs per core. In contrast to DYNCTA that monitors the workload behavior for the entire kernel execution, LCS leverages GTO scheduler to find the optimal number of thread blocks at the early beginning of kernel execution. They found out that calculating the number of executed instructions under GTO within the first sampling execution period can be used as a good measure to find the best number of thread blocks. Second, they also showed how block CTA scheduling (BCS), which assigns consecutive thread blocks to the same SM, can improve the inter-block locality and increase L1 hit rate. The two proposed LCS and BCS are synergistically combined together to achieve the maximum performance.

4.1.3. Fine-grained warp throttling

Rogers et al. [195] noticed that, due to the massive multithreading and the limited capacity of L1 cache, divergent GPGPU applications cause severe cache contention. Hence, they proposed Cache Conscious Wavefront Scheduling (CCWS), a hardware mechanism that adaptively throttles the number of active warps to alleviate L1 cache thrashing. Unlike the previous approach, CCWS does the throttling at fine-grain (warp level) rather than coarse-grain (CTA level). CCWS uses a victim tag array, called lost locality detector, to detect warps that have lost locality due to thrashing. These warps are prioritized till they exploit their locality while other warps are descheduled (i.e., not allowed to issue any load instructions). CWWS can outperform any cache replacement scheme including the Belady-optimal policy. Later, Rogers et al. [196] introduced a follow-up work and proposed Divergence-Aware Warp Scheduling (DAWS). DAWS is a divergence-based cache footprint predictor to calculate the amount of locality in loops required by each warp. DAWS uses these predictions to prioritize a group of warps such that the cache footprint of these warps do not exceed the capacity of the L1 cache. Compared to CCWS, DAWS is a proactive mechanism that prevents lost locality before it happens and adapts throttling mechanism to the current warps divergence state.

4.1.4. Throttling and cache bypassing

Li et al. [137] observed that previous CTA or warp throttling techniques leave memory bandwidth and other chip resources (L2 cache, interconnection and execution units) significantly underutilized. Thus, they proposed a cache bypassing scheme built on top of CCWS, named Priority-based Cache Allocation (PCAL). At the beginning of kernel execution, PCAL executes an optimal number of active warps, that alleviates thrashing and conflicts, then extra inactive warps are allowed to bypass cache and utilize the other on-chip resources. Thus, PCAL reduces cache thrashing and effectively utilizes the chip resources that would otherwise go unused by a pure thread throttling approach. A similar approach was proposed by Zheng et al. [274], called Adaptive Cache and Concurrency (CCA). CCA improves DAWS by allowing extra inactive warps and some streaming memory instructions from the active warps to bypass the L1 cache and utilize on-chip resources.

Li et al. [139] proposed a software-based CTA clustering technique to reshape the default CTA scheduling in order to exploit inter-CTA locality by grouping the CTAs with potential reuse together on the same SM. Further, they proposed a software CTA throttling and cache bypassing mechanism that limits the number of concurrent CTAs on an SM to alleviate the resources contention. The proposed techniques show a significant performance gain on real GPU systems.

Table 5
Alleviating cache thrashing, and resource contention works.

CTA/Warp scheduling policy	• Two-level warp scheduling: [54,94,160,266] • Coarse-grained CTA scheduling/throttling: [97,130,211,252,254] • Fine-grained warp scheduling/throttling: [124,149,195,196,244,268] • Throttling + cache bypassing [28,137,139,274] • Critical warp awareness: [14,21,121,132,145,177]
Cache management scheme	• Cache replacement policy and cache bypassing: [28,33,39,87,90,117,131,138,223,243] • Compiler- and software-based cache bypassing: [34,88,135,157,253,254]
Ordering buffers	[87,115,203]
Resource tuning	[181,204]

4.1.5. *Critical warp awareness*

Lee et al. [121] observed that, due to memory contention and workload imbalance of some irregular GPGPU applications, some warps may be assigned more workload and exhibit longer latency compared to other warps within the same Thread Block. Hence, fast warps are idle at a synchronization barrier or at the end of kernel execution until the critical (i.e., the slowest) warp finishes execution. Thus, the overall execution time is dominated by the performance of these critical warps. To tackle this problem, they proposed criticality-aware warp acceleration (CAWA). CAWA dynamically identifies critical warps and coordinates warp scheduling and cache prioritization to accelerate the critical warp execution.

4.1.6. *Cache management and bypassing*

Chen et al. [33] proposed G-Cache to alleviate cache thrashing. To detect thrashing, they equip L2 cache tag array with extra bits (victim bits) to provide L1 cache with some information about the hot lines that have been evicted before. An adaptive cache replacement policy is used by L1 cache to protect these hot lines. Chen et al. [28] continued their work and proposed Coordinated Bypassing and Warp Throttling (CBWT). CBWT adopts a thrashing-resistant CPU cache management scheme, Protection Distance Prediction (PDP) [42], to GPU cache. PDP employs cache bypassing to protect hot cache lines. Excessive bypassing may over-saturate the on-chip network. Therefore, cache bypassing policy is coordinated with a dynamic warp throttling mechanism to prevent on-chip resources from being over-saturated.

4.1.7. *Ordering buffers*

Jia et al. [87] proposed Memory Request Prioritization Buffer (MPPB) which improves caching efficiency of massively parallel workloads. The idea of MRPB is two-fold. First, a FIFO requests buffer is used to reorder memory references so that requests from the same warp are grouped and sent to the cache together in a more cache-friendly order. This results in drastically reducing cache contention and improving use of the limited per-thread cache capacity. Second, MRPB allows memory request that encounters associativity stall to bypass L1 cache. Thus, when bursts of conflicting memory requests occur, cache bypassing increases request processing throughput and prevents caches from becoming congested.

4.1.8. *Resource tuning*

Sethia and Scott [204] noticed that the high number of active threads, GPGPU executes concurrently, leads to contention and saturation for one of the on-chip resources like execution units, data cache and memory bandwidth, while leaving other resources underutilized. To address this issue, they proposed, Equalizer, a dynamic runtime system that tunes number of thread blocks, core and memory frequency to match the requirements of the running kernel, leading to efficient execution and energy saving. The sources of efficiency behind Equalizer are two-fold. First, it reduces power by throttling underutilized resources with minimal performance degradation. Second, the bottleneck resource is boosted to mitigate contention and achieve higher performance without significant energy increase.

4.2. *High-bandwidth many-thread-aware memory hierarchy*

GPUs are throughput-oriented architecture that run hundreds of threads simultaneously. For this reason, memory bandwidth is a critical bottleneck for modern GPU systems due to limited off-chip bandwidth and the increasing gap between GPU core and memory performance. Even with high-bandwidth memory, a high-throughput interconnection throughout the memory hierarchy is also required to supply cores with data as fast as memory bandwidth, otherwise the expensive high-bandwidth memory will be useless. Thus, designing a high-bandwidth many-thread aware memory hierarchy is required to sustain the memory bandwidth demands of the running threads. Different methods have been proposed in literature to alleviate memory bandwidth bottleneck. We summarize these methods in Table 6. In the following paragraphs, we discuss these methods in more detail.

4.2.1. *Mitigating off-chip bandwidth bottleneck*

Rhu et al. [193] observed that regularly structured GPGPU applications benefit from coarse-grained (CG) memory access (i.e., fetching large block size per memory access) by exploiting spatial locality and achieving the peak memory bandwidth. On the other hand, emerging irregular workloads benefit from fine-grain (FG) memory access by avoiding unnecessary data transfers, that may be happened under CG policy, resulting in efficient utilization of the off-chip memory bandwidth and energy saving. To achieve the best of both schemes, they proposed a locality-aware memory hierarchy (LAMAR) that adaptively tunes the memory access granularity for the running kernel. LAMAR employs CG accesses for kernels with high temporal and spatial locality, while applying FG accesses for irregular divergent workloads in attempt to reduce memory over-fetching. LAMAR employs a low-cost hardware predictor mechanism, based on bloom filter, to determine the best access granularity.

It has been found that off-chip memory bandwidth is a limiting bottleneck for many memory-intensive GPGPU applications. Data compression is one of the powerful techniques that can be used to alleviate the memory bandwidth bottleneck. To that end, Vijaykumar et al. [231] proposed, Core-Assisted Bottleneck Acceleration (CABA) framework, that exploits the underutilized computational resources to perform useful work and alleviate different bottlenecks in GPU execution. For instance, to alleviate memory bandwidth bottleneck, CABA dynamically creates assist warps that execute with the original warps side by side on the same GPU core. Assist warps opportunistically use idle computational units to perform data decompression for the incoming compressed cache blocks and compression for the outgoing cache blocks, leading to less transferring data from memory and mitigating memory bandwidth problem.

Table 6
High-bandwidth many-thread-aware memory hierarchy works.

Reducing Off-Chip Memory Traffic and Mitigating Off-chip Bandwidth Bottleneck	• Reducing memory over-fetching: adaptive access granularity [193], removing duplicate memory traffics [176], fine-grained DRAM [169] • Data compression: [182,200,231] • Value prediction and approximation: [198,261] • Processing-In-Memory: [79,180]
Memory Divergence Normalization	• L1 cache normalization [221,243] • Memory scheduler normalization [27] • L2 cache and memory scheduler [18]
Interconnection Network	[22,85,107,276]
Main Memory Scheduling	• Low-complexity memory scheduling [110,269] • Managing DRAM latency divergence [18,27,118] • Fairness-awareness [91,92] • Critical-awareness [134]
Heterogeneous Memory Management	• Page placement strategy: [4,5]
Reducing CPU–GPU Memory Transfer Overhead	• Overlapping GPU execution and data transfers: [60,106,109,111,147] • Kernel fusion/fission: [248] • Shared HMC-based memory systems: [108]

Samadi et al. [198] introduced an approximate computing for GPUs that allows trading off performance with output accuracy based on the desired level of quality. They proposed three compiler optimization techniques. One optimization is data packing wherein the number of bits used to store input arrays is decreased, thereby sacrificing precision to reduce memory bandwidth contention and increase throughput. Similarly, Yazdanbakhsh et al. [261] presented an approximation technique in which the GPU drops some portion of load requests which miss in the cache after approximating their values. The load requests that have the smallest impact on quality are selected for approximation. Dropping requests mitigates memory bandwidth demand by removing them from the system. A drop rate is selected by the compiler to control the performance/energy and quality tradeoff.

3D-stacked memory technology provides a promising opportunity to mitigate the memory bandwidth bottleneck by tightly connecting a logic layer and DRAM layers with high bandwidth connections. In these architectures, it enables processing in memory (PIM) by offloading memory-intensive computation portions to be executed in memory logic layer. However, it is challenging on how to enable computation offloading and data mapping transparently without burdening the programmer. To address this challenge, Hsieh et al. [79] proposed a compiler-based technique for offloading code segments to PIM logic layer based on a simple cost–benefit analysis. Further, they introduced a runtime prediction mechanism to locate data that will be accessed by offloaded code in the same memory stack. The proposed techniques minimize off-chip bandwidth consumption without requiring programmer effort.

4.2.2. Memory divergence normalization

Chatterjee et al. [27] showed experimentally that there is a high variance in the latency of memory requests issued by threads of the same warp. This memory latency divergence occurs because of inter-warp interference at the cache and DRAM memory controller. In this case, the warp is unable to make progress until the last memory request from a vector load instruction is returned. To alleviate this problem, they proposed a memory scheduling policy to balance this latency divergence. They added an interconnection network between all memory controllers for exchanging memory access latency divergence information. By using this information, the memory access latency divergence is reduced. Similarly, to alleviate the same issue, Ausavarungnirun et al. [18] proposed a more holistic solution that ensures memory latency normalization at both DRAM system as well as last level cache.

Jog et al. [93] noticed that RR and TL-RR are not effective to hide memory latency in the presence of simple prefetcher. They observed that simple prefetcher that generates a prefetch memory request for the consecutive warp of the same fetch group or for the warp of the next fetch group may cause late prefetching (i.e., prefetching request and the original memory request are issued around the same time) as well as inaccurate useless prefetching. To overcome these problems and avoid the complexity of using sophisticated prefetchers, they proposed prefetch-aware warp scheduling, that coordinates simple data prefetcher and warp scheduling in an intelligent manner such that the scheduling of two consecutive warps are separated in time, and thus prefetching becomes more effective.

4.2.3. Interconnection network

As previously mentioned, GPGPUs run hundreds of threads concurrently. In order to quickly feed these massive number of threads with the required data, GPGPUs are equipped with high bandwidth DRAM memory (e.g. GDDR or HBM). However, a high-bandwidth interconnection is also required to supply cores with data as fast as memory bandwidth, otherwise the expensive high-bandwidth memory will be useless. Building a high throughput interconnection network could be very expensive in terms of area and power consumption. Hence, Bakhoda et al. [22] proposed a throughput-effective on-chip network that is optimized for higher application throughput per area. In particular, they proposed a checkerboard organization that exploits half-routers to reduce network cost with minimal loss in performance. Further, they extended the checkerboard network with multi-port routers to address the many-to-few-to-many bottleneck and provide a throughput-effective microarchitectural technique to improve network performance by increasing the terminal bandwidth of the network.

4.2.4. Main memory scheduling

Yuan et al. [269] observed that the interconnection network which is between cores and memory controllers can destroy memory access row-buffer locality. To tackle this problem, they proposed an interconnection network arbitration scheme to reserve row locality and reduce complexity circuit design of FR-FCFS DRAM controller. To achieve that, they employ an interconnection arbitration scheme to prioritize memory requests accessing the same row first. Using this scheme, they achieve a performance similar to the complex FR-FCFS only using a simple FIFO memory controller.

4.2.5. Heterogeneous memory management

In next-generation cache-coherent non-uniform-memory-access (CC-NUMA) CPU–GPU systems, both CPU and GPU will be able to access a unified globally-addressed memory, and thus programmers no longer need to explicitly transfer data to and from GPU memory. In such systems, a question arises for GPGPU workloads, where to initially place a new memory page, in the far capacity-optimized CPU memory or in the near bandwidth-optimized GPU memory, such that the overall system efficiency is maximized? Agarwal et al. [5] showed that applying traditional Linux page placements policies, which have been used for CPU-only NUMA systems and aim to minimize the memory request latency, may not be effective in CPU–GPU NUMA systems. This is due to the fact that GPU performance is more sensitive to memory bandwidth. Thus, they proposed bandwidth-aware placement that maximizes GPU performance by balancing page placement across the memories based on the aggregate memory bandwidth available in a system.

4.2.6. CPU–GPU memory transfer overhead

Previous work have shed light on the importance of CPU–GPU data transfer and communication in case of discrete GPU is connected with CPU via PCIe [60,147]. They showed that data transfer and kernel launch overhead can be a dominant and limited factor to the overall GPU performance, which may negate the potential speedup from using GPU acceleration and limit the scope of applications that benefit from GPUs. Typically, CPU sends data to GPU in coarse-grained data transfers and synchronization. Therefore, GPU has to wait and synchronize for all of data blocks to be transferred and ready, but this long latency often results in a performance slowdown. To address this issue, Lustig and Martonosi [147] proposed fine-grained CPU–GPU synchronization enabled by a hardware-managed full-empty bits to track when regions of data have been transferred. Thus, the GPU is able to start execution once the required block of data is available. Software-level APIs are proposed to allow programmer to launch kernel earlier and overlap data transfer with execution.

5. Increasing parallelism and improving execution pipelining

GPGPUs achieve the highest performance by running many concurrent threads on their massively parallel architecture. However, some applications have a low number of active thread blocks due to the small input size or the unavailability of some required resources in SM (e.g. registers or shared memory), thus they fail to efficiently utilize the execution units. This results in inefficient utilization of execution unit and hinders the GPU ability to hide long memory latency. Previous works proposed new techniques in order to reduce resource fragmentation and run the maximum number of warps per core. Further, other approaches proposed running multiple applications on the same GPU to exploit these underutilized resources and increase overall throughput. Another way to improve execution efficiency and increase parallelism is to exploit scalar opportunities and value similarity between the running warps such that scalar instructions can be executed concurrently along with other SIMT instructions. In this paper, we summarize these works that aim to increase parallelism and improve execution pipelining in Table 7. In the following subsections, we discuss these works in more detail.

5.1. Reducing resource fragmentation and increasing parallelism

Gebhart et al. [55] noticed that application's needs for on-chip storage such as registers, cache, and scratchpad memory, vary widely from one application to another, whereas traditional GPU designs fix the capacities of these storages at design time. Thus, many GPGPU applications are limited by the capacity of a particular local storage. To provide more design flexibility, they proposed a unified local memory which integrates the register file, L1 cache, and scratchpad memory into one large on-chip storage. Then, the hardware can dynamically partition the on-chip storage according to each application's needs. This flexible partitioning of capacity increases available parallelism and improves both performance and energy consumption.

Yang et al. [258] discussed the problems that are associated with thread block level resource management and warp-level divergence. They observed that different warps in a thread block can finish at different times. In this scenario, the allocated resources (e.g. registers, shared memory, etc.) of the early completed warps are not available until the longest running warp in the same thread block finishes. This results in severe resource underutilization and affects the TLP that may be achieved if new warps are launched instead. They classified the resource underutilization problems as temporal and spatial. Temporal underutilization, as previously mentioned, is caused due to differences in run times of warps of a thread block, whereas spatial underutilization is caused because of unavailability of enough resources to launch a new thread block. To overcome both spatial and temporal resource underutilization problems, they proposed, WarpMan, a fine-grained warp-level resource management instead of coarse-grained thread-block-level. In particular, they introduced a hardware solution to launch a partial thread block when there are not enough resources to launch a full thread block. This way, WarpMan effectively increases the number of active warps and parallelism.

5.2. GPU multitasking

Previous work [2,174] observed that many GPGPU applications are not able to proportionally scale and effectively utilize the growing compute resources with each GPU generation. Therefore, multitasking (or multi-programmed) execution, in which multiple kernels from the same application or from different applications execute concurrently on the same GPU platform, can efficiently utilize the growing GPU resources. This paradigm has two advantages. First, it significantly improves the GPU utilization and overall throughput. Second, hardware support for concurrent applications will facilitate operating system multitasking as well as consolidate jobs from multiple independent users, which is an important feature to enable GPU virtualization and deploy GPUs in the cloud platform. To this end, Pai et al. [174] proposed spatial multitasking of GPU across concurrent applications. That is, multiple applications execute simultaneously on different cores within the same GPU substrate. They evaluated a variety of heuristics on how to partition GPU cores among running applications. On the other hand, Awatramani et al. [20] and Lee et al. [130] proposed mixed concurrent kernels execution, in which two applications execute concurrently on the same core. They showed how such sharing execution can improve the overall system throughput, especially mixture of memory-intensive and compute-intensive workloads. In this mixture, the compute-intensive workload's warps hide the memory latency of memory-intensive workload's warps and thus efficiently utilize the execution units.

Previous spatial multitasking work do not discuss how to execute new application on GPU while there exist some applications running concurrently and they fully consume the available resources. In this scenario, the new application will have to wait until one of the running applications finishes execution. This sharing paradigm is well known as cooperative multitasking. However, this strategy may cause high-priority application suffering from a long latency to execute. Additionally, it makes GPU vulnerable to a malicious attacks. For example, malfunctioning application, that may never yield GPU, will prevent other applications to start execution. Thus, a task preemption strategy is required to improve GPU multitasking. Tanasic et al. [218] proposed two preemption mechanisms, context switching and draining, that can be used to implement GPU multi-kernel scheduling policies. Context switching mechanism preempts all the running thread blocks and saves

Table 7
Increasing parallelism and improving execution pipelining works.

Reducing Resource Fragmentation and Increasing Parallelism	[55,230,251,258,265]
Multitasking (Concurrency Support)	• Hardware-based spatial multitasking – Resources partitioning [2,6,7] – Fairness-aware memory subsystem [91,92] – Mixed Concurrent kernels execution [20,130,179,240–242,255] • Software-based spatial multitasking: [59,174,275] • Preemptive Multitasking: [178,218,249]
Exploiting Scalarization Opportunities	• Compiler-based static detection [10,29,125,217,257] • Hardware-based dynamic detection [36,56,113,141,246,250,262]
Improving Execution Pipelining	Data forwarding and Out-of-Order execution [57,112], Sliced datapath [56], In-core special unit accelerator [234,260]

the execution contexts to off-chip memory (i.e., similar to conventional operating system schedulers), whereas draining allows running thread blocks to continue execution. When the core finishes all the running thread blocks, the core is preempted and can be assigned to a new kernel.

To further reduce preemption latency, Park et al. [178] introduced core flushing which drops an execution of a thread block without context saving and re-executes the dropped thread block from the beginning when it is relaunched. To ensure correctness of core flushing, a GPU kernel must be idempotent (i.e., it produces the same result regardless of the number of times it is executed). They also proposed a collaborative preemption approach, Chimera, that synergistically combines all the three previous mechanisms together. Chimera can meet the deadline of a given preemption latency with minimal throughput overheads. To achieve this, when a preemption request is received, Chimera checks the execution progress of the running thread block. If thread block is at the begging, middle or the end of execution, then Chimera employs flushing, context switching and draining mechanism respectively. Selecting the best policy based on the execution progress significantly reduces the preemption overhead.

5.3. *Exploiting scalar and value similarity opportunities*

Previous works observed that many GPGPU workloads have scalar instructions in which computation is identical across multiple threads within the same warp instruction (i.e., operands are identical for all the threads in a warp). On average, 38% of static SIMD instructions are detected by the compiler as scalar opportunities [29]. These scalar opportunities can be exploited by saving the scalar vector in only one scalar register, leading to power saving. In addition, these scalar instructions can be executed only once and eliminate computation redundancy, resulting in higher instruction throughput and performance. Thus, modern GPU microarchitecture, like AMD's GCN [10], leverages these scalar opportunities by statically detecting scalar instructions and executing them on a separate scalar unit attached with each GPU core. Further, other previous works observed that many GPGPU workloads may also contain affine vector instructions. A vector is defined as an affine, when the vector contains a consecutive strided values, i.e., the vector values can be represented as $V(i) = b + i * s$, where b is the base, s is the stride and i is the thread index. These affine vector instructions can be similarly exploited as scalar instructions by saving them in only two registers (to save base and stride values) and executed them on a separate scalar special unit [113]. This results in improving GPU performance and energy efficiency. Table 8 summarizes and compares between previous works that aim to exploit scalar and affine opportunities. We classify these works based on the following metrics:

- Detection method: (a) dynamic detection using hardware mechanism [36,56,113,246,250,262] or (b) static detection using compiler support [29,125,217] or (c) programmable scalar unit (i.e., the scalar unit can execute a separate instruction stream that is either generated automatically by the compiler or manually developed by expert developers) [257].
- Execution approach: the scalar instructions are executed on (a) a dedicated scalar unit located in each GPU core [10,56, 113] or (b) using one lane of the SIMD unit lanes and power-off the other lanes [250,262] or (c) a Temporal-SIMT fashion, where the vector and scalar instruction are executed on a single scalar lane [99,125].
- Register file: the scalar operands can be saved at (a) the same vector registers and a token-based bit is used to identify whether a scalar or vector operand is saved in the register [36] or (b) a separate scalar register file is used to save scalar operands [10,113].
- Fetch-decode-issue stage: (a) the scalar instruction can share the fetch, decode and issue units with other vector instructions [10,113] or (b) they can use separate fetch, decode and issue units that execute in parallel with other vector units [257].
- Exploited instructions type: the proposed approach may detect and leverage (a) Intra-warp scalar opportunities (i.e., threads from the same warp perform the same computation) [10,113] or (b) Inter-warp scalar opportunities (i.e., threads from different warps perform the same computation) [250,262] or (c) affine instructions opportunities [36,113, 262].

5.4. *Improving execution pipelining*

Other previous works have been proposed to improve GPU pipeline execution efficiency. Gilani et al. [57] observed that many GPGPU applications do not have enough active threads that are ready to issue instructions and hide short read-after-write (RAW) dependencies caused by deep execution pipeline stages. Thus, they proposed a low-power forwarded network that can considerably improve the performance of many compute-intensive GPGPU applications. Another way to improve the execution pipeline efficiency was proposed by Gilani et al. [56]. They observed that 16 bits are sufficient for accurate representation of operands for many GPGPU instructions. This can be exploited to improve GPU performance by splitting the existing 32-bit datapath into two 16-bit datapath slices. As a result, the GPU instruction throughput can be increased by issuing dual 16-bit instructions from two different warps in parallel using the sliced 32-bit datapath. Kim et al. [112] presented a pre-execution approach for improving GPU latency hiding and performance by employing run-ahead out-of-order execution [158]. In their approach, when a warp stalls for a long-latency operation such as off-chip memory accesses,

Table 8
GPU scalarization works.

	Detection	Execution	Register file	Fetch-Decode-Issue stages	Exploited Instructions
Chen et al. [29]	static	separate/shared	shared	shared	Intra-warp scalar
Lee et al. [125], Keckler et al. [99]	static	Temporal-SIMT	shared	shared	Intra-warp scalar
Yang et al. [257]	static/ programmable	separate	separate	separate	Intra-warp scalar
Collange et al. [36]	dynamic	shared	shared token-based register	shared	Intra-warp scalar/affine
Gilani et al. [56]	dynamic	separate	separate	shared	Intra-warp scalar
Xiang et al. [250]	dynamic	shared	separate	shared	Intra- and Inter-warp scalar
Kim et al. [113]	dynamic	separate	separate	shared	Intra-warp scalar/affine
Yilmazer et al. [262]	Intra-warp static, Inter-warp dynamic	shared	shared	shared	Intra- and Inter-warp scalar
AMD's GCN [10]	static	separate	separate	shared	Intra-warp scalar

it continues to fetch and pre-execute successive instructions that are not on the long latency dependence chain resulting in hiding processing delay of operations and performance improvement.

6. Enhancing GPGPU programmability

GPGPU programming is hard and complex. Prior work have explored new techniques to enhance GPGPU programmability. In fact, most of these works were about addressing the same challenges that were found in conventional CPU multi-core programming (e.g. cache coherence, memory consistency, synchronization and transactional memory). However, this is not a trivial task for GPUs, since GPUs run thousands of threads concurrently, whereas multi-core CPUs run 4–16 threads. Building a scalable hardware to enhance GPGPU programmability is a key challenge. It is worth mentioning that enhancing GPGPU programmability is an important feature for future GPUs in order to simplify GPGPU programming and broaden the scope of applications that can benefit from GPU acceleration. More importantly, enhancing programmability strongly encourages High-performance computing (HPC) and cloud computing communities to fully rely on GPU as a compelling general purpose accelerator for emerging data-intensive workloads in competition with other accelerator platforms, such as in-field programmable gate array (FPGA). Thus, there is no wonder that three previous works which aimed to augment GPU with transactional memory [50], cache coherence [209] and virtual memory [183] have been selected for IEEE Micro top picks[4] in 2011, 2013 and 2014 respectively. In all, we summarize these works that aim to improve GPGPU programmability in Table 9. In the following subsections, we discuss these works in more detail.

[4] Every year, IEEE Micro top picks community selects the most 10–13 significant research papers in computer architecture based on novelty and potential for long-term impact.

6.1. *Coherence and consistency model*

Current GPUs lack hardware cache coherence and require disabling of private L1 caches or employing software-based bulk coherence decisions (i.e., flush/invalidate all private L1 caches at synchronization points) if an application needs coherent memory view. Coherence largely simplifies supporting well-defined consistency and memory models for high-level languages on GPUs. However, applying naive CPU-like read-for-ownership protocol (e.g. MOESI) will incur significant overheads. GPU runs 1000s threads concurrently which requires high coherence traffic overheads and impractical amount of storage. To enable a scalable and practical GPU cache coherence, Singh et al. [209] proposed a time-based coherence framework for GPUs, named Temporal Coherence (TC). In contrast to conventional coherence protocols that rely on explicit messages to invalidate and maintain cache coherence, TC uses globally synchronized counters to self-invalidate cache blocks and maintain coherence without explicit messages. Synchronized counters approach not only enables cache coherence, but also eliminates all coherence traffic and protocol races. TC significantly improves the performance of GPU applications with inter-CTA communication over disabling private L1 caches.

As mentioned earlier, current GPUs employ software-based bulk coherence decisions. However, bulk coherence actions negatively affect performance. To address this problem, Gaster et al. [52], Orr et al. [172] proposed scoped synchronization that follows heterogeneous-race-free (HRF) model. Scopes take advantage of the GPUs hierarchical memory model to limit the cost of bulk coherence actions. In this model, threads executing on the same SM can communicate through the L1 cache without issuing any cache flushes or invalidates, whereas threads from differents SMs need to communicate through the higher level caches. However, scoped HRF complicates GPU programming and does not use caches effectively to optimize dynamic sharing patterns like work stealing [9]. To this end, Sinclair et al. [206] showed that it is possible to have the simple data-race-free (DRF) model without scopes, meanwhile achieving the performance benefits of scoped synchronization, by applying the DeNovo coherence protocol to GPUs.

Table 9
GPGPU programmability works.

Coherence and Consistency Model	• Timestamp-based coherence [209] • Hybrid hardware-software coherence [100,116] • Scoped heterogeneous-race-free (HRF) model [52,67,78,172] • Non-scoped data-race-free (DRF) model [9,206,207] • Strong consistency model [68,188,208,213] • Evaluating current GPU memory model [8]
Hardware Synchronization and Atomics	[43,45,263]
Transactional Memory (TM)	Hardware TM [30,31,47,50], Software TM [26,256]
Determinism, Debugging and Data Races Detection	Deterministic GPGPU [95], Data race detection [76]
Memory Management and Exception Support	Exception support [104,153,219], Hardware-based memory management [109,111,273], Hardware-based virtual memory support [19,183,183,185,185]

Hechtman and Sorin [68] analyze the issues of hardware consistency models in the context of GPGPUs. They compared various hardware consistency models for GPGPUs, including sequential consistency, total store order and relaxed memory model, in terms of performance, energy-efficiency, hardware complexity, and programmability. Surprisingly, they found out that employing hardware consistency models has negligible impact on the performance of GPGPUs workloads. More importantly, GPGPUs can be strongly ordered and often incur only little performance loss compared to weaker consistency models. Furthermore, stronger models enable simpler and more energy-efficient hardware implementations and are likely easier for programmers to reason about. However, the major drawbacks for GPU strong consistency model works is that they omit dynamic sharing workloads in their evaulation, and the proposed approachs are not scalable for cuurent massive GPUs[9].

6.2. Transactional memory

Intra-block GPU communication between different threads that belong to the same thread block is provided via Intra-core scratchpad memory. Recent GPUs also support global inter-block communication, in which threads from different thread blocks can communicate with each other, through global atomic operations. These atomic operations can be used to build software synchronization primitives, such as fine-grain locks, that can simplify the programming and ensure execution correctness for many emerging GPGPU workloads. However, lock-based synchronization is prone to deadlock and may cause incorrect execution behavior, especially for GPU that executes hundreds of threads in parallel. To enable efficient deadlock-free inter-block communication, Fung et al. [50] proposed KILO TM, a scalable hardware Transactional Memory (TM) system for GPGPUs. KILO TM does not rely on cache coherence nor global atomic operations. Instead, it detects conflicts via a fine-grain value-based approach that supports thousands of concurrent transactions and requires negligible storage overhead. Further, it employs bloom filter mechanism to speculate conflict detection, resulting in increased parallelism in transactions commit and performance improvement.

6.3. Deterministic GPU

In nondeterminism environment, running the same multi-threaded program multiple times with the same input may produce different outputs. This behavior hinders programmer's ability to test and debug their GPGPU applications. In fact, as previously mentioned, GPU runs thousands of threads which makes achieving low-cost scalable deterministic GPU a key challenge. To that end, Jooybar et al. [95] proposed deterministic GPU, GPUDet, a scalable hardware mechanism that provides determinism in GPU architectures in order to ease debugging and testing of GPU applications. GPUDet enables a broader class of software applications that can benefit from GPU acceleration. The key ideas behind GPUDet are two-fold. First, it leverages the inherent determinism of the SIMD hardware in GPUs to provide determinism within a wavefront at no cost. Second, GPUDet exploits the Z-Buffer Unit, an existing GPU hardware unit for graphics rendering, to allow parallel out-of-order memory writes to produce a deterministic output.

6.4. Memory management

Kim et al. [109] proposed GPUdmm, a high-performance dynamic memory management for GPU architecture. GPUdmm enables dynamic memory management for discrete GPU environments by using GPU memory as a cache of CPU memory with on-demand CPU–GPU data transfers. The benefits of GPUdmm are three-fold. First, GPUdmm simplifies the GPGPU programming by relieving the programmer of CPU–GPU memory management burden. Second, it provides programmers a view of the CPU memory-sized programming space. Third, GPUdmm effectively overlaps GPU executions and CPU–GPU data transfers.

Pichai et al. [183] examined address translation in CPU/GPU architecture to push toward fully-unified virtual address spaces in heterogeneous platforms. This simplifies programming models and reduces the burden on programmers to manage memory. Therefore, they explored the design space of GPU cache-parallel address translation. However, they found that adding CPU-style address translation at the L1-level of the GPUs can degrade performance. They showed that simple GPU-aware modifications to conventional translation lookaside buffers (TLBs) and hardware page table walkers (PTWs) can significantly reduce performance overheads associated with address translation. They also showed employing address translation with two previous proposals for improving GPU performance, TBC and CCWS, that we have discussed in Sections 3 and 4.1 respectively, imposes noticeable overhead. However, augmenting CCWS and TBC with TLB-awareness and a few simple adjustments can recover most of this lost performance and move address translation overheads into a range considered acceptable in the CPU world.

7. CPU–GPU heterogeneous architecture

In order to amortize the increasing die area, recent years have seen a noticeable trend from the industry to integrate CPU and GPU cores on the same chip, as it can be seen in Intel's Haswell [82], AMD's accelerated processing units (APU), like AMD Fusion Kaveri [11], and NVIDIA's Denver project [162]. In these architectures, the concurrent CPU and GPU applications will share most of the on-chip resources, such as memory controller, interconnection network and last level cache). However, GPUs run thousands of active threads concurrently, while CPUs run between 4–16 threads. This leads to more frequent memory requests generated by GPU cores. Therefore, a fairness-aware scheme is required to manage the concurrent applications and avoid GPU applications to monopolize the available resources. Further, when we integrate CPU and GPU on chip, some sort of programmability is also required to improve data sharing between CPU and GPU cores (e.g. cache coherence and unified virtual memory space). Enhancing programmability and communication between CPU and GPU will lead to broaden the space of applications that may benefit from GPU as a general purpose accelerator. For the purpose of this study, we classify the works that aim to improve CPU–GPU integration into the different categories summarized in Table 10. In the following subsections, we discuss these works in more detail.

7.1. Impacts of CPU–GPU integration

As we stated earlier, in order to take the full advantage of integrated CPU–GPU heterogeneous architecture, programmers need to rewrite their legacy code such that data-parallel portion is offloaded to GPU while the remaining part is executed on CPU. As the GPU becomes more programmable and CPU–GPU communication cost is reduced, more applications will be adopted to GPU acceleration. Arora et al. [15] demonstrated that the remaining serial code that the CPU will be expected in an integrated CPU–GPU environment is entirely different than the code it has been optimized for over the past many CPU generations. Thus, they studied the characteristics of this new code that is supposed to drive future CPU design and architecture. In particular, they showed that the remaining CPU code tends to have lower instruction-level parallelism (ILP), more complex load/store operations to prefetch and more difficult branch prediction. Further, the serial code will not benefit significantly from SIMD instructions or increasing the number of CPU cores, owing to the limited availability of thread-level parallelism (TLP) and data-level parallelism (DLP) that will be already captured and exploited by the GPU instead.

7.2. CPU–GPU programmability

Power et al. [184] presented a framework for providing directory-based hardware coherence between CPU and GPU cores to ease programming and enable fine-grained sharing. However, they experimentally observed that employing conventional block-level directory-based cache coherence significantly hinders the performance due to the tremendous coherence traffic generated by the massive-multithreaded GPU. Therefore, in order to mitigate the coherence bandwidth effects of GPU memory requests, they replace the fine-grained 64B-block-level directory with a coarse-grained 1KB-region-level directory. Increasing the directory granularity basis significantly reduces the coherence traffic and improves performance. This is due to the fact that many GPGPU applications exhibit high spatial locality in the memory access stream, such as streaming applications. Thus, most requests will not need to access the region directory because the permissions for the requested region have been obtained already by prior requests.

7.3. Exploiting heterogeneity

Yang et al. [259] proposed COMPASS, a compute GPU-assisted data prefetching scheme, to leverage the underutilized GPU resource for improving CPU single-threaded performance on a CPU–GPU integrated system. COMPASS uses idle GPU core resources to act as data prefetchers for CPU execution and successfully improve the memory performance of single-thread applications. Their proposed scheme requires very lightweight architectural support. Similarly, Woo and Lee [247] proposed to collaboratively utilize CPU resources to act as programmable data prefetchers for GPGPU applications. In this scheme, a novel compiler algorithms are developed to automatically create CPU programs from GPU kernels. These CPU programs run ahead of GPU threads to prefetch the required data into the shared last level cache for the GPU, resulting in higher cache hit and better performance for GPGPU applications.

7.4. Shared resources management

As multiple CPU and GPU cores integrated together on the same die chip, they eventually share the memory subsystem resources, including last level cache, on-chip network and main memory. As has been mentioned, such sharing mechanism can lead to low system performance and starvation of CPU cores, since GPU cores generate a large traffic of pending memory requests that can heavily interfere with latency-sensitive CPU memory requests and monopolize the memory subsystem resources. Two kinds of approaches have been explored to mitigate interference: application-aware resource management and throttling-based management. Ausavarungnirun et al. [17], Lee and Kim [123] and Lee et al. [127] equip shared resources, such as memory controller, cache and interconnection network respectively, with the ability to detect interference between CPU and GPU applications and prevent unfair CPU application slowdowns. On the other hand, Kayiran et al. [98] preclude interference by throttling the number of active threads running at GPU cores, and consequently controlling the high rate of memory requests issued by GPU. In the following sections, we discuss in detail the mechanism used by each approach.

In order to mitigate CPU–GPU inter-application interference at main memory, Ausavarungnirun et al. [17] proposed Staged Memory Scheduling (SMS), a decentralized architecture for application-aware memory scheduling in the context of integrated CPU–GPU systems. SMS decouples memory controller into three significantly simpler stages that together improve system performance and fairness. The first stage of SMS groups requests based on row-buffer locality. At the second stage, SMS ensures fairness between CPU and GPU memory requests by applying CPU-biased shortest job first scheduling policy or GPU-biased round robin scheduling policy. A dynamically configurable parameter is used to select between the two policies based on the system's needs. The last stage consists of simple per-bank FIFO queue to issue low-level memory commands.

To avoid GPU monopolizing last level cache, Lee and Kim [123] proposed a thread-level parallelism (TLP) aware cache management policy for CPU–GPU systems. As previously mentioned, GPGPU applications can hide some of the off-chip access latency by having a huge number of threads and continuing to switch to the next available threads. However, they observed that other GPGPU applications are cache-sensitive and are not able to tolerate long memory latency. Hence, they proposed a core-sampling technique, which applies a different cache management policy to each GPU core and regularly collects statistics on the performance of these cores to see how these polices affect GPU applications. A noticeable variance in performance of these cores implies that GPU performance is sensitive to the cache policy. As a result, GPU cache-insensitive applications are forced to bypass last level cache, while CPU and GPU cache-sensitive applications are allowed to use cache side by side. Moreover, they found out that GPU cores typically access caches much more frequently than CPU cores. To ensure fairness between CPU and cache-sensitive GPU workloads, they introduced cache block lifetime normalization approach, which

Table 10
CPU–GPU heterogeneous architecture works.

Studying the Efficacy and Impacts of CPU–GPU integration	• Performance and system impacts [37,69,245,270] • Impacts on CPU architecture: [15] • Impacts on memory subsystem [51,73,142,214,228]
CPU–GPU Programmability	• Cache coherence: [3,184] • Unified virtual memory: [170,183,185] • Synchronization and communication: [72,147]
Exploiting Heterogeneity	• CPU-Assisted GPU: data prefetching [259] • GPU-Assisted CPU: data prefetching [247], on-the-fly computation offloading [129,146,197]
Shared Resources Management	• Interconnection network: [127,128,264] • Last Level cache management: [123,150,187,271] • Main memory: [17,86,225,238] • Throttling-based technique: [98]

enforces a similar cache lifetime to both CPU and GPGPU applications and prevent GPGPU application to monopolize the shared cache.

Lee et al. [127] tackled the problem of resource-sharing at the on-chip network in CPU–GPU heterogeneous systems. They proposed a feedback-directed virtual channel partitioning (VCP) organization that fairly partitions the available network bandwidth among CPU and GPU applications. To avoid GPU interference with CPU applications, VCP dedicates separate injection queues and virtual channels to CPU and GPU applications. Then, VCP samples different virtual channels partitioning over different sampling periods and adaptively chooses the best performing configuration that mitigates the interference and balance on-chip network bandwidth.

Kayiran et al. [98] showed that, in CPU–GPU heterogeneous architecture, because of the high thread-level parallelism (TLP) GPU cores execute, GPU applications may monopolize the shared memory subsystem resources, such as interconnection network and memory controller. That is, the higher TLP GPU core runs, the more memory requests GPU generates. Thus, they proposed GPU concurrency management that dynamically throttles/boosts TLP (i.e., number of active warps) of GPU cores in order to minimize shared resources interference between CPU and GPU. They proposed two different schemes to control the interference. The first scheme aims to reduce GPU concurrency, leading to some performance loss in some GPU applications, in an attempt to mitigate interference and improve CPU performance. The second proposed scheme tries to improve both CPU and GPU performance by balancing the sharing between the running applications.

8. Future directions

GPUs continue to evolve as new applications arise or to make executing current applications more efficient in terms of power and performance. In this section, we will look at some of the advances in GPU research that started now and are expected to continue and evolve in the future.

8.1. GPUs and machine learning

With the increased popularity of applications dependent on AI, more specifically machine learning, GPUs take more important role. One subfield of machine learning that is gaining huge popularity in many domains is deep learning. Deep learning depends on neural networks with large number of layers. The inner working of deep neural networks involves updating the weights of interconnections among huge number of neurons. This operation can be stated as matrix multiplication, which is a very GPU friendly operation. As neural network becomes *deeper*, that is, with many more layers, GPUs become more efficient due to the increased data-parallelism. Updating the weights is very compute intensive and is the learning part of a two phase operation: learning and inference. GPU is the most efficient one in the learning part due to the large amount of data parallelism, even though lately new architectures are proposed for learning part, such as Google's third generation Tensor Processing Units (TPU) [96]. It is also good in inference but has some competitors like Google's TPUs,[5] Microsoft Catapult project based on FPGA [186] (and more recently project brainwave), many startups that design custom silicon for inference, and, of course, traditional multicore processors.

Recently, NVIDIA introduced tensor cores in its newest GPU architecture Volta [168] to speedup both inference and learning phases. Tensor cores are included in SMs of the new GPU and are programmable matrix-multiply-and-accumulate units. Each tensor core can do: $R = A*B+C$ where A, B, C, and the result R are all 4×4 matrices. In the V100 GPU, for example, each SM contains 8 tensor cores. This increased tremendously the throughput of both learning and inference. Many machine learning schemes do not require high-precision floating point operations, unlike many scientific simulations. This is why GPUs provide half-precision, 16-bit, floating point support.

Recent works have been proposed to improve deep learning execution on GPUs. Diamos et al. [41] exploited the large size of on-chip register files of modern GPU to cache the Recurrent Neural Networks (RNN) parameters and reuse them over multiple timesteps during training. Rhu et al. [192] proposed a runtime memory manager that virtualizes the memory usage of DNNs such that it can utilize both GPU and CPU memory transparently for training larger DNNs without burdening the programmer. Song et al. [210], Yu et al. [267], Hill et al. [75] proposed novel techniques to improve sparse deep learning inference on mobile GPUs.

There are many interesting research questions here. What are other customized processing units or storages which can be added to GPU and accelerate other emerging machine learning networks, such as Sparsity in Convolution Networks inference [65,175], Long

[5] First generation TPU was designed for inference. But subsequent generations are targeting training too. However, comparing it to GPUs is a bit tricky. For example, the 16-bit floating point formats for TPU and GPU are different.

short-term memory (LSTM) [64], and Generative Adversarial Networks (GAN) [212]?

8.2. GPUs and approximate computing

Approximate computing is trading-off precision for efficiency [140,155,198]. Efficiency here can be higher performance, lower power, or both. Not all applications require double precision floating point, for example. Maxwell GPU, as an example, sacrifices many double precision units for the sake of power and performance. Approximate computing techniques can be software-based or hardware-based.

At the software-side, in GPUs, there are techniques like using fixed-point or low-precision floating point operations (fp16). At the hardware side, inexact hardware is used in floating-point units (FPU) and special function units (SFU) because they are consuming large amount of power and affect performance. For instance, Li et al. [140] proposed a software-based transparent SFU-driven approximate acceleration on GPUs.

The above techniques, targeting GPUs, can be combined with general approximate computing methods like reducing the number of iterations to calculate an average number, at the expense of less precise solutions.

A future direction is to determine needed precision dynamically. For example, reduce the precision dynamically when power consumption reaches a specific threshold, in a way similar to dynamic voltage and frequency scaling. Another challenge is when multiple kernels are being executed on the device simultaneously and each one needs a different solution quality. This requires a smart scheduling to provide the needed quality-of-service within the needed power or performance budget.

Because we are talking about multiple kernels, how about multi-tenancy in GPUs?

8.3. GPUs and multi-tenancy

GPUs are now an integral part of datacenters and supercomputers. The main reason for that is the excellent performance per watt GPUs have. In order to amortize the power consumed by GPUs, high-utilization is a must. Besides the traditional tricks in coding, for example streams in CUDA, scheduling different applications on GPU through multi-tenancy is proposed [202]. In that work, the authors propose to increase GPU utilization by servicing requests from multiple tenants. The proposed scheme has several goals: fairness to the different applications, meeting real-time deadlines, and high throughput for the GPUs. The main idea is to have a two-level software scheduling. The highest level distributes requests to different GPUs to ensure load-balancing. The second level ensures the efficient multiplexing of kernels assigned to a GPU. Scalability of multi-tenancy to very large number is then a useful research direction as datacenters scale-up to serve more and more requests. Further, as GPU is used by multiple tenants, security concerns are raised. Naghibijouybari et al. [159] showed various covert channel attacks are possible in modern GPUs. Building secure GPU that ensures a complete isolation of the running applications without sacrificing throughput is another important area of research.

8.4. GPUs and OS

The traditional way OS treats GPU is like an I/O device. This was acceptable in the past when GPUs were just as their name says: *graphics* processing units. Now, as GPUs are more general purpose, this is becoming very inefficient. Treating GPUs as I/O devices and not first-rate entity by the OS is that GPUs cannot access the file system of a disk for example. It has to be done through the host, which wastes a lot of time and bandwidth.

Table 11
Key characteristics of recent NVIDIA GPUs [168].

	Fermi	Kepler	Maxwell	Pascal	Volta
#Cores	16	15	24	56	80
GFLOPS	1.3	5	6.8	10.6	15.7
Mem BW (GB/sec)	177	288	288	720	920
Mem cap (GB)	3	6	12	16	16
L2 cache (KB)	768	1536	3072	4096	6144
Transistors (B)	3	7.1	8.0	15.3	21.1
Tech. node (nm)	40	28	28	16	12
Chip size (mm2)	529	551	601	610	815

Authors in [205,227] present a compelling case for allowing GPUs to use system calls. This way, GPUs will have access to many features that were only available to the host CPUs. There are many details to be resolved. However, this direction is certainly a needed one.

Another important step in OS and GPU relationship is preemption. Starting from NVIDIA Pascal GPU, the OS can preempt the compute task of the GPU at the instruction granularity instead of the traditional thread-block granularity in older versions (e.g. Kepler, Maxwell) [167].

There are many research directions that are needed to enhance how the OS interacts with GPUs to achieve better and more efficient overall performance.

8.5. Future GPUs and performance scalability

GPU applications with abundant parallelism available in exascale applications (such as those found in deep learning and exascale computing [229,232]) will see continuous performance improvement as the computational and memory capabilities in the GPU increase. Table 11 shows the scalability trend over Nvidia GPU generations [168]. As shown in table, as we scale down in technology node, number of cores keeps to increase over generations. The recent Volta architecture contains 80 GPU cores in one gigantic chip that has 21 billion transistors. However, in order to maintain performance scalability, building a larger GPU (beyond Volta) with dozens of GPU cores in one monolithic chip may not be possible due to low manufacture yield and high cost of building huge chips at small technology nodes [16,229].

To address this issue, previous works [16,229] proposed partitioning large monolithic GPUs into easily manufacturable chiplets (a.k.a. GPU Chip Module) connected via high-speed, in-package communication network. However, disintegration of GPU into chiplets may lead to performance loss due to Non-Uniform Memory Access (NUMA) effect [16]. Addressing NMUA effect in chiplet-based GPU architecture is challenging, especially in irregular workloads wherein data accesses are randomly scattered over chiplets' memory modules [16,154,229].

Another key observation from Table 11 is that memory capacity and speed do not scale at the same pace as performance. For example, performance capability (GFLOPS) has increased 50% from Pascal to Volta, whereas memory speed has only increased 25% and memory capacity remains the same. As this trend continues to grow in the future, the memory capacity and bandwidth will become a major bottleneck for many workloads. High bandwidth memory technologies [12] and employing heterogeneous memory [229] seem to be promising solutions to improve memory scalability; however, more research are still required to reduce the gap between GPU core and memory performance.

8.6. *More tightly-coupled CPU+GPU integration*

There have been a growing interest in improving the integration of CPU and GPU. The aim of these works is to more tightly couple CPU and GPU efficiently and use them in tandem as produce-consumer model. Recently, OpenCL and CUDA have introduced new features such as, shared virtual memory, memory coherence, and system-wide atomics [58]. This opens new collaboration patterns to allow CPU and GPU to communicate at fine-granularity and allows more efficient execution flow and overlapping. However, there is still a performance gap between the coherence and atomic within GPU threads and between CPU and GPU threads. More research are required to reduce this gap and improve CPU and GPU communication.

9. Conclusion

Recent years have been witnessing the emergence of using GPUs for general purpose computing due to their massive computational power and energy efficiency. The ultimate goal of this growing interest is to make GPUs a real general purpose many-core accelerator that can be used side-by-side with CPU in order to improve the performance of compute-intensive workloads and reduce energy and power consumption. That is, to efficiently utilize the emerging CPU–GPU heterogeneous architecture, we need to execute the latency-sensitive portions of our programs on CPUs and launch data-parallel portions on GPUs. To that end, many issues need be addressed to rely on GPGPUs as a compelling general purpose accelerator for the next power-limited big-data era. In this paper, we survey architecture approaches which aim to (1) improving the GPGPUs performance for emergence irregular workloads, (2) alleviating resource contention and efficiently utilizing the memory subsystem bandwidth, (3) exploiting scalar opportunities, increasing the available parallelism and enabling concurrency, (4) enhancing GPGPU programmability, and (5) improving the CPU–GPU integration. We strongly believe that this survey paper will be insightful to the researcher into working on improving GPU architecture for general purpose computing.

References

[1] Amir Ali Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, Daniel Wong, Wireframe: supporting data-dependent parallelism through dependency graph execution in gpus, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 600–611.
[2] Jacob T. Adriaens, Katherine Compton, Nam Sung. Kim, Michael J. Schulte, The case for gpgpu spatial multitasking, in: High Performance Computer Architecture, HPCA, 2012 IEEE 18th International Symposium on, IEEE, 2012, pp. 1–12.
[3] N. Agarwal, D. Nellans, E. Ebrahimi, T.F. Wenisch, J. Danskin, S.W. Keckler, Selective GPU caches to eliminate CPU–GPU HW cache coherence, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 494–506.
[4] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W. Keckler, Thomas F. Wenisch, Unlocking bandwidth for GPUs in CC-NUMA systems, in: High Performance Computer Architecture, HPCA, 2015 IEEE 21st International Symposium on, 2015.
[5] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, S. Keckler, Page placement strategies for GPUs within heterogeneous memory systems, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2015.
[6] P. Aguilera, K. Morrow, Nam Sung Kim, Qos-aware dynamic resource allocation for spatial-multitasking GPUs, in: Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific, 2014.
[7] P. Aguilera, K. Morrow, Nam Sung Kim, Fair share: Allocation of GPU resources for both performance and fairness, in: Computer Design, ICCD, 2014 IEEE 32nd International Conference on, IEEE, 2014.
[8] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, John Wickerson, GPU concurrency: weak behaviours and programming assumptions, in: Architectural Support for Programming Languages and Operating Systems, ASPLOS'15, 2015 20th International Conference on, 2015.
[9] J. Alsop, M.S. Orr, B.M. Beckmann, D.A. Wood, Lazy release consistency for GPUs, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–14.
[10] AMD, Graphics Core Next Arhcitecure whitepaper, 2013, www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf.
[11] AMD, AMD fusion kaveri, 2014, www.amd.com/ComputeCores/.
[12] AMD, High bandwidth memory, 2015, https://www.amd.com/Documents/High-B{and}width-Memory-HBMpdf.
[13] AMD Evegreen, 2009, www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/.
[14] Jayvant Anantpur, R. Govindarajan, PRO: Progress aware GPU warp scheduling algorithm, in: Parallel and Distributed Processing Symposium, IPDPS, 2015 IEEE International, 2015.
[15] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott Baden, Dean Tullsen, Redefining the role of the CPU in the era of CPU–GPU integration, in: IEEE Micro, 2012.
[16] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, David Nellans, Mcm-gpu: Multi-chip-module gpus for continued performance scalability, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 320–332.
[17] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, Onur Mutlu, Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems, in: Proceedings of the 39th International Symposium on Computer Architecture, IEEE Press, 2012, pp. 416–427.
[18] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H. Loh, Chita R. Das, Mahmut T. Kandemir, Onur Mutlu, Exploiting inter-warp heterogeneity to improve GPGPU performance, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT, PACT '15, 2015.
[19] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, Onur Mutlu, Mosaic: a gpu memory manager with application-transparent support for multiple page sizes, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 136–150.
[20] Mihir Awatramani, Joseph Zambreno, Diane Rover, Increasing GPU throughput using kernel interleaved thread block scheduling, in: Computer Design, ICCD, 2013 IEEE 31st International Conference on, IEEE, 2013.
[21] Mihir Awatramani, Xian Zhu, Joseph Zambreno, Diane Rover, Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT, PACT '15, 2015.
[22] Ali Bakhoda, John Kim, Tor M. Aamodt, Throughput-effective on-chip networks for manycore accelerators, in: Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture, IEEE Computer Society, 2010, pp. pages 421–432.
[23] Ali Bakhoda, George L. Yuan, Wilson W.L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009.
[24] Peter Bakkum, Kevin Skadron, Accelerating SQL database operations on a GPU with CUDA, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, 2010, pp. 94–103.
[25] Nicolas Brunie, Sylvain Collange, Gregory Diamos, Simultaneous branch and warp interweaving for sustained GPU performance, in: Proceedings of the 39th International Symposium on Computer Architecture, IEEE Press, 2012, pp. 49–60.
[26] Daniel Cederman, Philippas Tsigas, Muhammad Tayyab Chaudhry, Towards a software transactional memory for graphics processors, in: Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization, EG PGV'10, 2010, pp. 121–129.
[27] Niladrish Chatterjee, Mike O'Connor, Gabriel H. Loh, Nuwan Jayasena, Rajeev Balasubramonian, Managing DRAM latency divergence in irregular GPGPU applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, 2014.
[28] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Lv Ji, Zhiying Wang, Wen mei Hwu, Adaptive cache management for energy-efficient GPU computing, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[29] Zhongliang Chen, David Kaeli, Norman Rubin, Characterizing scalar opportunities in GPGPU applications, in: Performance Analysis of Systems and Software, ISPASS, 2013 IEEE International Symposium on, IEEE, 2013, pp. 225–234.
[30] S. Chen, L. Peng, Improving GPU hardware transactional memory performance via conflict and contention reduction, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016.
[31] Sui Chen, Lu Peng, Samuel Irving, Accelerating gpu hardware transactional memory with snapshot isolation, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 282–294.

[32] Guoyang Chen, Xipeng Shen, Free launch: Optimizing GPU dynamic kernel launches through thread reuse, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[33] Xuhao Chen, Shengzhao Wu, Li-Wen Chang, Wei-Sheng Huang, Carl Pearson, Zhiying Wang, Wen-Mei W. Hwu, Adaptive cache bypass and insertion for many-core accelerators, in: Proceedings of International Workshop on Manycore Embedded Systems, MES '14, 2014.
[34] Hyojin Choi, Jaewoo Ahn, Wonyong Sung, Reducing off-chip memory traffic by selective cache management scheme in GPGPUs, in: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, 2012.
[35] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, Ng Andrew, Deep learning with COTS HPC systems, in: Proceedings of the 30th International Conference on Machine Learning, 2013.
[36] Sylvain Collange, David Defour, Yao Zhang, Dynamic detection of uniform and affine vectors in gpgpu computations, in: Euro-Par 2009–Parallel Processing Workshops, Springer, 2010, pp. 46–55.
[37] Mayank Daga, Ashwin M. Aji, Wu-chun Feng, On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing, in: Application Accelerators in High-Performance Computing, SAAHPC, 2011 Symposium on, IEEE, 2011, pp. 141–149.
[38] William J. Dally, The end of denial architecture and the rise of throughput computing, in: Keynote Speech at Desgin Automation Conference, 2010.
[39] Jeffrey R. Diamond, Donald S. Fussell, Stephen W. Keckler, Arbitrary modulus indexing, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[40] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, Sudhakar Yalamanchili, SIMD re-convergence at thread frontiers, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 477–488.
[41] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, Sanjeev Satheesh, Persistent rnns: Stashing recurrent weights on-chip, in: International Conference on Machine Learning, 2016.
[42] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, Alexander V. Veidenbaum, Improving cache management policies using dynamic reuse distances, in: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2012, pp. 389–400.
[43] A. ElTantawy, T.M. Aamodt, MIMD synchronization on SIMT architectures, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–14.
[44] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, Tor M. Aamodt, A scalable multi-path microarchitecture for efficient GPU control flow, in: High Performance Computer Architecture, HPCA, 2014 IEEE 20th International Symposium on, 2014.
[45] Sean Franey, Mikko Lipasti, Accelerating atomic operations on GPGPUs, in: Networks on Chip, NoCS, 2013 Seventh IEEE/ACM International Symposium on, IEEE, 2013, pp. 1–8.
[46] Wilson W.L. Fung, Tor M. Aamodt, Thread block compaction for efficient SIMT control flow, in: High Performance Computer Architecture, HPCA, 2011 IEEE 17th International Symposium on, IEEE, 2011, pp. 25–36.
[47] Wilson W.L. Fung, Tor M. Aamodt, Energy efficient GPU transactional memory via space–time optimizations, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 408–420.
[48] Wilson W.L. Fung, Ivan Sham, George Yuan, 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, IEEE Computer Society, 2007, pp. 407–420.
[49] Wilson W.L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt, Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware, ACM Trans. Archit. Code Optim. (2009).
[50] Wilson W.L. Fung, Inderpreet Singh, Andrew Brownsword, Tor M. Aamodt, Hardware transactional memory for GPU architectures, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 296–307.
[51] V. Garcia, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, A.J. Pena, Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications, in: 2016 IEEE International Symposium on Workload Characterization, IISWC, 2016, pp. 1–10.
[52] Benedict R. Gaster, Derek Hower, Lee Howes, HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models, ACM Trans. Archit. Code Optim. (2015).
[53] Benedict R. Gaster, Lee Howes, Can GPGPU programming be liberated from the data-parallel bottleneck? in: IEEE Computer, vol. 45, IEEE, 2012, pp. 42–52.
[54] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, Kevin Skadron, Energy-efficient mechanisms for managing thread context in throughput processors, in: ACM SIGARCH Computer Architecture News, vol. 39, ACM, 2011, pp. 235–246.
[55] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, William J. Dally, Unifying primary cache scratch and register file memories in a throughput processor, in: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2012, pp. 96–106.
[56] Syed Zohaib Gilani, Nam Sung Kim, Michael. J. Schulte, Power-efficient computing for compute-intensive gpgpu applications, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ACM, 2012, pp. 445–446.
[57] Syed Zohaib Gilani, Nam Sung Kim, Michael J. Schulte, Exploiting GPU peak-power and performance tradeoffs through reduced effective pipeline latency, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 74–85.
[58] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Pena, Wen-mei Hwu, Chai: collaborative heterogeneous applications for integrated-architectures, in: Performance Analysis of Systems and Software, ISPASS, 2017 IEEE International Symposium on, IEEE, 2017, pp. 43–54.
[59] Chris Gregg, Jonathan Dorn, Kim Hazelwood, Kevin Skadron, Fine-grained resource sharing for concurrent GPGPU kernels, in: Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, USENIX Association, 2012, 10–10.
[60] Chris Gregg, Kim Hazelwood, Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, in: Performance Analysis of Systems and Software, ISPASS, 2011 IEEE International Symposium on, IEEE, 2011, pp. 134–144.
[61] Beayna Grigorian, Glenn Reinman, Accelerating divergent applications on simd architectures using neural networks, ACM Trans. Archit. Code Optim. (2015).
[62] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, Uri C. Weiser, Many-core vs many-thread machines: Stay away from the valley, Comput. Archit. Lett. 8 (1) (2009) 25–28.
[63] I.E. Hajj, J. Gomez-Luna, C. Li, L.W. Chang, D. Milojicic, W.m. Hwu, KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–12.
[64] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al., Ese: Efficient speech recognition engine with sparse lstm on fpga, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017, pp. 75–84.
[65] Song. Han, Huizi Mao, William J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015, arXiv preprint arXiv:1510.00149.
[66] Bingsheng. He, Wenbin. Fang, Qiong. Luo, Naga.K. Govindaraju, Tuyong. Wang, Mars: a mapreduce framework on graphics processors, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008.
[67] Blake A. Hechtman, Shuai. Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, David A. Wood, QuickRelease: a throughput oriented approach to release consistency on GPUs, in: Proceedings of the 20th International Symposium on High Performance Computer Architecture, HPCA, 2014.
[68] Blake A. Hechtman, Daniel J. Sorin, Exploring memory consistency for massively-threaded throughput-oriented processors, in: Proceedings of the 40th International Symposium on Computer Architecture, ISCA, 2013.
[69] Blake A. Hechtman, Daniel J. Sorin, Evaluating cache coherent shared virtual memory for heterogeneous multicore chips, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2013.
[70] John L. Hennessy, David A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, fifth ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011, ISBN 012383872X, 9780123838728.
[71] Justin Hensley, Close to the Metal, in: Proceedings of SIGGRAPH, 2007, pp. 120–130.
[72] Joel Hestness, Stephen W. Keckler, David A. Wood, GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU–GPU processors, in: Workload Characterization, IISWC, 2015 IEEE International Symposium on, 2015.
[73] Joel Hestness, Stephen W. Keckler, David A. Wood, A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior, in: Workload Characterization, IISWC, 2014 IEEE International Symposium on, IEEE, 2014.
[74] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, Tor M. Aamodt, Characterizing and evaluating a key–value store application on heterogeneous CPU–GPU systems, in: Performance Analysis of Systems and Software, ISPASS, 2012 IEEE International Symposium on, IEEE, 2012, pp. 88–98.

[75] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, Jason Mars, Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 786–799.
[76] Anup Holey, Vineeth Mekkat, Antonia Zhai, HAccRG: Hardware-accelerated data race detection in GPUs, in: Parallel Processing, ICPP, 2013 42nd International Conference on, IEEE, 2013, pp. 60–69.
[77] Sunpyo Hong, Hyesoon Kim, An integrated gpu power and performance model, in: ACM SIGARCH Computer Architecture News, vol. 38, ACM, 2010, pp. 280–289.
[78] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, David A. Wood, Heterogeneous-race-free memory models, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, APLOS, ACM, 2014, pp. 427–440.
[79] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, S.W. Keckler, Transparent Offloading and Mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, 2016, pp. 204–216.
[80] Wen-mei W. Hwu, GPU Computing Gems Emerald Edition, Elsevier, 2011.
[81] Wen-mei W. Hwu, GPU Computing Gems Jade Edition, Morgan Kaufmann Publishers Inc., 2011.
[82] Intel, Intel haswell cpu, 2014, http://www.intel.com/content/www/us/en/processors/core/4th-gen-core-processor-family.html.
[83] Intel, Intel skylake, 2015, http://ark.intel.com/products/88195.
[84] James A. Jablin, Thomas B. Jablin, Onur Mutlu, Maurice Herlihy, Warp-aware trace scheduling for GPUs, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ACM, 2014, pp. 163–174.
[85] Hyunjun Jang, Jinchun Kim, Paul Gratz, Ki Hwan Yum, Eun Jung Kim, Bandwidth-efficient on-chip interconnect designs for GPGPUs, in: Design Automation Conference, DAC, 2015 52nd ACM/EDAC/IEEE, 2015.
[86] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, Nigel Paver, A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC, in: Proceedings of the 49th Annual Design Automation Conference, 850–855, 2012.
[87] Wenhao Jia, Kelly A. Shaw, Margaret Martonosi, MRPB: Memory request prioritization for massively parallel processors, in: High Performance Computer Architecture, HPCA, 2014 IEEE 20th International Symposium on, 2014.
[88] Wenhao Jia, Kelly A. Shaw, Margaret Martonosi, Characterizing and improving the use of demand-fetched caches in GPUs, in: Proceedings of the 26th ACM international conference on Supercomputing, ACM, 2012, pp. 15–24.
[89] Zhen Hang Jiang, Yunsi Fei, David Kaeli, A complete key recovery timing attack on a gpu, in: High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on, 2016.
[90] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, X. Liang, Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–12.
[91] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, Chita R. Das, Application-aware memory system for fair and efficient execution of concurrent gpgpu applications, in: Proceedings of Workshop on General Purpose Processing Using GPUs, ACM, 2014, p. 1.
[92] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, Chita R. Das, Anatomy of GPU memory system for multi-application execution, in: Proceedings of the 1st International Symposium on Memory Systems, 2015.
[93] Adwai Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, Chita R. Das, Orchestrated scheduling and prefetching for GPGPUs, in: Proceedings of the 40th International Symposium on Computer Architecture, ISCA, 2013.
[94] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, Chita R. Das, OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2013.
[95] Hadi Jooybar, Wilson W.L. Fung, Mike O'Connor, Joseph Devietti, Tor M. Aamodt, GPUDet: a deterministic GPU architecture, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, 2013, pp. 1–12.
[96] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara. Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, Doe Hyun Yoon, In-datacenter performance analysis of a tensor processing unit, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, ACM, New York, NY, USA, ISBN: 978-1-4503-4892-8, 2017, pp. 1–12, http://doi.acm.org/10.1145/3079856.3080246.
[97] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, Chita Ranjan Das, Neither more nor less: Optimizing thread-level parallelism for GPGPUs, in: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, IEEE Press, 2013, pp. 157–166.
[98] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, Chita R. Das, Managing GPU concurrency in heterogeneous architectures, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[99] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, David Glasco, GPUs and the future of parallel computing, Micro 31 (5) (2011) 7–17.
[100] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, Sanjay J. Patel, Cohesion: A hybrid memory model for accelerators, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, 2010.
[101] Farzad Khorasani, Rajiv Gupta, Laxmi N. Bhuyan, Efficient warp execution in presence of divergence with collaborative context collection, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[102] KHRONOS Group, The opencl specification version 2.0, 2014, https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.
[103] Emmett Kilgariff, Randima Fernando, The geforce 6 series GPU architecture, in: ACM SIGGRAPH 2005 Courses, ACM, 2005, p. 29.
[104] Hyesoon Kim, Supporting virtual memory in GPGPU without supporting precise exceptions, in: Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, ACM, 2012, pp. 70–71.
[105] Ji Kim, Christopher Batten, Accelerating irregular algorithms on GPGPUs using fine-grain hardware worklists, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[106] Gwangsun Kim, Jiyun Jeong, John Kim, Mark Stephenson, Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, 2016.
[107] Hanjoon Kim, John Kim, Woong Seo, Yeongon Cho, Soojung Ryu, Providing cost-effective on-chip network bandwidth in GPGPUs, in: Computer Design, ICCD, 2012 IEEE 30th International Conference on, 2012.
[108] Gwangsun Kim, Minseok Lee, Jiyun Jeong, John Kim, Multi-GPU system design with memory network, in: Proceedings of the 2010 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[109] Youngsok Kim, Jaewon Lee, Jae-Eon Jo, Jangwoo Kim, GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management, in: High Performance Computer Architecture, HPCA, 2014 IEEE 20th International Symposium on, 2014.
[110] Yonggon Kim, Hyunseok Lee, John Kim, An alternative memory access scheduling in manycore accelerators, in: Parallel Architectures and Compilation Techniques, PACT, 2011 International Conference on, IEEE, 2011, pp. 195–196.
[111] Y. Kim, J. Lee, D. Kim, J. Kim, ScaleGPU: GPU architecture for memory-unaware GPU programming, Comput. Archit. Lett. PP (99) (2013) 1–1a.
[112] K. Kim, S. Lee, M.K. Yoon, G. Koo, W.W. Ro, M. Annavaram, Warped-preexecution: A gpu pre-execution approach for improving latency hiding, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 163–175b.
[113] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, Christopher Batten, Microarchitectural mechanisms to exploit value structure in SIMT architectures, Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, 2013, 130–141b.
[114] D.B. Kirk, W.H. Wen-mei, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.
[115] John Kloosterman, Jonathan Beaumont, Michael Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, Scott Mahlke, WarpPool: Reducing congestion with inter-warp coalescing for throughput processors, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.

[116] Rakesh Komuravelli, Matthew D. Sinclair, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, Vikram S. Adve, Stash: Have your scratchpad and cache it too, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.
[117] Gunjae Koo, Yunho Oh, Won Woo Ro, Murali Annavaram, Access pattern-aware cache management for improving data utilization in gpu, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 307–319.
[118] Nagesh B. Lakshminarayana, Jaekyu Lee, Hyesoon Kim, Jinwoo Shin, Dram scheduling policy for gpgpu architectures based on a potential function, Comput. Archit. Lett. 11 (2) (2012) 33–36.
[119] Ahmad Lashgar, Amirali Baniasadi, Ahmad Khonsari, Dynamic warp resizing: Analysis and benefits in high-performance SIMT, in: Computer Design, ICCD, 2012 IEEE 30th International Conference on, IEEE, 2012, pp. 502–503.
[120] Ahmad Lashgar, Ahmad Khonsari, Amirali Baniasadi, HARP: Harnessing inactive threads in many-core processors, ACM Trans. Embedded Comput. Syst. 13 (3s) (2014) 114.
[121] Shin-Ying Lee, Akhil Arunkumar, Carole-Jean Wu, CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.
[122] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler, Krste Asanovic, Exploring the design space of SPMD divergence management on data-parallel architectures, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.
[123] Jaekyu Lee, Hyesoon Kim, TAP: A TLP-aware cache management policy for a CPU–GPU heterogeneous architecture, in: High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, IEEE, 2012, pp. 1–12.
[124] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, S. Ryu, iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 370–381.
[125] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, Krste Asanovic, Convergence and scalarization for data-parallel architectures, in: Code Generation and Optimization, CGO, 2013 IEEE/ACM International Symposium on, IEEE, 2013, pp. 1–11.
[126] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, Richard Vuduc, Many-thread aware prefetching mechanisms for GPGPU applications, in: Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on, IEEE, 2010, pp. 213–224.
[127] Jaekyu Lee, Si Li, Hyesoon Kim, Sudhakar Yalamanchili, Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures, ACM Trans. Des. Autom. Electron. Syst. 18 (4) (2013) 48d.
[128] Jaekyu Lee, Si Li, Hyesoon Kim, Sudhakar Yalamanchili, Design space exploration of on-chip ring interconnection for a CPU–GPU heterogeneous architecture, J. Parallel Distrib. Comput. 73 (12) (2013) 1525–1538.
[129] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, Scott Mahlke, Transparent CPU–GPU collaboration for data-parallel kernels on heterogeneous systems, in: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, IEEE Press, 2013, pp. 245–256.
[130] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, Soojung Ryu, Improving GPGPU resource utilization through alternative thread block scheduling, in: High Performance Computer Architecture, HPCA, 2014 IEEE 20th International Symposium on, 2014, 260–271b.
[131] Jeyull Lee, Dong-Gyun Woo, Heonhwan Kim, Mani Azimi, GREEN cache: Exploiting the disciplined memory model of opencl on GPUs, IEEE Trans. Comput. (2015).
[132] Shin-Ying Lee, Carole-Jean Wu, CAWS: Criticality-aware warp scheduling for GPGPU workloads, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, 2014.
[133] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, Vijay Janapa Reddi, Gpuwattch: enabling energy optimizations in gpgpus, in: ACM SIGARCH Computer Architecture News, vol. 41, ACM, 2013, pp. 487–498.
[134] Dongdong Li, Tor M. Aamodt, Inter-core loclaity aware memory scheudling, Comput. Archit. Lett. (2015).
[135] Ang Li, Gert-Jan van den Braak, Akash Kumar, Henk Corporaal, Adaptive and transparent cache bypassing for gpus, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 17.
[136] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, Shuaiwen Leon Song, Warp-consolidation: A novel execution model for gpus, 2018.
[137] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, Stephen W. Keckler, Priority-based cache allocation in throughput processors, in: High Performance Computer Architecture, HPCA, 2014 IEEE 21th International Symposium on, 2015.
[138] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, Huiyang Zhou, Locality-driven dynamic GPU cache bypassing, in: Proceedings of the 29th ACM on International Conference on Supercomputing, 2015.
[139] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, Henk Corporaal, Corporaal locality-aware cta clustering for modern gpus, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2017, pp. 297–311.
[140] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, Henk Corporaal, Sfu-driven transparent approximation acceleration on gpus, in: Proceedings of the 2016 International Conference on Supercomputing, ACM, 2016, p. 15.
[141] Ang Li, Wenfeng Zhao, Shuaiwen Leon Song, Bvf: enabling significant on-chip power savings via bit-value-favor for throughput processors, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017.
[142] Jieun Lim, Hyesoon Kim, Design space exploration of memory model for heterogeneous computing, in: Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, ACM, 2012, pp. 74–75.
[143] Erik Lindholm, Mark J. Kilgard, Henry Moreton, A user-programmable vertex engine, in: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, 2001, pp. 149–158.
[144] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, NVIDIA Tesla: A unified graphics and computing architecture, Micro (2008).
[145] Jiwei Liu, Jun Yang, Rami Melhem, SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[146] Chi-Keung Luk, Sunpyo Hong, Hyesoon Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: Microarchitecture, 2009 MICRO-42 42nd Annual IEEE/ACM International Symposium on, IEEE, 2009, pp. 45–55.
[147] Daniel Lustig, Margaret Martonosi, Reducing GPU offload latency via fine-grained CPU–GPU synchronization, in: High Performance Computer Architecture, HPCA, 2013 IEEE 19th International Symposium on, 2013, pp. 354–365.
[148] Roman Malits, Evgeny Bolotin, Avinoam Kolodny, Avi Mendelson, Exploring the limits of gpgpu scheduling in control flow bound applications, in: ACM Trans. Archit. Code Optim., vol. 8, 2012.
[149] Mengjie Mao, Jingtong Hu, Yiran Chen, Hai Li, VWS: a versatile warp scheduler for exploring diverse cache localities of GPGPU applications, in: Design Automation Conference, DAC, 2015 52nd ACM/EDAC/IEEE, IEEE, 2015, pp. 1–6.
[150] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, Antonia Zhai, Managing shared last-level cache in a heterogeneous multicore processor, in: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT, IEEE Press, 2013, pp. 225–234.
[151] Jiayuan Meng, Jeremy W. Sheaffer, Kevin Skadron, Robust simd: Dynamically adapted SIMD width and multi-threading depth, in: Parallel & Distributed Processing Symposium, IPDPS, 2012 IEEE 26th International, IEEE, 2012, pp. 107–118.
[152] Jiayuan Meng, David Tarjan, Kevin Skadron, Dynamic warp subdivision for integrated branch and memory divergence tolerance, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, 2010.
[153] Jaikrishnan. Menon, Marc De Kruijf, Karthikeyan Sankaralingam, iGPU: exception support and speculative execution on GPUs, in: Proceedings of the 39th International Symposium on Computer Architecture, ISCA, IEEE Press, 2012, pp. 72–83.
[154] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, David Nellans, Beyond the socket: Numa-aware gpus, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 123–135.
[155] Sparsh Mittal, A survey of techniques for approximate computing, ACM Comput. Surv. (ISSN: 0360-0300) 48 (4) (2016) 62:1–62:33, http://dx.doi.org/10.1145/2893356.
[156] Sparsh Mittal, Jeffrey S. Vetter, A survey of methods for analyzing and improving GPU energy efficiency, ACM Comput. Surv. (2014).
[157] Reza Mokhtari, Michael Stumm, S-L1: A software-based GPU L1 cache that outperforms the hardware L1 for data processing applications, in: Proceedings of the International Symposium on Memory Systems, MEMSYS '15, 2015.
[158] Onur Mutlu, Jared Stark, Chris Wilkerson, Yale N. Patt, Runahead execution: An alternative to very large instruction windows for out-of-order processors, in: High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, 2003, pp. 129–140.
[159] Hoda Naghibijouybari, Khaled N. Khasawneh, Nael Abu-Ghazaleh, Constructing and characterizing covert channels on gpgpus, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 354–366.
[160] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, Yale N. Patt, Improving GPU performance via large warps and two-level warp scheduling, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 308–317.

[161] John Nickolls, William J. Dally, The GPU computing era, Micro 30 (2) (2010) 56–69.
[162] NVIDIA, Project Denver, 2011, http://blogs.nvidia.com/blog/2014/08/11/tegra-k1-denver-64-bit-for-android/.
[163] NVIDIA, CUDA dynamic parallelism programming guide, 2014, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[164] NVIDIA, NVIDIA GeForce GTX 980: Featuring Maxwell The Most Advanced GPU Ever Made, 2014, http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINALPDF.
[165] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2014, http://nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[166] NVIDIA, CUDA C Programming guide v6.5, 2015, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[167] NVIDIA, Titanx, 2016, https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.
[168] NVIDIA TESLA V100 GPU architecture. White Paper, 2017, URL http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[169] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, William J. Dally, Fine-grained dram: energy-efficient dram for extreme bandwidth systems, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 41–54.
[170] Lena E. Olson, Jason Power, David A. Wood, Mark D. Hill, Border control: Sandboxing accelerators, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[171] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, David A. Wood, Fine-grain task aggregation and coordination on GPUs, in: Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA '14, 2014.
[172] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, David A. Wood, Synchronization using remote-scope promotion, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, 2015.
[173] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell, A survey of general-purpose computation on graphics hardware, in: Computer Graphics Forum, vol. 26, Wiley Online Library, 2007, pp. 80–113.
[174] Sreepathi Pai, Matthew J. Thazhuthaveetil, R. Govindarajan, Improving GPGPU concurrency with elastic kernels, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2013, pp. 407–418.
[175] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, William J. Dally, Scnn: An accelerator for compressed-sparse convolutional neural networks, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 27–40.
[176] Eunhyeok Park, Junwhan Ahn, Sungpack Hong, Sungjoo. Yoo, Sunggu Lee, Memory fast-forward: a low cost special function unit to enhance energy efficiency in GPU for big data processing, in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, 2015.
[177] Jason Jong Kyu Park, Yongjun Park, Scott Mahlke, ELF: maximizing memory-level parallelism for GPUs with coordinated warp and fetch scheduling, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, 2015.
[178] Jason Jong Kyu Park, Yongjun Park, Scott Mahlke, Chimera: Collaborative preemption for multitasking on a shared GPU, 2015.
[179] Jason Jong Kyu Park, Yongjun Park, Scott Mahlke, Dynamic resource management for efficient utilization of multitasking gpus, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2017, pp. 527–540.
[180] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Chita R. Das, Scheduling techniques for GPU architectures with processing-in-memory capabilities, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, 2016, pp. 31–44.
[181] Indrani Paul, Wei Huang, Manish Arora, Sudhakar Yalamanchili, Harmonia: Balancing compute and memory power in high-performance gpus, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.
[182] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T.C. Mowry, S.W. Keckler, A case for toggle-aware compression for GPU systems, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 188–200.
[183] Bharath Pichai, Lisa Hsu, Abhishek Bhattacharjee, Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces, in: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, ACM, 2014, pp. 743–758.
[184] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, David A. Wood, Wood heterogeneous system coherence for integrated CPU–GPU systems, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 457–467.
[185] Jason Power, M. Hill, D. Wood, Supporting x86-64 address translation for 100s of GPU lanes, in: Proceedings of the 20th International Symposium on High Performance Computer Architecture, 2014.
[186] Andrew Putnam, Fpgas in the datacenter: Combining the worlds of hardware and software development, in: Proceedings of the on Great Lakes Symposium on VLSI 2017, GLSVLSI '17, ACM, New York, NY, USA, ISBN: 978-1-4503-4972-7, 2017, pp. 5–5, http://doi.acm.org/10.1145/3060403.3066860.
[187] Siddharth Rai, Mainak Chaudhuri, Exploiting dynamic reuse probability to manage shared last-level caches in CPU–GPU heterogeneous processors, in: Proceedings of the 2016 International Conference on Supercomputing, 2016, pp. 3:1–3:14.
[188] Xiaowei Ren, Mieszko Lis, Efficient sequential consistency in gpus via relativistic cache coherence, in: High Performance Computer Architecture, HPCA, 2017 IEEE International Symposium on, IEEE, 2017, pp. 625–636.
[189] Minsoo Rhu, Mattan Erez, Maximizing simd resource utilization in GPGPUs with SIMD lane permutation, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, 2013.
[190] Minsoo Rhu, Mattan Erez, The dual-path execution model for efficient GPU control flow, in: Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture, HPCA '13, 201, pp. 591–602b.
[191] Minsoo Rhu, Mattan Erez, CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures, in: Proceedings of the 39th International Symposium on Computer Architecture, IEEE Press, 2012, pp. 61–71.
[192] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, Stephen W. Keckler, vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design, in: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016.
[193] Minsoo Rhu, Michael Sullivan, Jingwen Leng, Mattan Erez, A locality-aware memory hierarchy for energy-efficient GPU architectures, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 86–98.
[194] Timothy Rogers, Daniel Johnson, Mike O'Connor, A variable warp size architecture, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.
[195] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt, Cache-conscious wavefront scheduling, in: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2012, pp. 72–83.
[196] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt, Divergence-aware warp scheduling, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 99–110.
[197] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, Scott Mahlke, Paragon: collaborative speculative loop execution on GPU and CPU, in: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, ACM, 2012, pp. 64–73.
[198] Mehrzad Samadi, Janghaeng Lee, D.A.noushe Jamshidi, Amir Hormati, Scott Mahlke, SAGE: self-tuning approximation for graphics engines, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 13–24.
[199] John Sartori, Rakesh Kumar, Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications, IEEE Trans. Multimed. 15 (2) (2013) 279–290.
[200] Vijay Sathish, Michael J. Schulte, Nam Sung Kim, Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ACM, 2012, pp. 325–334.
[201] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert. Cavin, et al., Larrabee: a many-core x86 architecture for visual computing, ACM Trans. Graph. (2008).
[202] D. Sengupta, A. Goswami, K. Schwan, K. Pallavi, Scheduling multi-tenant cloud workloads on accelerator-based systems, in: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 513–524, http://dx.doi.org/10.1109/SC.2014.47.
[203] Ankit Sethia, Anoushe Jamshidi, Scott Mahlke, Mascar: Speeding up GPU warps by reducing memory pitstops, in: High Performance Computer Architecture, HPCA, 2015 IEEE 21th International Symposium on, 2015.
[204] Ankit Sethia, Mahlke Scott, Equalizer: Dynamic tuning of GPU resources for efficient execution, in: Proceedings of the 47nd Annual 2014 IEEE/ACM International Symposium on Microarchitecture, 2014.
[205] Mark Silberstein, Bryan Ford, Emmett Witchel, Gpufs: The case for operating system services on gpus, Commun. ACM (ISSN: 0001-0782) 57 (12) (2014) 68–79, http://dx.doi.org/10.1145/2656206.

[206] Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve, Efficient GPU synchronization without scopes: Saying no to complex consistency models, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.

[207] Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve, Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 161–174.

[208] Abhayendra Singh, Shaizeen Aga, Satish Narayanasamy, Efficiently enforcing memory ordering in GPUs, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.

[209] Inderpreet Singh, Arrvindh Shriraman, Wilson W.L. Fung, Mike O'Connor, Tor M. Aamodt, Cache coherence for GPU architectures, in: Proceedings of the 20th International Symposium on High Performance Computer Architecture, HPCA '13, 2013, pp. 578–590.

[210] Mingcong Song, Yang Hu, Huixiang Chen, Tao Li, Towards pervasive and user satisfactory cnn across gpu microarchitectures, in: High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on, IEEE, 2017.

[211] Seokwoo Song, Minseok Lee, John Kim, Woong Seo, Yeongon Cho, Soojung Ryu, Energy-efficient scheduling for memory-intensive GPGPU workloads, in: Design, Automation and Test in Europe Conference and Exhibition, DATE, IEEE, 2014, pp. 1–6.

[212] Mingcong Song, Jiaqi Zhang, Huixiang Chen, Tao Li, Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning, in: High Performance Computer Architecture, HPCA, 2018 IEEE International Symposium on, IEEE, 2018, pp. 66–77.

[213] Tyler Sorensen, Ganesh Gopalakrishnan, Vinod Grover, Towards shared memory consistency models for GPUs, in: Proceedings of the 27th international ACM conference on International conference on supercomputing, ACM, 2013, pp. 489–490.

[214] Kyle L. Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, Jeffrey S. Vetter, The tradeoffs of fused memory hierarchies in heterogeneous computing architectures, in: Proceedings of the 9th conference on Computing Frontiers, ACM, 2012, pp. 103–112.

[215] Michael Steffen, Joseph Zambreno, Improving SIMT efficiency of global rendering algorithms with architectural support for dynamic micro-kernels, in: Microarchitecture, MICRO, 2010 43rd Annual IEEE/ACM International Symposium on, IEEE, 2010, pp. 237–248.

[216] Sylvain Collange, Stack-Less SIMT Reconvergence at Low Cost, Technical report, Technical Report HAL-00622654, INRIA, 2011.

[217] Sylvain Collange, Identifying Scalar Behavior in Cuda Kernels, Tech. Rep. hal-00555134, INRIA, France, 2011.

[218] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, Mateo Valero, Enabling preemptive multiprogramming on GPUs, in: Proceedings of the 41th International Symposium on Computer Architecture, ISCA, 2014.

[219] Ivan Tanasic, Isaac Gelado, Marc Jorda, Eduard Ayguade, Nacho Navarro, Efficient exception handling support for gpus, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 109–122.

[220] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, Chita R. Das, Controlled kernel launch for dynamic parallelism in gpus, in: High Performance Computer Architecture, HPCA, 2017 IEEE International Symposium on, IEEE, 2017, pp. 649–660.

[221] David Tarjan, Jiayuan Meng, Kevin Skadron, Increasing memory miss tolerance for SIMD cores, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 22.

[222] Chris J. Thompson, Sahngyun Hahn, Mark Oskin, Using modern graphics architectures for general-purpose computing: a framework and analysis, in: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, IEEE Computer Society Press, 2002, pp. 306–317.

[223] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, Daniel A. Jiménez, Adaptive GPU cache bypassing, in: Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, GPGPU 2015, 2015.

[224] Top500, 2015, http://www.top500org/lists/2015/11/.

[225] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, Onur Mutlu, Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators, ACM Trans. Archit. Code Optim. (2016).

[226] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk. Woo, Roy Saharoy, Mani Azimi, SIMD divergence optimization through intra-warp compaction, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ACM, 2013, pp. 368–379.

[227] J. Veselý, A. Basu, A. Bhattacharjee, G. Loh, M. Oskin, S.K. Reinhardt, GPU system calls, May 2017, ArXiv e-prints.

[228] J. Vesely, A. Basu, M. Oskin, G.H. Loh, A. Bhattacharjee, Observations and opportunities in architecting shared virtual memory for heterogeneous systems, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2016, pp. 161–171.

[229] Thiruvengadam Vijayaraghavany, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, et al., Design and analysis of an apu for exascale computing, in: High Performance Computer Architecture, HPCA, 2017 IEEE International Symposium on, IEEE, 2017, pp. 85–96.

[230] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P.B. Gibbons, O. Mutlu, Zorua: A holistic approach to resource virtualization in GPUs, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–14.

[231] Nandita. Vijaykumar, Gennady. Pekhimenko, Adwait. Jog, Abhishek. Bhowmick, Rachata. Ausavarungnirun, Chita. Das, Mahmut. Kandemir, Todd C. Mowry, Onur. Mutlu, A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.

[232] Oreste Villa, Daniel R. Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al., Scaling the power wall: a path to exascale, in: High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for, IEEE, 2014, pp. 830–841.

[233] Dani Voitsechov, Yoav Etsion, Single-graph multiple flows: Energy efficient design alternative for gpgpus, in: Computer Architecture, ISCA, 2014 ACM/IEEE 41st International Symposium on, 2014.

[234] Dani Voitsechov, Yoav Etsion, Control flow coalescing on a hybrid dataflow/von Neumann GPGPU, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.

[235] Yaohua Wang, Shuming Chen, Jianghua Wan, Jiayuan Meng, Kai Zhang, Wei Liu, Xi Ning, A multiple SIMD multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments, in: Proceedings of the 20th International Symposium on High Performance Computer Architecture, HPCA '13, 201, pp. 603–614.

[236] Jin Wang, Norm Rubin, Albert Sidelnik, Sudhakar Yalamanchili, Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, 2015.

[237] J. Wang, N. Rubin, A. Sidelnik, S. Yalamanchili, LaPerm: Locality aware scheduler for dynamic parallelism on GPUs, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, 2016, pp. 583–595a.

[238] Hao Wang, Ripudaman Singh, Michael J. Schulte, Nam Sung Kim, Memory scheduling towards high-throughput cooperative heterogeneous computing, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, 2014.

[239] Jin Wang, Sudhakar Yalamanchili, Characterization and analysis of dynamic parallelism in unstructured GPU applications, in: Workload Characterization, IISWC, 2014 IEEE International Symposium on, 2014.

[240] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 358–369c.

[241] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, Minyi Guo, Simultaneous multikernel: Fine-grained sharing of GPGPUs, Comput. Archit. Lett. (2015).

[242] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, Minyi Guo, Quality of service support for fine-grained sharing on gpus, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 269–281.

[243] Bin Wang, Weikuan Yu, Xian-He Sun, Xinning Wang, DaCache: Memory divergence-aware GPU cache management, in: Proceedings of the 29th ACM on International Conference on Supercomputing, ACM, 2015, pp. 89–98.

[244] Bin Wang, Yue Zhu, Weikuan Yu, OAWS: Memory occlusion aware warp scheduling, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT '16, 2016, pp. 45–55b.

[245] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, Hong Wang, Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008.

[246] D. Wong, N.S. Kim, M. Annavaram, Approximating warps with intra-warp operand value similarity, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 176–187.

[247] Dong Hyuk Woo, Hsien-Hsin S. Lee, COMPASS: a programmable data prefetcher using idle GPU shaders, in: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, 2010.

[248] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, Srimat Chakradhar, Optimizing data warehousing applications for GPUs using kernel fusion/fission, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW, 2012 IEEE 26th International, IEEE, 2012, pp. 2433–2442.

[249] Bo. Wu, Xu. Liu, Xiaobo. Zhou, Changjun Jiang, Flep: Enabling flexible and efficient preemption on gpus, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2017, pp. 483–496.
[250] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, Huiyang Zhou, Michael Mantor, Norman Rubin, Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ACM, 2013, pp. 433–442.
[251] Ping Xiang, Yi Yang, Huiyang Zhou, Warp-level divergence in GPUs: Characterization, impact, and mitigation, in: High Performance Computer Architecture, HPCA, 2014 IEEE 20th International Symposium on, 2014.
[252] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, Dongrui Fan, Enabling coordinated register allocation and thread-level parallelism optimization for GPUs, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[253] Xiaolong Xie, Yun Liang, Guangyu Sun, Deming Chen, An efficient compiler framework for cache bypassing on GPUs, in: Computer-Aided Design, ICCAD, 2013 IEEE/ACM International Conference on, 2013.
[254] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, Tao Wang, Coordinated static and dynamic cache bypassing for GPUs, in: High Performance Computer Architecture, HPCA, 2015 IEEE 21st International Symposium on, 2015.
[255] Q. Xu, H. Jeon, K. Kim, W.W. Ro, M. Annavaram, Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, 2016, pp. 230–242.
[256] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, Depei Qian, Software transactional memory for GPU architectures, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, 2014, p. 1.
[257] Yi Yang, Ping Xiang, Michael Mantor, Norman Rubin, Lisa Hsu, Qunfeng Dong, Huiyang Zhou, A case for a flexible scalar unit in SIMT architecture, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 93–102.
[258] Y. Yang, P. Xiang, M. Mantor, N. Rubin, H. Zhou, Shared memory multiplexing: A novel way to improve GPGPU performance, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, 2012.
[259] Yi Yang, Ping Xiang, Mike Mantor, Huiyang Zhou, CPU-assisted GPGPU on fused CPU–GPU architectures, in: High Performance Computer Architecture, HPCA, 2012 IEEE 18th International Symposium on, IEEE, 2012, pp. 1–12.
[260] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kerman, Hadi Esmaeilzadeh, Neural acceleration for gpu throughput processors, in: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.
[261] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, Todd C. Mowry, RFVP: rollback-free value prediction with safe-to-approximate loads, ACM Trans. Archit. Code Optim. 12 (4) (2016) 62.
[262] A. Yilmazer, Zhongliang Chen, D. Kaeli, Scalar waving: Improving the efficiency of simd execution on GPUs, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014.
[263] Ayse Yilmazer, David Kaeli, HQL: A scalable synchronization mechanism for GPUs, in: Parallel & Distributed Processing, IPDPS, 2013 IEEE 27th International Symposium on, IEEE, 2013, pp. 475–486.
[264] J. Yin, O. Kayiran, M. Poremba, N.E. Jerger, G.H. Loh, Efficient synthetic traffic models for large, complex SoCs, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 297–308.
[265] M.K. Yoon, K. Kim, S. Lee, W.W. Ro, M. Annavaram, Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, 2016, pp. 609–621.
[266] Myung Kuk Yoon, Yunho Oh, Sangpil Lee, Seung Hun Kim, Deokho Kim, Won Woo Ro, DRAW: Investigating benefits of adaptive fetch group size on GPU, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2015.
[267] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, Scott Mahlke, Scalpel: Customizing dnn pruning to the underlying hardware parallelism, in: ACM SIGARCH Computer Architecture News, 2017.
[268] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, Xin Chen, A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in GPGPUs, in: Proceedings of the 29th ACM on International Conference on Supercomputing, 2015.
[269] George L. Yuan, Ali Bakhoda, Tor M. Aamodt, Complexity effective memory access scheduling for many-core accelerator architectures, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2009, pp. 34–44.
[270] Vitaly Zakharenko, Tor Aamodt, Andreas Moshovos, Characterizing the performance benefits of fused CPU/GPU systems using FusionSim, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2013, 2013, pp. 685–688.
[271] J. Zhan, O. Kay'ran, G.H. Loh, C.R. Das, Y. Xie, OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU–GPU architectures, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2016, pp. 1–13.
[272] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, Xipeng Shen, On-the-fly elimination of dynamic irregularities for GPU computing, in: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2011.
[273] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, S.W. Keckler, Towards high performance paged memory for GPUs, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 345–357.
[274] Z. Zheng, Z. Wang, M. Lipasti, Adaptive cache and concurrency allocation on GPGPUs, Comput. Archit. Lett. PP (99) (2014).
[275] Jianlong Zhong, Bingsheng He, Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling, IEEE Trans. Parallel Distrib. Syst. (2013).
[276] Amir Kavyan Ziabari, José L. Abellán, Yenai Ma, Ajay Joshi, David Kaeli, Asymmetric NoC architectures for gpu systems, in: Proceedings of the 9th International Symposium on Networks-on-Chip, ACM, 2015, p. 25.

Mahmoud Khairy received his B.Sc. and M.Sc. in Computer Engineering from Cairo University, Egypt. He is currently a Ph.D. student with the Electrical and Computer Engineering Department at Purdue University, US. His research interests include GPGPU architecture, FPGAs, heterogeneous architecture and emerging memory technologies.

Amr G. Wassal received his Ph.D. degree in Electrical and Computer Engineering from the University of Waterloo, Ontario, Canada, in 2000. He has held several senior technical positions in the industry at SiWare Systems, PMC-Sierra, and IBM Technology Group. He is currently a Professor with the Computer Engineering Department, Cairo University. He has a number of conference and journal papers and patent applications in the areas of multi-core architectures and their applications in DSP and sensor fusion.

Mohamed Zahran received his Ph.D. in Electrical and Computer Engineering from University of Maryland at College Park in 2003. He is currently a faculty member with the Computer Science Department at NYU. His research interest spans several aspects of computer architecture, such as architecture of heterogeneous systems, hardware/software interaction, and biologically-inspired architectures. Zahran is a senior member of IEEE, senior member of ACM, and Sigma Xi scientific honor society.