A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps

Nandita Vijaykumar Gennady Pekhimenko Adwait Jog[†] Abhishek Bhowmick
Rachata Ausavarungnirun Chita Das[†] Mahmut Kandemir[†] Todd C. Mowry Onur Mutlu

Carnegie Mellon University [†] **Pennsylvania State University**

{nandita,abhowmick,rachata,onur}@cmu.edu
{gpekhime,tcm}@cs.cmu.edu {adwait,das,kandemir}@cse.psu.edu

Abstract

Modern Graphics Processing Units (GPUs) are well provisioned to support the concurrent execution of thousands of threads. Unfortunately, different bottlenecks during execution and heterogeneous application requirements create imbalances in utilization of resources in the cores. For example, when a GPU is bottlenecked by the available off-chip memory bandwidth, its computational resources are often overwhelmingly idle, waiting for data from memory to arrive.

This paper introduces the Core-Assisted Bottleneck Acceleration (CABA) framework that employs idle on-chip resources to alleviate different bottlenecks in GPU execution. CABA provides flexible mechanisms to automatically generate "assist warps" that execute on GPU cores to perform specific tasks that can improve GPU performance and efficiency.

CABA enables the use of idle computational units and pipelines to alleviate the memory bandwidth bottleneck, e.g., by using assist warps to perform data compression to transfer less data from memory. Conversely, the same framework can be employed to handle cases where the GPU is bottlenecked by the available computational units, in which case the memory pipelines are idle and can be used by CABA to speed up computation, e.g., by performing memoization using assist warps.

We provide a comprehensive design and evaluation of CABA to perform effective and flexible data compression in the GPU memory hierarchy to alleviate the memory bandwidth bottleneck. Our extensive evaluations show that CABA, when used to implement data compression, provides an average performance improvement of 41.7% (as high as 2.6X) across a variety of memory-bandwidth-sensitive GPGPU applications.

1. Introduction

Modern Graphics Processing Units (GPUs) play an important role in delivering high performance and energy efficiency for many classes of applications and different computational platforms. GPUs employ fine-grained multi-threading to hide the high memory access latencies with thousands of concurrently running threads [50]. GPUs are well provisioned with different resources (e.g., SIMD-like computational units, large register files) to support the execution of a large number of these hardware contexts. Ideally, if the demand for all types of resources is properly balanced, all these resources should be fully utilized by the application. Unfortunately, this balance is very difficult to achieve in practice.

As a result, bottlenecks in program execution, e.g., limitations in memory or computational bandwidth, lead to long stalls and idle periods in the shader pipelines of modern GPUs [45, 46, 60, 74]. Alleviating these bottlenecks with optimizations implemented in dedicated hardware requires significant engineering cost and effort. Fortunately, the resulting under-utilization of on-chip computational and memory resources from these imbalances in application requirements, offers some new opportunities. For example, we can use these resources for efficient integration of *hardware-generated threads* that perform useful work to accelerate the execution of the primary threads. Similar *helper threading* ideas have been proposed in the context of general-purpose processors [19, 20, 24, 27, 28, 69, 86] to either extend the pipeline with more contexts or use spare hardware contexts to pre-compute useful information that aids main code execution (e.g., to aid branch prediction, prefetching, etc.).

We believe that the general idea of helper threading can lead to even more powerful optimizations and new opportunities in the context of modern GPUs than in CPUs because (1) the abundance of on-chip resources in a GPU obviates the need for idle hardware contexts [24, 25] or the addition of more storage (registers, rename tables, etc.) and compute units [19, 59] required to handle more contexts and (2) the relative simplicity of the GPU pipeline avoids the complexities of handling register renaming, speculative execution, precise interrupts, etc. [20]. However, GPUs that execute and manage thousands of thread contexts at the same time pose new challenges for employing helper threading, which must be addressed carefully. First, the numerous regular program threads executing in parallel could require an equal or larger number of helper threads to be managed at low cost. Second, the compute and memory resources are dynamically partitioned between threads in GPUs, and resource allocation for helper threads should be cognizant of resource interference and overheads. Third, lock-step execution and complex scheduling—which are characteristic of GPU architectures—exacerbate the complexity of fine-grained management of helper threads.

In this paper, we develop a new, flexible framework for bottleneck acceleration in GPUs via helper threading (called *Core-Assisted Bottleneck Acceleration* or CABA), which exploits the aforementioned new opportunities while effectively handling the new challenges. CABA performs acceleration by generating special warps—*assist warps* —that can execute code to speed up application execution. To simplify the support of the numerous assist threads with CABA, we manage their execution at the granularity of a *warp* and use a centralized mechanism to track

ISCA'15, June 13-17, 2015, Portland, OR USA
 ISBN 978-1-4503-3402-0/15/06 $15.00
DOI: http://dx.doi.org/10.1145/2749469.2750399

the progress of each *assist warp* throughout its execution. To reduce the overhead of providing and managing new contexts for each generated thread, as well as to simplify scheduling and data communication, an assist warp *shares the same context* as the regular warp it assists. Hence, the regular warps are overprovisioned with *available registers* to enable each of them to host its own assist warp.

Use of CABA for compression. We illustrate an important use case for the CABA framework: alleviating the memory bandwidth bottleneck by enabling *flexible data compression* in the memory hierarchy. The basic idea is to have assist warps that (1) compress cache blocks before they are written to memory, and (2) decompress cache blocks before they are placed into the cache.

CABA-based compression/decompression provides several benefits over a purely hardware-based implementation of data compression for memory. First, CABA primarily employs hardware that is already available on-chip but is otherwise underutilized. In contrast, hardware-only compression implementations require dedicated logic for specific algorithms. Each new algorithm (or a modification of an existing one) requires engineering effort and incurs hardware cost. Second, different applications tend to have distinct data patterns [65] that are more efficiently compressed with different compression algorithms. CABA offers versatility in algorithm choice as we find that many existing hardware-based compression algorithms (e.g., Base-Delta-Immediate (BDI) compression [65], Frequent Pattern Compression (FPC) [4], and C-Pack [22]) can be implemented using different assist warps with the CABA framework. Third, not all applications benefit from data compression. Some applications are constrained by other bottlenecks (e.g., oversubscription of computational resources), or may operate on data that is not easily compressible. As a result, the benefits of compression may not outweigh the cost in terms of additional latency and energy spent on compressing and decompressing data. In these cases, compression can be easily disabled by CABA, and the CABA framework can be used in other ways to alleviate the current bottleneck.

Other uses of CABA. The generality of CABA enables its use in alleviating other bottlenecks with different optimizations. We discuss two examples: (1) using assist warps to perform *memoization* to eliminate redundant computations that have the same or similar inputs [12, 26, 77], by storing the results of frequently-performed computations in the main memory hierarchy (i.e., by converting the computational problem into a storage problem) and, (2) using the idle memory pipeline to perform opportunistic *prefetching* to better overlap computation with memory access. Assist warps offer a hardware/software interface to implement hybrid prefetching algorithms [31] with varying degrees of complexity.

Contributions. We make the following contributions:

- We introduce the *Core-Assisted Bottleneck Acceleration (CABA) Framework*, which can mitigate different bottlenecks in modern GPUs by using underutilized system resources for *assist warp* execution.
- We provide a detailed description of how our framework can be used to enable effective and flexible data compression in GPU memory hierarchies.
- We comprehensively evaluate the use of CABA for data compression to alleviate the memory bandwidth bottleneck. Our evaluations across a wide variety applications from Mars [39], CUDA [62], Lonestar [17], and Rodinia [21] benchmark suites show that CABA-based compression on average (1) reduces memory bandwidth by 2.1X, (2) improves performance by 41.7%, and (3) reduces overall system energy by 22.2%.

2. Motivation

We observe that different bottlenecks and imbalances during program execution leave resources unutilized within the GPU cores. We motivate CABA by examining these inefficiencies and leverage them as an opportunity to perform useful work.

Unutilized Compute Resources. A GPU core employs fine-grained multithreading of *warps*, i.e., groups of threads executing the same instruction, to hide long memory and ALU operation latencies. If the number of available warps is insufficient to cover these long latencies, the core stalls or becomes idle. To understand the key sources of inefficiency in GPU cores, we conduct an experiment where we show the breakdown of the applications' execution time spent on either useful work (*Active Cycles*) or stalling due to one of the four reasons: *Compute, Memory, Data Dependence Stalls* and *Idle Cycles*. We also vary the amount of available off-chip memory bandwidth: (i) half (1/2xBW), (ii) equal to (1xBW), and (iii) double (2xBW) the peak memory bandwidth of our baseline GPU architecture. Section 5 details our architecture and methodology.

Figure 1 shows the percentage of total issue cycles, divided into five components (as described above). The first two components—*Memory and Compute Stalls*—are attributed to the main memory and ALU-pipeline structural stalls. These stalls are because of backed-up pipelines due to oversubscribed resources that prevent warps from being issued to the respective pipelines. The third component (*Data Dependence Stalls*) is due to data dependence stalls. These stalls prevent warps from issuing new instruction(s) when the previous instruction(s) from the same warp are stalled on long-latency operations (usually memory load operations). In some applications (e.g., `dmr`), special-function-unit (SFU) ALU operations that may take tens of cycles to finish are also the source of data dependence stalls. The fourth component, *Idle Cycles*, refers to idle cycles when all the available warps are either issued to the pipelines and not ready to execute their next instruction or the instruction buffers may have been flushed due to a mispredicted branch. All these components are sources of inefficiency that cause the cores to be underutilized. The last component, *Active Cycles*, indicates the fraction of cycles during which at least one warp was successfully issued to the pipelines.

We make two observations from Figure 1. First, *Compute, Memory*, and *Data Dependence Stalls* are the major sources of underutilization in many GPU applications. We distinguish applications based on their primary bottleneck as either *Memory* or *Compute Bound*. We observe that a majority of the applications in our workload pool (17 out of 27 studied) are *Memory Bound*, and bottlenecked by the off-chip memory bandwidth.

Second, for the *Memory Bound* applications, we observe that the *Memory* and *Data Dependence* stalls constitute a significant fraction (61%) of the total issue cycles on our baseline GPU architecture (1xBW). This fraction goes down to 51% when the peak memory bandwidth is doubled (2xBW), and increases significantly when the peak bandwidth is halved (1/2xBW), indicating that limited off-chip memory bandwidth is a crit-

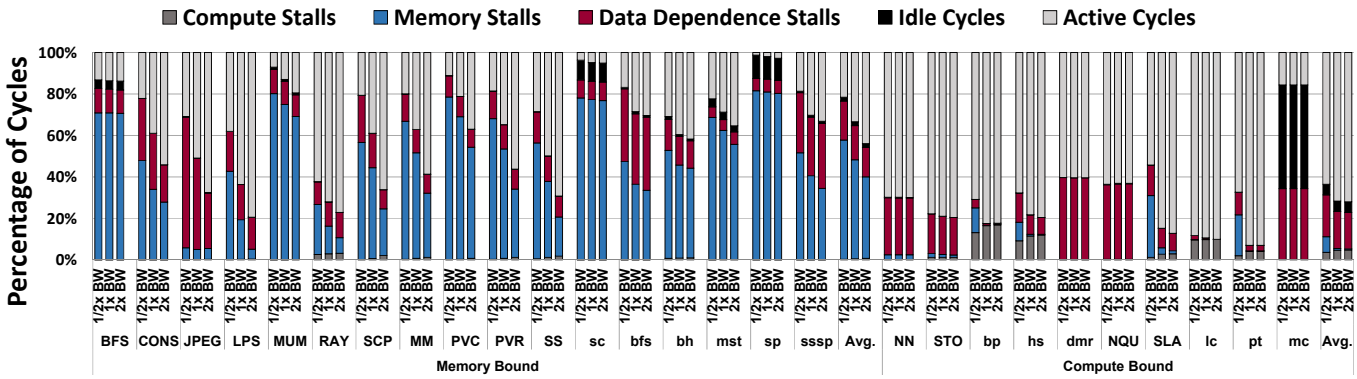

Figure 1: Breakdown of total issue cycles for 27 representative CUDA applications. See Section 5 for methodology.

ical performance bottleneck for *Memory Bound* applications. Some applications, e.g., *BFS*, are limited by the interconnect bandwidth. In contrast, the *Compute Bound* applications are primarily bottlenecked by stalls in the ALU pipelines. An increase or decrease in the off-chip bandwidth has little effect on the performance of these applications.

Unutilized On-chip Memory. The occupancy of any GPU Streaming Multiprocessor (SM), i.e., the number of threads running concurrently, is limited by a number of factors: (1) the available registers and shared memory, (2) the hard limit on the number of threads and thread blocks per core, (3) the number of thread blocks in the application kernel. Very often, the factor determining the occupancy is the thread or thread block limit imposed by the architecture. In this case, there are many registers that are left unallocated to any thread block. Also, the number of available registers may not be a multiple of those required by each thread block. The remaining registers are not enough to schedule an entire extra thread block, which leaves a significant fraction of the register file and shared memory unallocated and unutilized by the thread blocks. Figure 2 shows the fraction of statically unallocated registers in a 128KB register file (per SM) with a 1536 thread, 8 thread block occupancy limit, for different applications. We observe that on average 24% of the register file remains unallocated. This phenomenon has previously been observed and analyzed in detail in [3, 35, 36, 37, 52]. We observe a similar trend with the usage of shared memory (not graphed).

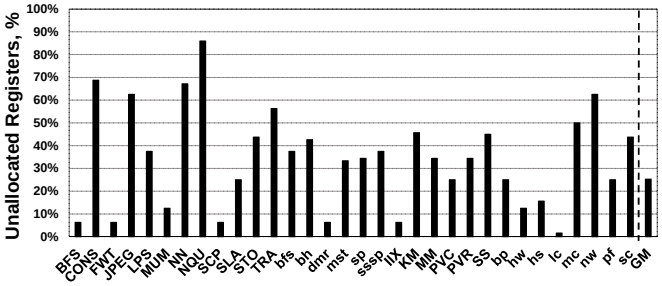

Figure 2: Fraction of statically unallocated registers.

Our Goal. We aim to exploit the underutilization of compute resources, registers and on-chip shared memory as an opportunity to enable different optimizations to accelerate various bottlenecks in GPU program execution. To do so, we need to dynamically generate threads in hardware that use the available on-chip resources. In the next section, we present the detailed design of our CABA framework that enables the generation and management of these threads.

3. The CABA Framework

In order to understand the major design choices behind the CABA framework, we first present our major design goals and describe the key challenges in applying helper threading to GPUs. We then show the detailed design, hardware changes, and operation of CABA. Finally, we briefly describe potential applications of our proposed framework.

3.1. Goals and Challenges

The purpose of CABA is to leverage underutilized GPU resources for useful computation. To this end, we need to efficiently execute subroutines that perform optimizations to accelerate bottlenecks in application execution. The key difference between CABA's *assisted execution* and regular execution is that CABA must be *low overhead* and, therefore, helper threads need to be treated differently from regular threads. The *low overhead* goal imposes several key requirements in designing a framework to enable helper threading. First, we should be able to easily manage helper threads—to enable, trigger, and kill threads when required. Second, helper threads need to be flexible enough to adapt to the runtime behavior of the regular program. Third, a helper thread needs to be able to communicate with the original thread. Finally, we need a flexible interface to specify new subroutines, with the framework being generic enough to handle various optimizations.

With the above goals in mind, enabling helper threading in GPU architectures introduces several new challenges. First, execution on GPUs involves context switching between hundreds of threads. These threads are handled at different granularities in hardware and software. The programmer reasons about these threads at the granularity of a thread block. However, at any point in time, the hardware executes only a small subset of the thread block, also referred to as a warp. Therefore, we need to define the *abstraction levels* for reasoning about and managing helper threads from the point of view of the programmer, the hardware as well as the compiler/runtime. In addition, each of the thousands of executing threads could simultaneously invoke an associated helper thread subroutine. To keep the management overhead low, we need an efficient mechanism to handle helper threads at this magnitude.

Furthermore, GPUs use fine-grained multithreading [76, 80] to time multiplex the fixed number of compute units among the hundreds of threads. Similarly, the on-chip memory resources (i.e., the register file and shared memory) are statically partitioned between the different threads at compile time. Helper threads also require their own registers and compute cycles to

execute. A straightforward approach would be to add a few registers and compute units just for helper thread execution, but this option is both expensive and wasteful. In fact, our primary motivation is to utilize *existing idle resources* for helper thread execution. In order to do this, we aim to enable sharing of the existing resources between primary threads and helper threads at low cost, while minimizing the interference to primary thread execution. In the remainder of this section, we describe the design of our low-overhead CABA framework.

3.2. Design of the CABA Framework

We choose to implement CABA using a hardware/software co-design, as pure hardware or pure software approaches pose certain challenges that we describe below. There are two alternatives for a fully software-based approach to helper threads. The first alternative, treating each helper thread as independent kernel code, has high overhead, since we are now treating the helper threads as, essentially, regular threads. This would reduce the primary thread occupancy in each SM (there is a hard limit on the number of threads and blocks that an SM can support). Moreover, this would require additional hardware changes to support blocks executing different program code simultaneously within an SM. It would also complicate the data communication between the primary and helper threads, since no simple interface exists for inter-kernel communication. The second alternative, embedding the helper thread code within the primary thread kernel itself, offers little flexibility in adapting to runtime requirements, since such helper threads cannot be triggered or squashed independently of the primary thread.

On the other hand, a pure hardware solution would make register allocation for the assist warps and the data communication between the helper threads and primary threads more difficult. Registers are allocated to each thread block by the compiler and are then mapped to the sections of the hardware register file at runtime. Mapping registers for helper threads and enabling data communication between those registers and the primary thread registers would be non-trivial. Furthermore, a fully hardware approach would make offering the programmer a flexible interface more challenging.

Hardware support enables simpler fine-grained management of helper threads, aware of micro-architectural events and runtime program behavior. Compiler/runtime support enables simpler context management for helper threads and more flexible programmer interfaces. Thus, to get the best of both worlds, we propose a *hardware/software cooperative approach*, where the hardware manages the scheduling and execution of helper thread subroutines, while the compiler performs the allocation of shared resources (e.g., register file and shared memory) for the helper threads and the programmer or the microarchitect provides the helper threads themselves.

3.2.1. Hardware-based management of threads. To use the available on-chip resources the same way that thread blocks do during program execution, we dynamically insert sequences of instructions into the execution stream. We track and manage these instructions at the granularity of a warp, and refer to them as ***Assist Warps***. An assist warp is a set of instructions issued into the core pipelines. Each instruction is executed in lock-step across all the SIMT lanes, just like any regular instruction, with an active mask to disable lanes as necessary. The assist warp does *not* own a separate context (e.g., registers, local memory), and instead shares both a context and a warp ID with the regular warp that invoked it. In other words, each assist warp is coupled with a *parent warp*. In this sense, it is different from a regular warp and does not reduce the number of threads that can be scheduled on a single SM. Data sharing between the two warps becomes simpler, since the assist warps share the register file with the parent warp. Ideally, an assist warp consumes resources and issue cycles that would otherwise be idle. We describe the structures required to support hardware-based management of assist warps in Section 3.3.

3.2.2. Register file/shared memory allocation. Each helper thread subroutine requires a different number of registers depending on the actions it performs. These registers have a short lifetime, with no values being preserved between different invocations of an assist warp. To limit the register requirements for assist warps, we impose the restriction that only one instance of each helper thread routine can be active for each thread. All instances of the same helper thread for each parent thread use the same registers, and the registers are allocated to the helper threads statically by the compiler. One of the factors that determines the runtime SM occupancy is the number of registers required by a thread block (i.e, per-block register requirement). For each helper thread subroutine that is enabled, we add its register requirement to the per-block register requirement, to ensure the availability of registers for both the parent threads as well as every assist warp. The registers that remain unallocated after allocation among the parent thread blocks should suffice to support the assist warps. If not, register-heavy assist warps may limit the parent thread block occupancy in SMs or increase the number of register spills in the parent warps. Shared memory resources are partitioned in a similar manner and allocated to each assist warp as and if needed.

3.2.3. Programmer/developer interface. The assist warp subroutine can be written in two ways. First, it can be supplied and annotated by the programmer/developer using CUDA extensions with PTX instructions and then compiled with regular program code. Second, the assist warp subroutines can be written by the microarchitect in the internal GPU instruction format. These helper thread subroutines can then be enabled or disabled by the application programmer. This approach is similar to that proposed in prior work (e.g., [19]). It offers the advantage of potentially being highly optimized for energy and performance while having flexibility in implementing optimizations that are not trivial to map using existing GPU PTX instructions. The instructions for the helper thread subroutine are stored in an on-chip buffer (described in Section 3.3).

Along with the helper thread subroutines, the programmer also provides: (1) the *priority* of the assist warps to enable the warp scheduler to make informed decisions, (2) the trigger conditions for each assist warp, and (3) the live-in and live-out variables for data communication with the parent warps.

Assist warps can be scheduled with different priority levels in relation to parent warps by the warp scheduler. Some assist warps may perform a function that is required for correct execution of the program and are *blocking*. At this end of the spectrum, the *high priority* assist warps are treated by the scheduler as always taking higher precedence over the parent warp execution. Assist warps should be given a high priority only when they are required for correctness. *Low priority* assist warps, on the other hand, are scheduled for execution

only when computational resources are available, i.e., during idle cycles. There is no guarantee that these assist warps will execute or complete.

The programmer also provides the conditions or events that need to be satisfied for the deployment of the assist warp. This includes a specific point within the original program and/or a set of other microarchitectural events that could serve as a *trigger* for starting the execution of an assist warp.

3.3. Main Hardware Additions

Figure 3 shows a high-level block diagram of the GPU pipeline [38]. To support assist warp execution, we add three new components: (1) an Assist Warp Store to hold the assist warp code, (2) an Assist Warp Controller to perform the deployment, tracking, and management of assist warps, and (3) an Assist Warp Buffer to stage instructions from triggered assist warps for execution.

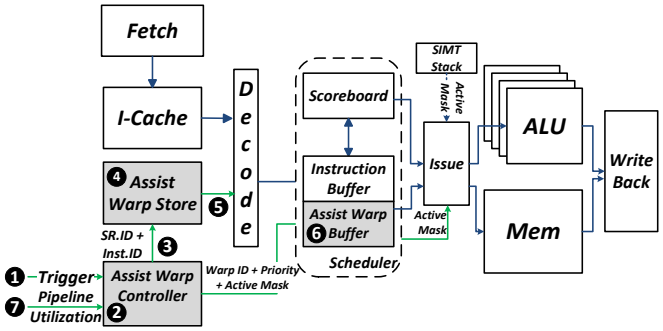

Figure 3: CABA framework flow within a typical GPU pipeline [38]. The shaded blocks are the components introduced for the framework.

Assist Warp Store (AWS). Different assist warp subroutines are possible based on the purpose of the optimization. These code sequences for different types of assist warps need to be stored on-chip. An on-chip storage structure called the Assist Warp Store (❹) is preloaded with these instructions before application execution. It is indexed using the subroutine index (SR.ID) along with the instruction ID (Inst.ID).

Assist Warp Controller (AWC). The AWC (❷) is responsible for the triggering, tracking, and management of assist warp execution. It stores a mapping between trigger events and a subroutine index in the AWS, as specified by the programmer. The AWC monitors for such events, and when they take place, triggers the fetch, decode and execution of instructions from the AWS for the respective assist warp.

Deploying all the instructions within an assist warp, back-to-back, at the trigger point may require increased fetch/decode bandwidth and buffer space after decoding [20]. To avoid this, at each cycle, only a few instructions from an assist warp, at most equal to the available decode/issue bandwidth, are decoded and staged for execution. Within the AWC, we simply track the next instruction that needs to be executed for each assist warp and this is stored in the Assist Warp Table (AWT), as depicted in Figure 4. The AWT also tracks additional metadata required for assist warp management, which is described in more detail in Section 3.4.

Assist Warp Buffer (AWB). Fetched and decoded instructions (❷) belonging to the assist warps that have been triggered need to be buffered until the assist warp can be selected for

Figure 4: Fetch Logic: Assist Warp Table (contained in the AWC) and the Assist Warp Store (AWS).

issue by the scheduler. These instructions are then staged in the Assist Warp Buffer (❻) along with their warp IDs. The AWB is contained within the *instruction buffer (IB)*, which holds decoded instructions for the parent warps. The AWB makes use of the existing IB structures. The IB is typically partitioned among different warps executing in the SM. Since each assist warp is associated with a parent warp, the assist warp instructions are directly inserted into the *same partition* within the IB as that of the parent warp. This simplifies warp scheduling, as the assist warp instructions can now be issued as if they were parent warp instructions with the same warp ID. In addition, using the existing partitions avoids the cost of separate dedicated instruction buffering for assist warps. We do, however, provision a small additional partition with two entries within the IB, to hold non-blocking *low priority* assist warps that are scheduled only during idle cycles. This additional partition allows the scheduler to distinguish *low priority* assist warp instructions from the parent warp and *high priority* assist warp instructions, which are given precedence during scheduling, allowing them to make progress.

3.4. The Mechanism

Trigger and Deployment. An assist warp is triggered (❶) by the AWC (❷) based on a specific set of architectural events and/or a triggering instruction (e.g., a load instruction). When an assist warp is triggered, its specific instance is placed into the Assist Warp Table (AWT) within the AWC (Figure 4). Every cycle, the AWC selects an assist warp to deploy in a round-robin fashion. The AWS is indexed (❸) based on the subroutine ID (SR.ID)—which selects the instruction sequence to be executed by the assist warp, and the instruction ID (Inst.ID)—which is a pointer to the next instruction to be executed within the subroutine (Figure 4). The selected instruction is entered (❺) into the AWB (❻) and, at this point, the instruction enters the active pool with other active warps for scheduling. The Inst.ID for the assist warp is updated in the AWT to point to the next instruction in the subroutine. When the end of the subroutine is reached, the entry within the AWT is freed.

Execution. Assist warp instructions, when selected for issue by the scheduler, are executed in much the same way as any other instructions. The scoreboard tracks the dependencies between instructions within an assist warp in the same way as any warp, and instructions from different assist warps are interleaved in execution in order to hide latencies. We also provide an active mask (stored as a part of the AWT), which allows for statically disabling/enabling different lanes within a warp. This is useful to provide flexibility in lock-step instruction execution when we do not need all threads within a warp to execute a specific assist warp subroutine.

Dynamic Feedback and Throttling. Assist warps, if not properly controlled, may stall application execution. This can happen due to several reasons. First, assist warps take up issue cycles, and only a limited number of instructions may be issued

per clock cycle. Second, assist warps require structural resources: the ALU units and resources in the load-store pipelines (if the assist warps consist of computational and memory instructions, respectively). We may, hence, need to throttle assist warps to ensure that their performance benefits outweigh the overhead. This requires mechanisms to appropriately balance and manage the aggressiveness of assist warps at runtime.

The overheads associated with assist warps can be controlled in different ways. First, the programmer can statically specify the priority of the assist warp. Depending on the criticality of the assist warps in making forward progress, the assist warps can be issued either in idle cycles or with varying levels of priority in relation to the parent warps. For example, warps performing *decompression* are given a high priority whereas warps performing *compression* are given a low priority. Low priority assist warps are inserted into the dedicated partition in the IB, and are scheduled only during idle cycles. This priority is statically defined by the programmer. Second, the AWC can control the number of times the assist warps are deployed into the AWB. The AWC monitors the utilization of the functional units (❼) and idleness of the cores to decide when to throttle assist warp deployment.

Communication and Control. An assist warp may need to communicate data with its parent warp. For example, memory addresses from the parent warp need to be communicated to assist warps performing decompression or prefetching. The IDs of the registers containing the live-in data for each assist warp are saved in the AWT when an assist warp is triggered. Similarly, if an assist warp needs to report results to its parent warp (e.g., in the case of memoization), the register IDs are also stored in the AWT. When the assist warps execute, *MOVE* instructions are first executed to copy the live-in data from the parent warp registers to the assist warp registers. Live-out data is communicated to the parent warp in a similar fashion, at the end of assist warp execution.

Assist warps may need to be *killed* when they are not required (e.g., if the data does not require decompression) or when they are no longer beneficial. In this case, the entries in the AWT and AWB are simply flushed for the assist warp.

3.5. Applications of the CABA Framework

We envision multiple applications for the CABA framework, e.g., data compression [4, 22, 65, 84], memoization [12, 26, 77], data prefetching [13, 34, 47, 64]. In Section 4, we provide a detailed case study of enabling data compression with the framework, discussing various tradeoffs. We believe CABA can be useful for many other optimizations, and we discuss some of them briefly in Section 7.

4. A Case for CABA: Data Compression

Data compression is a technique that exploits the redundancy in the applications' data to reduce capacity and bandwidth requirements for many modern systems by saving and transmitting data in a more compact form. Hardware-based data compression has been explored in the context of on-chip caches [4, 10, 22, 29, 43, 65, 67, 71, 84] and main memory [2, 33, 66, 75, 82] as a means to save storage capacity as well as memory bandwidth. In modern GPUs, memory bandwidth is a key limiter to system performance in many workloads (Section 2). As such, data compression is a promising technique to help alleviate this bottleneck. Compressing data enables less data to be transferred from/to DRAM and the interconnect.

In bandwidth-constrained workloads, idle compute pipelines offer an opportunity to employ CABA to enable data compression in GPUs. We can use assist warps to (1) decompress data, before loading it into the caches and registers, and (2) compress data before writing it back to memory. Since assist warps execute instructions, CABA offers some flexibility in the compression algorithms that can be employed. Compression algorithms that can be mapped to the general GPU execution model can be flexibly implemented with the CABA framework.

4.1. Mapping Compression Algorithms into Assist Warps

In order to employ CABA to enable data compression, we need to map compression algorithms into instructions that can be executed within the GPU cores. For a compression algorithm to be amenable for implementation with CABA, it ideally needs to be (1) reasonably parallelizable and (2) simple (for low latency). Decompressing data involves reading the encoding associated with each cache line that defines how to decompress it, and then triggering the corresponding decompression subroutine in CABA. Compressing data, on the other hand, involves testing different encodings and saving data in the compressed format.

We perform compression at the granularity of a cache line. The data needs to be decompressed before it is used by any program thread. In order to utilize the full SIMD width of the GPU pipeline, we would like to decompress/compress all the words in the cache line in parallel. With CABA, helper thread routines are managed at the warp granularity, enabling fine-grained triggering of assist warps to perform compression/decompression when required. However, the SIMT execution model in a GPU imposes some challenges: (1) threads within a warp operate in lock-step, and (2) threads operate as independent entities, i.e., they do not easily communicate with each other.

In this section, we discuss the architectural changes and algorithm adaptations required to address these challenges and provide a detailed implementation and evaluation of *Data Compression* within the CABA framework using the *Base-Delta-Immediate compression* algorithm [65]. Section 4.1.3 briefly discusses implementing other compression algorithms.

4.1.1. Algorithm Overview. Base-Delta-Immediate compression (BDI) is a simple compression algorithm that was originally proposed in the context of caches [65]. It is based on the observation that many cache lines contain data with low dynamic range. BDI exploits this observation to represent a cache line with low dynamic range using a common *base* (or multiple bases) and an array of *deltas* (differences between values within the cache line and the common base). Since the *deltas* require fewer bytes than the values themselves, the combined size after compression can be much smaller. Figure 5 shows the compression of an example 64-byte cache line from the *PageViewCount (PVC)* application using BDI. As Figure 5 indicates, in this case, the cache line can be represented using two bases (an 8-byte base value, $0x8001D000$, and an implicit zero value base) and an array of eight 1-byte differences from these bases. As a result, the entire cache line data can be represented using 17 bytes instead of 64 bytes (1-byte metadata, 8-byte base, and eight 1-byte deltas), saving 47 bytes of the originally used space.

Our example implementation of the BDI compression algorithm [65] views a cache line as a set of fixed-size values i.e., 8

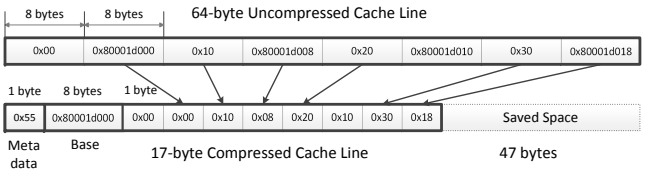

Figure 5: Cache line from *PVC* compressed with BDI.

8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line. For the size of the deltas, it considers three options: 1, 2 and 4 bytes. The key characteristic of BDI, which makes it a desirable compression algorithm to use with the CABA framework, is its fast parallel decompression that can be efficiently mapped into instructions that can be executed on GPU hardware. Decompression is simply a masked vector addition of the deltas to the appropriate bases [65].

4.1.2. Mapping BDI to CABA. In order to implement BDI with the CABA framework, we need to map compression/decompression algorithms into GPU instruction subroutines (stored in the AWS and deployed as assist warps). We provide a brief overview of how to do this, and leave the details to our extended technical report [83].

Decompression. To decompress the data compressed with BDI, we need a simple addition of deltas to the appropriate bases. The CABA decompression subroutine first loads the words within the compressed cache line into assist warp registers, and then performs the base-delta additions in parallel on the wide ALU pipeline.[1] The subroutine then writes back the uncompressed cache line to the cache. It skips the addition for the lanes with an implicit base of zero by updating the active lane mask based on the cache line encoding. We store a separate subroutine for each possible BDI encoding that loads the appropriate bytes in the cache line as the base and the deltas.

Compression. To compress data, the CABA compression subroutine tests several possible encodings (each representing a different size of base and deltas) in order to achieve a high compression ratio. The first few bytes (2–8 depending on the encoding tested) of the cache line are always used as the base. Each possible encoding is tested to check whether the cache line can be successfully encoded with it. In order to perform compression at a warp granularity, we need to check whether all of the words at every SIMD lane were successfully compressed. In other words, if any one word cannot be compressed, that encoding cannot be used across the warp. We can perform this check by adding a global predicate register, which stores the logical AND of the per-lane predicate registers. Applications with homogeneous data structures can typically use the same encoding for most of their cache lines. We use this observation to reduce the number of encodings we test to just *one* in many cases. All necessary operations are done in parallel using the full width of the GPU SIMD pipeline.

4.1.3. Implementing Other Algorithms. The BDI compression algorithm is naturally amenable towards implementation using assist warps because of its data-parallel nature and simplicity. The CABA framework can also be used to realize other algorithms. The challenge in implementing algorithms like FPC [5] and C-Pack [22][2], which have variable-length compressed words, is primarily in the placement of compressed words within the compressed cache lines. In BDI, the compressed words are in *fixed* locations within the cache line and, for each encoding, all the compressed words are of the same size and can, therefore, be processed in parallel. In contrast, C-Pack may employ multiple dictionary values as opposed to just one base in BDI. In order to realize algorithms with *variable length words* and *dictionary values* with assist warps, we leverage the coalescing/address generation logic [61, 63] already available in the GPU cores. We make two minor modifications to these algorithms [5, 22] to adapt them for use with CABA. First, similar to prior works [5, 22, 33], we observe that few encodings are sufficient to capture almost all the data redundancy. In addition, the impact of any loss in compressibility due to fewer encodings is minimal as the benefits of bandwidth compression are only at multiples of a single DRAM burst (e.g., 32B for GDDR5 [41]). We exploit this to reduce the number of supported encodings. Second, we place all the metadata containing the compression encoding at the head of the cache line to be able to determine how to decompress the entire line upfront. In the case of C-Pack, we place the dictionary entries after the metadata. Our technical report [83] describes how these algorithms are enabled with the CABA framework.

We note that it can be challenging to implement complex algorithms efficiently with the simple computational logic available in GPU cores. Fortunately, there are already Special Function Units (SFUs) [18, 55] present in the GPU SMs, used to perform efficient computations of elementary mathematical functions. SFUs could potentially be extended to implement primitives that enable the fast iterative comparisons performed frequently in some compression algorithms. This would enable more efficient execution of the described algorithms, as well as implementation of more complex compression algorithms, using CABA. We leave the exploration of an SFU-based approach to future work.

4.2. Walkthrough of CABA-based Compression

We show the detailed operation of CABA-based compression and decompression mechanisms in Figure 6. We assume a baseline GPU architecture with three levels in the memory hierarchy – two levels of caches (private L1s and a shared L2) and the main memory. Different levels can potentially store compressed data. In this section and in our evaluations, we assume that only the L2 cache and main memory contain compressed data. Note that there is no capacity benefit in the baseline mechanism as compressed cache lines still occupy the full uncompressed slot, i.e., we only evaluate the bandwidth-saving benefits of compression in GPUs.

4.2.1. The Decompression Mechanism. Load instructions that access global memory data in the compressed form trigger the appropriate assist warp to decompress the data before it is used. The subroutines to decompress data are stored in the *Assist Warp Store (AWS)*. The AWS is indexed by the compression encoding at the head of the cache line and by a bit indicating whether the instruction is a load (decompression is required) or a store (compression is required). Each decompression assist warp is given *high priority* and, hence, stalls the progress of its parent warp until it completes its execution.

L1 Access. We store data in L1 in the uncompressed form. An L1 hit does not require an assist warp for decompression.

L2/Memory Access. Global memory data cached in

[1] Multiple instructions are required if the number of deltas exceeds the width of the ALU pipeline. We use a 32-wide pipeline.

[2] Our technical report [83] and the original works [5, 22] provide more details on the specifics of these algorithms.

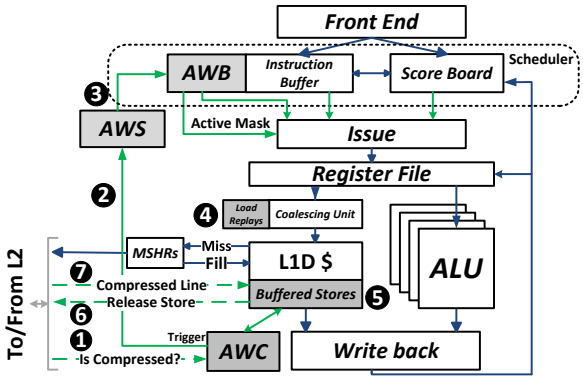

Figure 6: Walkthrough of CABA-based Compression.

L2/DRAM could potentially be compressed. A bit indicating whether the cache line is compressed is returned to the core along with the cache line (❶). If the data is uncompressed, the line is inserted into the L1 cache and the writeback phase resumes normally. If the data is compressed, the compressed cache line is inserted into the L1 cache. The encoding of the compressed cache line and the warp ID are relayed to the Assist Warp Controller (AWC), which then triggers the AWS (❷) to deploy the appropriate assist warp (❸) to decompress the line. During regular execution, the load information for each thread is buffered in the coalescing/load-store unit [61, 63] until all the data is fetched. We continue to buffer this load information (❹) until the line is decompressed.

After the CABA decompression subroutine ends execution, the original load that triggered decompression is resumed (❹).

4.2.2. The Compression Mechanism. The assist warps to perform compression are triggered by store instructions. When data is written to a cache line (i.e., by a store), the cache line can be written back to main memory either in the compressed or uncompressed form. Compression is off the critical path and the warps to perform compression can be scheduled when the required resources are available.

Pending stores are buffered in a few dedicated sets within the L1 cache or in available shared memory (❺). In the case of an overflow in this buffer space (❺), the stores are released to the lower levels of the memory system in the uncompressed form (❻). Upon detecting the availability of resources to perform the data compression, the AWC triggers the deployment of the assist warp that performs compression (❷) into the AWB (❸), with *low priority*. The scheduler is then free to schedule the instructions from the compression subroutine.

L1 Access. On a hit in the L1 cache, the cache line is already available in the uncompressed form. Depending on the availability of resources, the cache line can be scheduled for compression or simply written to the L2 and main memory uncompressed when evicted.

L2/Memory Access. Data in memory is compressed at the granularity of a full cache line, but stores can be at granularities smaller than the size of the cache line. This poses some additional difficulty if the destination cache line for a store is already compressed in main memory. Partial writes into a compressed cache line would require the cache line to be decompressed first, then updated with the new data, and written back to main memory. The common case—where the cache line being written into is uncompressed initially—can be easily handled. However, in the worst case, the cache line being partially written to is already in the compressed form in memory. We now describe the mechanism to handle these cases.

Initially, to reduce the store latency, we assume that the cache line is uncompressed, and issue a store to the lower levels of the memory hierarchy, while buffering a copy in L1. If the cache line is found in L2/memory in the uncompressed form (❶), the assumption was correct. The store then proceeds normally and the buffered stores are evicted from L1. If the assumption is incorrect, the cache line is retrieved (❼) and decompressed before the store is retransmitted to the lower levels of the memory hierarchy.

4.3. Realizing Data Compression

Supporting data compression requires additional support from the main memory controller and the runtime system, as we describe below. Our technical report [83] contains more details.

4.3.1. Initial Setup and Profiling. Data compression with CABA requires a one-time data setup before the data is transferred to the GPU. We assume initial software-based data preparation where the input data is stored in CPU memory in the compressed form with an appropriate compression algorithm before transferring the data to GPU memory. Transferring data in the compressed form can also reduce PCIe bandwidth usage.[3]

Memory-bandwidth-limited GPU applications are the best candidates for employing data compression using CABA. The compiler (or the runtime profiler) is required to identify those applications that are most likely to benefit from this framework. For applications where bandwidth is not a bottleneck, data compression is simply disabled.

4.3.2. Memory Controller Changes. Data compression reduces off-chip bandwidth requirements by transferring the same data in fewer DRAM bursts. The memory controller (MC) needs to know whether the cache line data is compressed and how many bursts (1–4 bursts in GDDR5 [41]) are needed to transfer the data from DRAM to the MC. Similar to prior work [66, 72], we require metadata information for every cache line that keeps track of how many bursts are needed to transfer the data. Similar to prior work [72], we simply reserve 8MB of GPU DRAM space for the metadata (~0.2% of all available memory). Unfortunately, this simple design would require an additional access for the metadata for every access to DRAM effectively doubling the required bandwidth. To avoid this, a simple *metadata (MD) cache* that keeps frequently-accessed metadata on chip (near the MC) is required. Our experiments show that a small 8 KB 4-way associative MD cache is sufficient to provide a hit rate of 85% on average (more than 99% for many applications) across all applications in our workload pool.[4] Hence, in the common case, a second access to DRAM to fetch compression-related metadata can be avoided.

5. Methodology

We model the CABA framework in GPGPU-Sim 3.2.1 [14]. Table 1 provides the major parameters of the simulated system. We use GPUWattch [54] to model GPU power and CACTI [81] to evaluate the power/energy overhead associated with the MD cache (Section 4.3.2) and the additional components (AWS and

[3] This requires changes to the DMA engine to recognize compressed lines.

[4] For applications where MD cache miss rate is low, we observe that MD cache misses are usually also TLB misses. Hence, most of the overhead of MD cache misses in these applications is outweighed by the cost of page table lookups.

AWC) of the CABA framework. We implement BDI [65] using the Synopsys Design Compiler with 65nm library (to evaluate the energy overhead of compression/decompression for the dedicated hardware design for comparison to CABA), and then use ITRS projections [44] to scale our results to 32nm.

System Overview	15 SMs, 32 threads/warp, 6 memory channels
Shader Core Config	1.4GHz, GTO scheduler [68], 2 schedulers/SM
Resources / SM	48 warps/SM, 32768 registers, 32KB Shared Memory
L1 Cache	16KB, 4-way associative, LRU replacement policy
L2 Cache	768KB, 16-way associative, LRU replacement policy
Interconnect	1 crossbar/direction (15 SMs, 6 MCs), 1.4GHz
Memory Model	177.4GB/s BW, 6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling, 16 banks/MC
GDDR5 Timing [41]	$t_{CL} = 12, : t_{RP} = 12, : t_{RC} = 40, : t_{RAS} = 28$, $t_{RCD} = 12, : t_{RRD} = 6 : t_{CLDR} = 5 : t_{WR} = 12$

Table 1: Major parameters of the simulated systems.

Evaluated Applications. We use a number of CUDA applications derived from CUDA SDK [62] (*BFS, CONS, JPEG, LPS, MUM, RAY, SLA, TRA*), Rodinia [21] (*hs, nw*), Mars [39] (*KM, MM, PVC, PVR, SS*) and lonestar [17] (*bfs, bh, mst, sp, sssp*) suites. We run all applications to completion or 1 billion instructions (whichever comes first). CABA-based data compression is mainly beneficial for bandwidth-limited applications. In computation-resource limited applications, data compression is not only unrewarding, but it can also cause significant performance degradation due to the computational overheads associated with assist warps. We rely on static profiling to identify bandwidth-limited applications and disable CABA-based compression for the others. In our evaluation (Section 6), we demonstrate detailed results for applications that exhibit some compressibility in bandwidth (at least 10%). Applications without compressible data (e.g., sc, SCP) do not gain any performance from the CABA framework, and we verified that these applications do not incur any degradation (because the assist warps are not triggered for them).

Evaluated Metrics. We present Instruction per Cycle (*IPC*) as the primary performance metric. We also use *average bandwidth utilization*, defined as the fraction of total DRAM cycles that the DRAM data bus is busy, and *compression ratio*, defined as the ratio of the number of DRAM bursts required to transfer data in the compressed vs. uncompressed form. As reported in prior work [65], we use decompression/compression latencies of 1/5 cycles for the hardware implementation of BDI.

6. Results

To evaluate the effectiveness of using CABA to employ data compression, we compare five different designs: (i) *Base* - the baseline system with no compression, (ii) *HW-BDI-Mem* - hardware-based *memory bandwidth compression* with dedicated logic (data is stored compressed in main memory but uncompressed in the last-level cache, similar to prior works [66, 72]), (iii) *HW-BDI* - hardware-based *interconnect and memory bandwidth compression* (data is stored uncompressed only in the L1 cache) (iv) *CABA-BDI* - Core-Assisted Bottleneck Acceleration (CABA) framework (Section 3) with all associated overheads of performing compression (for both interconnect and memory bandwidth), (v) *Ideal-BDI* - compression (for both interconnect and memory) with no latency/power overheads for compression or decompression. This section provides our major results and analyses. Our technical report [83] provides more detailed analyses.

6.1. Effect on Performance and Bandwidth Utilization

Figures 7 and 8 show, respectively, the normalized performance (vs. *Base*) and the memory bandwidth utilization of the five designs. We make three major observations.

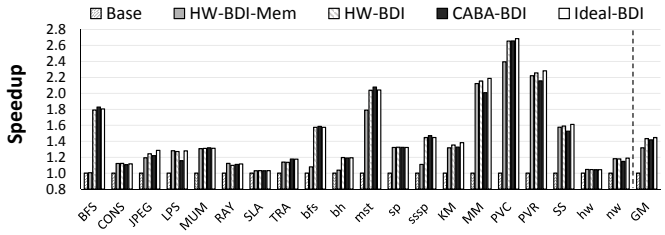

Figure 7: Normalized performance of CABA.

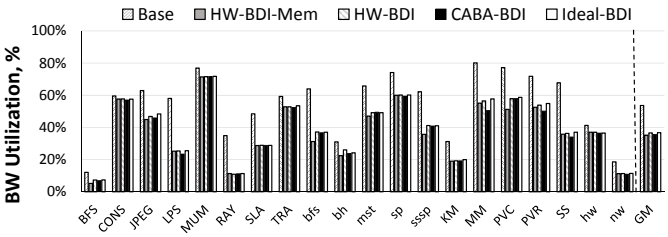

Figure 8: Memory bandwidth utilization.

First, all compressed designs are effective in providing high performance improvement over the baseline. Our approach (CABA-BDI) provides a 41.7% average improvement, which is only 2.8% less than the ideal case (Ideal-BDI) with none of the overheads associated with CABA. CABA-BDI's performance is 9.9% better than the previous [72] hardware-based memory bandwidth compression design (HW-BDI-Mem), and *only* 1.6% worse than the purely hardware-based design (HW-BDI) that performs both interconnect and memory bandwidth compression. We conclude that our framework is effective in enabling the benefits of compression without requiring specialized hardware compression and decompression logic.

Second, performance benefits, in many workloads, correlate with the reduction in memory bandwidth utilization. For a fixed amount of data, compression reduces the bandwidth utilization, and, thus, increases the effective available bandwidth. Figure 8 shows that CABA-based compression 1) reduces the average memory bandwidth utilization from 53.6% to 35.6% and 2) is effective in alleviating the memory bandwidth bottleneck in most workloads. In some applications (e.g., *bfs* and *mst*), designs that compress *both* the on-chip interconnect and the memory bandwidth, i.e. CABA-BDI and HW-BDI, perform better than the design that compresses only the memory bandwidth (HW-BDI-Mem). Hence, CABA seamlessly enables the mitigation of the interconnect bandwidth bottleneck as well, since data compression/decompression is flexibly performed at the cores.

Third, for some applications, CABA-BDI is slightly (within 3%) better in performance than Ideal-BDI and HW-BDI. The reason for this counter-intuitive result is the effect of warp oversubscription [49, 68]. In these cases, too many warps execute in parallel, polluting the last level cache. CABA-BDI sometimes

reduces pollution as a side effect of performing more computation in assist warps, which slows down the progress of the parent warps.

We conclude that the CABA framework can effectively enable data compression to reduce both on-chip interconnect and off-chip memory bandwidth utilization, thereby improving the performance of modern GPGPU applications.

6.2. Effect on Energy

Compression decreases energy consumption in two ways: 1) by reducing bus energy consumption, 2) by reducing execution time. Figure 9 shows the normalized energy consumption of the five systems. We model the static and dynamic energy of the cores, caches, DRAM, and all buses (both on-chip and off-chip), as well as the energy overheads related to compression: metadata (MD) cache and compression/decompression logic. We make two major observations.

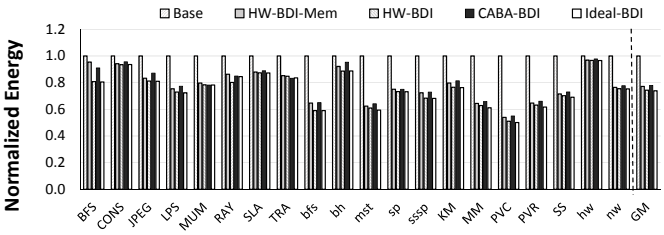

Figure 9: Normalized energy consumption of CABA.

First, CABA-BDI reduces energy consumption by as much as 22.2% over the baseline. This is especially noticeable for bandwidth-limited applications, e.g., *PVC*, *mst*. This is a result of two factors: (i) the reduction in the amount of data transferred between the LLC and DRAM (we observe a 29.5% average reduction in DRAM power) and (ii) the reduction in total execution time. This observation agrees with several prior works on bandwidth compression [66, 75]. We conclude that the CABA framework is capable of reducing the overall system energy, primarily by decreasing the off-chip memory traffic.

Second, CABA-BDI's energy consumption is only 3.6% more than that of the HW-BDI design, which uses dedicated logic for memory bandwidth compression. It is also only 4.0% more than that of the Ideal-BDI design, which has no compression-related overheads. CABA-BDI consumes more energy because it schedules and executes assist warps, utilizing on-chip register files, memory and computation units, which is less energy-efficient than using dedicated logic for compression. However, as results indicate, this additional energy cost is small compared to the performance gains of CABA (recall, 41.7% over Base), and may be amortized by using CABA for other purposes as well (see Section 7).

Power Consumption. CABA-BDI increases the system power consumption by 2.9% over the baseline (not graphed), mainly due to the additional hardware and higher utilization of the compute pipelines. However, the power overhead enables energy savings by reducing bandwidth use and can be amortized across other uses of CABA (Section 7).

6.3. Effect of Enabling Different Compression Algorithms

The CABA framework is *not limited to a single compression algorithm*, and can be effectively used to employ other hardware-based compression algorithms (e.g., FPC [4] and C-Pack [22]). The effectiveness of other algorithms depends on two key factors: (i) how efficiently the algorithm maps to GPU instructions, (ii) how compressible the data is with the algorithm. We map the FPC and C-Pack algorithms to the CABA framework and evaluate the framework's efficacy.[5]

Figure 10 shows the normalized speedup with four versions of our design: *CABA-FPC*, *CABA-BDI*, *CABA-C-Pack*, and *CABA-BestOfAll* with the FPC, BDI, C-Pack compression algorithms. CABA-BestOfAll is an idealized design that selects and uses the best of all three algorithms in terms of compression ratio for *each cache line*, assuming no selection overhead. We make three major observations.

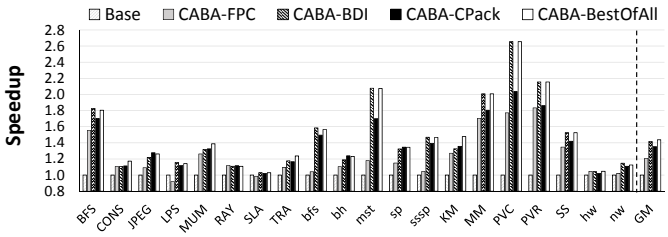

Figure 10: Speedup with different compression algorithms.

First, CABA significantly improves performance with any compression algorithm (20.7% with FPC, 35.2% with C-Pack). Similar to CABA-BDI, the applications that benefit the most are those that are both bandwidth-sensitive (Figure 8) and compressible (Figure 11). We conclude that our proposed framework, CABA, is general and flexible enough to successfully enable different compression algorithms.

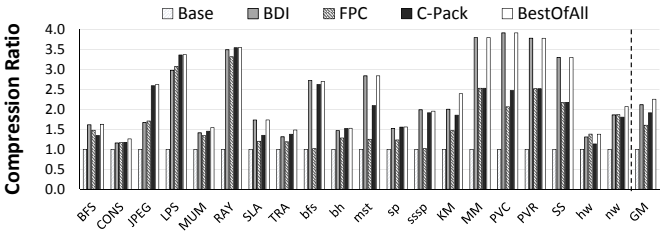

Figure 11: Compression ratio of algorithms with CABA.

Second, applications benefit differently from each algorithm. For example, *LPS, JPEG, MUM, nw* have higher compression ratios with FPC or C-Pack, whereas *MM, PVC, PVR* compress better with BDI. This motivates the necessity of having *flexible data compression* with different algorithms within the same system. Implementing multiple compression algorithms completely in hardware is expensive as it adds significant area overhead, whereas CABA can flexibly enable the use of different algorithms via its general assist warp framework.

Third, the design with the best of three compression algorithms, CABA-BestOfAll, can sometimes improve performance more than each individual design with just one compression algorithm (e.g., for *MUM* and *KM*). This happens because even within an application, different cache lines compress better with different algorithms. At the same time, different compression related overheads of different algorithms can cause one to have higher performance than another even though the latter may have a higher compression ratio. For example, CABA-BDI provides higher performance on *LPS* than CABA-FPC, even

[5] Our technical report [83] details how these algorithms are mapped to CABA.

though BDI has a lower compression ratio than FPC for *LPS*, because BDI's compression/decompression latencies are much lower than FPC's. Hence, a mechanism that selects the best compression algorithm based on *both* compression ratio and the relative cost of compression/decompression is desirable to get the best of multiple decompression algorithms. The CABA framework can flexibly enable the implementation of such a mechanism, whose design we leave for future work.

6.4. Sensitivity to Peak Main Memory Bandwidth

As described in Section 2, main memory (off-chip) bandwidth is a major bottleneck in GPU applications. In order to confirm that CABA works for different designs with varying amounts of available bandwidth, we conduct an experiment where CABA-BDI is used in three systems with 0.5X, 1X and 2X amount of bandwidth of the baseline.

Figure 12 shows the results of this experiment. We observe that, as expected, the CABA designs (**-CABA*) significantly outperform the corresponding baseline designs with the same amount of bandwidth. The performance improvement of CABA is often equivalent to the doubling the off-chip bandwidth. We conclude that CABA-based bandwidth compression, on average, offers almost all the performance benefits of doubling the available off-chip bandwidth with only modest complexity to support assist warps.

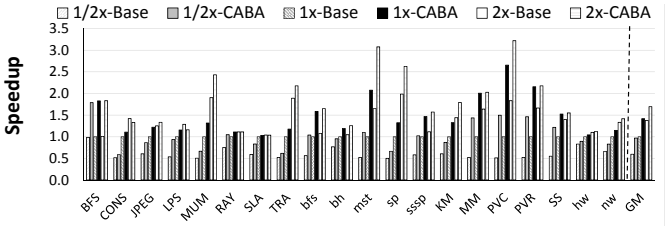

Figure 12: Sensitivity of CABA to memory bandwidth.

6.5. Selective Cache Compression with CABA

In addition to reducing bandwidth consumption, data compression can also increase the *effective capacity* of on-chip caches. While compressed caches can be beneficial—as higher effective cache capacity leads to lower miss rates—supporting cache compression requires several changes in the cache design [4, 22, 65, 71].

Figure 13 shows the effect of four cache compression designs using CABA-BDI (applied to both L1 and L2 caches with 2x or 4x the number of tags of the baseline[6]) on performance. We make two major observations. First, several applications from our workload pool are not only bandwidth sensitive, but also cache sensitive. For example, *bfs* and *sssp* significantly benefit from L1 cache compression, while *TRA* and *KM* benefit from L2 compression. Second, L1 cache compression can severely degrade the performance of some applications, e.g., *hw* and *LPS*. The reason for this is the overhead of decompression, which can be especially high for L1 caches as they are accessed very frequently. This overhead can be easily avoided by disabling compression at any level of the memory hierarchy.

The CABA framework allows us to store compressed data selectively at different levels of the memory hierarchy. We consider an optimization where we avoid the overhead of decompressing data in L2 by storing data in uncompressed form.

[6]The number of tags limits the effective compressed cache size [4, 65].

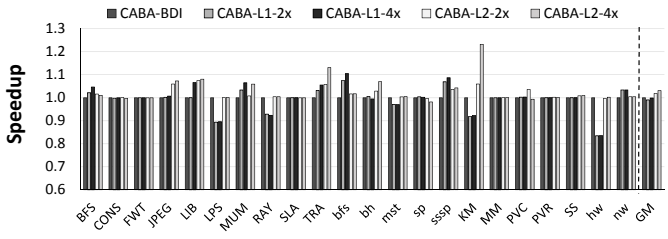

Figure 13: Speedup of cache compression with CABA.

This provides another tradeoff between the savings in on-chip traffic (when data in L2 is compressed – default option), and savings in decompression latency (when data in L2 is uncompressed). Several applications in our workload pool (e.g., *RAY*) benefit from storing data uncompressed as these applications have high hit rates in the L2 cache. We conclude that offering the choice of enabling or disabling compression at different levels of the memory hierarchy can provide application developers with an additional per-application performance knob.

7. Other Uses of the CABA Framework

The CABA framework can be employed in various ways to alleviate system bottlenecks and increase system performance and energy efficiency. In this section, we discuss two other potential applications of CABA: *Memoization* and *Prefetching*.

7.1. Memoization

Hardware memoization is a technique used to avoid redundant computations by reusing the results of previous computations that have the same or similar inputs. Prior work [8, 12, 70] observed redundancy in inputs to data in GPU workloads. In applications limited by available compute resources, memoization offers an opportunity to trade off computation for storage, thereby enabling potentially higher energy efficiency and performance. In order to realize memoization in hardware, a look-up table (LUT) is required to dynamically cache the results of computations as well as the corresponding inputs. The granularity of computational reuse can be at the level of fragments [12], basic blocks, functions [7, 9, 26, 40, 77], or long-latency instructions [23]. The CABA framework is a natural way to implement such an optimization. The availability of on-chip memory lends itself for use as the LUT. In order to cache previous results in on-chip memory, look-up tags (similar to those proposed in [37]) are required to index correct results. With applications tolerant of approximate results (e.g., image processing, machine learning, fragment rendering kernels), the computational inputs can be hashed to reduce the size of the LUT. Register values, texture/constant memory or global memory sections that are not subject to change are potential inputs. An assist warp can be employed to perform memoization in the following way: (1) compute the hashed value for look-up at predefined trigger points, (2) use the load/store pipeline to save these inputs in available shared memory, and (3) eliminate redundant computations by loading the previously computed results in the case of a hit in the LUT.

7.2. Prefetching

Prefetching has been explored in the context of GPUs [11, 45, 46, 52, 53, 58, 73] with the goal of reducing effective memory latency. With memory-latency-bound applications, the load-/store pipelines can be employed by the CABA framework

to perform opportunistic prefetching into GPU caches. The CABA framework can potentially enable the effective use of prefetching in GPUs due to several reasons: (1) Even simple prefetchers such as the stream [47, 64, 78] or stride [13, 34] prefetchers are non-trivial to implement in GPUs since access patterns need to be tracked and trained at the granularity of warps [53, 73]. CABA could enable fine-grained book-keeping by using spare registers and assist warps to save metadata for each warp. The computational units could then be used to continuously compute strides in access patterns both within and across warps. (2) It has been demonstrated that software prefetching and helper threads [1, 16, 25, 42, 42, 52, 57, 79] are very effective in performing prefetching for irregular access patterns. Assist warps offer the hardware/software interface to implement application-specific prefetching algorithms with varying degrees of complexity without the additional cost of hardware implementation. (3) In bandwidth-constrained GPU systems, uncontrolled prefetching could potentially flood the off-chip buses, delaying demand requests. CABA can enable flexible prefetch throttling (e.g., [30, 32, 78]) by scheduling assist warps that perform prefetching, only when the memory pipelines are idle. (4) Prefetching with CABA entails using load or prefetch instructions, which not only enables prefetching to the hardware-managed caches, but also simplifies usage of unutilized shared memory or register file as prefetch buffers.

8. Related Work

To our knowledge, this paper is the first to (1) propose a flexible and general framework for employing idle GPU resources for useful computation that can aid regular program execution, and (2) use the general concept of *helper threading* to perform memory and interconnect bandwidth compression. We demonstrate the benefits of our new framework by using it to implement multiple compression algorithms on a throughput-oriented GPU architecture. We briefly discuss related works in helper threading and bandwidth compression.

Helper Threading. Previous works [1, 16, 19, 20, 24, 25, 27, 28, 42, 48, 51, 56, 57, 79, 85, 86] demonstrated the use of *helper threads* in the context of Simultaneous Multithreading (SMT) and multi-core processors, primarily to speed up single-thread execution by using idle SMT contexts or idle cores in CPUs. These works typically use helper threads (generated by the software, the hardware, or cooperatively) to pre-compute useful information that aids the execution of the primary thread (e.g., by prefetching, branch outcome pre-computation, and cache management). No previous work discussed the use of helper threads for memory/interconnect bandwidth compression or cache compression.

While our work was inspired by these prior studies of helper threading in latency-oriented architectures (CPUs), developing a framework for helper threading (or *assist warps*) in throughput-oriented architectures (GPUs) enables new opportunities and poses new challenges, both due to the massive parallelism and resources present in a throughput-oriented architecture (as discussed in Section 1). Our CABA framework exploits these new opportunities and addresses these new challenges, including (1) low-cost management of dozens of assist warps that could be running concurrently with regular program warps, (2) means of state/context management and scheduling for assist warps to maximize effectiveness and minimize interference, and (3) different possible applications of the concept of assist warps in a throughput-oriented architecture.

In the GPU domain, CudaDMA [15] is a recent proposal that aims to ease programmability by decoupling execution and memory transfers with specialized DMA warps. This work does not provide a general and flexible hardware-based framework for using GPU cores to run warps that aid the main program.

Compression. Several prior works [6, 10, 66, 67, 72, 75, 82] study memory and cache compression with several different compression algorithms [4, 10, 22, 43, 65, 84], in the context of CPUs or GPUs. Our work is the first to demonstrate how one can adapt some of these algorithms for use in a general helper threading framework for GPUs. As such, compression/decompression using our new framework is more flexible since it does not require a specialized hardware implementation for any algorithm and instead utilizes the existing GPU core resources to perform compression and decompression. Finally, as discussed in Section 7, our CABA framework is applicable beyond compression and can be used for other purposes.

9. Conclusion

This paper makes a case for the Core-Assisted Bottleneck Acceleration (CABA) framework, which automatically generates assist warps to alleviate different bottlenecks in GPU execution. CABA is based on the key observation that various imbalances and bottlenecks in GPU execution leave on-chip resources, i.e., computational units, register files and on-chip memory, underutilized. We provide a detailed design and analysis of how CABA can be used to perform flexible data compression in GPUs to mitigate the memory bandwidth bottleneck. Our extensive evaluations across a variety of workloads and system configurations show that the use of CABA for memory compression significantly improves system performance (by 41.7% on average on a set of bandwidth-sensitive GPU applications) by reducing the bandwidth requirements of both the on-chip and off-chip buses. Hence, we conclude that CABA is a general substrate that can alleviate the memory bandwidth bottleneck in modern GPU systems by enabling flexible implementations of data compression algorithms. We believe CABA is a general framework that can have a wide set of use cases to mitigate many different system bottlenecks in throughput-oriented architectures, and we hope that future work explores both new uses of CABA and more efficient implementations of it.

Acknowledgments

We thank the reviewers for their valuable suggestions. We thank the members of the SAFARI group for their feedback and the stimulating research environment they provide. Special thanks to Evgeny Bolotin, Saugata Ghose and Kevin Hsieh for their feedback during various stages of this project. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, Qualcomm, VMware, and Samsung. This research was partially supported by NSF (grants 0953246, 1065112, 1205618, 1212962, 1213052, 1302225, 1302557, 1317560, 1320478, 1320531, 1409095, 1409723, 1423172, 1439021, 1439057), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation. Gennady Pekhimenko is supported in part by a Microsoft Research Fellowship. Rachata Ausavarungnirun is supported in part by the Royal Thai Government scholarship.

References

[1] T. M. Aamodt et al. Hardware support for prescient instruction prefetch. In *HPCA*, 2004.
[2] B. Abali et al. Memory Expansion Technology (MXT): Software Support and Performance. *IBM J.R.D.*, 2001.
[3] M. Abdel-Majeed et al. Warped register file: A power efficient register file for gpgpus. In *HPCA*, 2013.
[4] A. Alameldeen et al. Adaptive Cache Compression for High-Performance Processors. In *ISCA*, 2004.
[5] A. Alameldeen et al. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. Technical report, U. Wisconsin, 2004.
[6] A. Alameldeen et al. Interactions between compression and prefetching in chip multiprocessors. In *HPCA*, 2007.
[7] C. Alvarez et al. On the potential of tolerant region reuse for multimedia applications. In *ICS*, 2001.
[8] C. Alvarez et al. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 2005.
[9] C. Alvarez et al. Dynamic tolerance region computing for multimedia. *IEEE Trans. Comput.*, 2012.
[10] A. Arelakis et al. SC2: A Statistical Compression Cache Scheme. In *ISCA*, 2014.
[11] J. Arnau et al. Boosting mobile GPU performance with a decoupled access/execute fragment processor. In *ISCA*, 2012.
[12] J. Arnau et al. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *ISCA*, 2014.
[13] J. Baer et al. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 1995.
[14] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[15] M. Bauer et al. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *SC*, 2011.
[16] J. A. Brown et al. Speculative precomputation on chip multiprocessors. In *MTEAC*, 2001.
[17] M. Burtscher et al. A quantitative study of irregular programs on gpus. In *IISWC*, 2012.
[18] D. De Caro et al. High-performance special function unit for programmable 3-d graphics processors. *Trans. Cir. Sys. Part I*, 2009.
[19] R. S. Chappell et al. Simultaneous subordinate microthreading (SSMT). In *ISCA*, 1999.
[20] R. S. Chappell et al. Microarchitectural support for precomputation microthreads. In *MICRO*, 2002.
[21] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.
[22] X. Chen et al. C-pack: A high-performance microprocessor cache compression algorithm. In *IEEE Trans. on VLSI Systems*, 2010.
[23] D. Citron et al. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *ASPLOS*, 1998.
[24] J. D. Collins et al. Dynamic speculative precomputation. In *MICRO*, 2001.
[25] J. D. Collins et al. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *ISCA*, 2001.
[26] D. A. Connors et al. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO*, 1999.
[27] M. Dubois. Fighting the memory wall with assisted execution. In *CF*, 2004.
[28] M. Dubois et al. Assisted execution. Technical report, USC, 1998.
[29] J. Dusser et al. Zero-content augmented caches. In *ICS*, 2009.
[30] E. Ebrahimi et al. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*, 2009.
[31] E. Ebrahimi et al. Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *HPCA*, 2009.
[32] E. Ebrahimi et al. Prefetch-aware shared resource management for multi-core systems. *ISCA*, 2011.
[33] M. Ekman et al. A Robust Main-Memory Compression Scheme. In *ISCA-32*, 2005.
[34] J. W. C. Fu et al. Stride directed prefetching in scalar processors. In *MICRO*, 1992.
[35] M. Gebhart et al. A compile-time managed multi-level register file hierarchy. In *MICRO*, 2011.
[36] M. Gebhart et al. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *ISCA*, 2011.
[37] M. Gebhart et al. Unifying primary cache, scratch, and register file memories in a throughput processor. In *MICRO*, 2012.
[38] GPGPU-Sim v3.2.1. GPGPU-Sim Manual.
[39] B. He et al. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.
[40] J. Huang et al. Exploiting basic block value locality with block reuse. In *HPCA*, 1999.
[41] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.
[42] K. Z. Ibrahim et al. Slipstream execution mode for cmp-based multiprocessors. In *HPCA*, 2003.
[43] M. Islam et al. Zero-Value Caches: Cancelling Loads that Return Zero. In *PACT*, 2009.
[44] ITRS. International technology roadmap for semiconductors. 2011.
[45] A. Jog et al. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.
[46] A. Jog et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.
[47] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.
[48] M. Kamruzzaman et al. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. In *ASPLOS*, 2011.
[49] O. Kayiran et al. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.
[50] S. W. Keckler et al. GPUs and the future of parallel computing. *IEEE Micro*, 2011.
[51] D. Kim et al. Design and Evaluation of Compiler Algorithms for Pre-execution. In *ASPLOS*, 2002.
[52] N. Lakshminarayana et al. Spare register aware prefetching for graph algorithms on GPUs. In *HPCA*, 2014.
[53] J. Lee et al. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *MICRO*, 2010.
[54] J. Leng et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.
[55] E. Lindholm et al. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008.
[56] J. Lu et al. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor. In *MICRO*, 2005.
[57] C. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA*, 2001.
[58] J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.
[59] A. Moshovos et al. Slice-processors: An implementation of operation-based prediction. In *ICS*, 2001.
[60] V. Narasiman et al. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO*, 2011.
[61] B. S. Nordquist et al. Apparatus, system, and method for coalescing parallel memory requests, 2009. US Patent 7,492,368.
[62] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.
[63] L. Nyland et al. Systems and methods for coalescing memory accesses of parallel threads, 2011. US Patent 8,086,806.
[64] S. Palacharla et al. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, 1994.
[65] G. Pekhimenko et al. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *PACT*, 2012.
[66] G. Pekhimenko et al. Linearly Compressed Pages: A Low Complexity, Low Latency Main Memory Compression Framework. In *MICRO*, 2013.
[67] G. Pekhimenko et al. Exploiting Compressed Block Size as an Indicator of Future Reuse. In *HPCA*, 2015.
[68] T. G. Rogers et al. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.
[69] A. Roth et al. Speculative data-driven multithreading. In *HPCA*, 2001.
[70] M. Samadi et al. Sage: Self-tuning approximation for graphics engines. In *MICRO*, 2013.
[71] S. Sardashti et al. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-optimized Compressed Caching. In *MICRO*, 2013.
[72] V. Sathish et al. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *PACT*, 2012.
[73] A. Sethia et al. Apogee: adaptive prefetching on gpus for energy efficiency. In *PACT*, 2013.
[74] A. Sethia et al. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *MICRO*, 2014.
[75] A. Shafiee et al. MemZip: Exploring Unconventional Benefits from Memory Compression. In *HPCA*, 2014.
[76] B. Smith. A pipelined, shared resource MIMD computer. *Advance Computer Architecture*, 1986.
[77] A. Sodani et al. Dynamic Instruction Reuse. In *ISCA*, 1997.
[78] S. Srinath et al. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.
[79] K. Sundaramoorthy et al. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *ASPLOS*, 2000.
[80] J. E. Thornton. The CDC 6600 Project. *IEEE Annals of the History of Computing*, 1980.
[81] S. Thoziyoor et al. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, 2008.
[82] M. Thuresson et al. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans. Comput.*, 2008.
[83] N. Vijaykumar et al. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *SAFARI Technical Report No. 2015-006*, 2015.
[84] J. Yang et al. Frequent Value Compression in Data Caches. In *MICRO*, 2000.
[85] W. Zhang et al. Accelerating and adapting precomputation threads for effcient prefetching. In *HPCA*, 2007.
[86] C. Zilles et al. Execution-based Prediction Using Speculative Slices. In *ISCA*, 2001.