COMPASS: A Programmable Data Prefetcher Using Idle GPU Shaders

ABSTRACT

A traditional fixed-function graphics accelerator has evolved into a programmable general-purpose graphics processing unit over the last few years. These powerful computing cores are mainly used for accelerating graphics applications or enabling lower cost scientific computing. To further reduce the cost and form factor, an emerging trend is to integrate GPU along with the memory controllers onto the same die with the processor cores. However, given such a system-on-chip, the GPU, while occupying a substantial part of the silicon, will sit idle and contribute nothing to the overall system performance when running non-graphics workloads or applications lack of data-level parallelism. In this paper, we propose COMPASS, a compute shader-assisted data prefetching scheme, to leverage the GPU resource for improving single-threaded performance on an integrated system. By harnessing the GPU shaders with very lightweight architectural support, COMPASS can emulate the functionality of a hardware-based prefetcher using the idle GPU and successfully improve the memory performance of single-thread applications. Moreover, due to its flexibility and programmability offered by COMPASS, one can implement the best performing prefetch scheme to improve each specific application as demonstrated in this paper. With COMPASS, we envision that a future application vendor can provide a custom-designed COMPASS shader bundled with their software to be loaded at runtime to optimize the performance. Our simulation results show that COMPASS can improve the single-thread performance of memory-intensive applications by 68% on average.

1. INTRODUCTION

To meet the modern needs of game developers, a traditional fixed-function graphics accelerator has evolved into a programmable graphics processing unit (GPU), which allows game developers to write their own shader for their specific special effects. Given its vast computational capability, a GPU has been designed to run non-graphics, compute-intensive applications as well, referred to as general-purpose GPU (GPGPU) [24]. Recently, Intel and AMD announced an integrated solution to encompass the GPU, the memory controller, and the CPU onto a single die for netbook, laptop, and desktop markets [28, 36]. Although the integrated chip is not likely to be as powerful as a standalone CPU or GPU due to several reasons such as power budget, it lowers the overall system cost and reduces the form factor with reasonable performance for its particularly aimed applications and market. Furthermore, the performance can be compensated to some extent due to the substantially reduced latency between the host CPU and the integrated GPU.

Unfortunately, while the host CPU executes the sequential part of a parallelized application or an unparallelized legacy application, the integrated GPU will sit idle contributing nothing to the single-thread performance. Unlike symmetric multi-core processors in which many sequential processes can concurrently run on multiple cores, an idle GPU cannot run a conventional CPU process due mainly to the heterogeneity in the ISAs of the CPU and the GPU. Moreover, a current idle GPU cannot take advantage of speculative multi-threading or helper thread type of techniques [37, 16, 11, 8, 25, 4, 22, 29] to boost single-thread performance unless the GPU is completely re-designed to support it, which could unnecessarily complicate the entire GPU design and lead to severe performance degradation when running conventional graphics applications.

One way to improve the performance of a CPU while an on-chip GPU is idle is to exploit the remaining power budget. Because an idle GPU only consumes a small amount of idle power compared to an active GPU, the CPU can then be given the unused power by increasing its supply voltage and clock frequency, similar to the Turbo mode employed in Intel's Core i7 (Nehalem) processor [2], without exceeding the overall power envelope of the chip. Nonetheless, this method will not be able to improve the performance of memory-intensive, single-thread applications which are typically unscalable and insensitive to the clock frequency.

Instead of letting the GPU sitting idle, we envision that the OS can utilize the idle GPU to run a compute shader to enhance the memory performance of its integrated CPU. In this paper, we propose COMPASS, a **Comp**ute Shader-**Ass**isted Prefetching scheme to achieve our goal. With very lightweight architectural support, we demonstrate that COMPASS can enhance the single-thread performance of an integrated CPU by emulating a hardware prefetcher on the programmable shader.

The rest of this paper is organized as follows: Section 2 describes the details of the GPU architecture used as the baseline of this paper. Section 3 explains the general design of COMPASS, and Section 4 details the design and trade-off of various COMPASS shaders. Section 5 evaluates COMPASS in improving single-thread performance using SPEC2006. Section 6 discusses related work, and Section 7 concludes.

2. BASELINE GPU ARCHITECTURE

Figure 1(a) illustrates the baseline GPU architecture used in this paper. Because details on modern GPU architectures are not open to the public, we employed a baseline architecture that resembles an abstract GPU from several publicly available documents [1, 18, 26, 27, 34]. As shown in the figure, at the front-end of a GPU pipeline, a programmable command processor interprets command stream from a graphics driver. The command processor executes a RISC-based microcode with its computation logic and memory. Then, a setup engine prepares data for a different shader (e.g., a vertex, a fragment, and a compute shader) and submits threads of

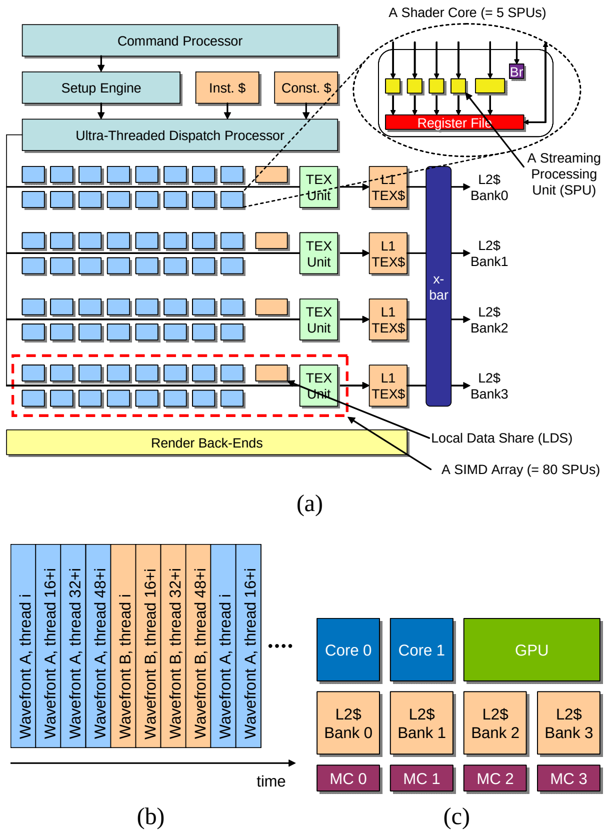

Figure 1: (a) Baseline GPU Architecture, (b) Thread Scheduling in Shader Core i, (c) Integrated Platform

each shader to an ultra-threaded dispatch processor (or a thread scheduler). The ultra-threaded dispatch processor maintains separate command queues for different shaders. It also has two arbiters for 16 cores of each SIMD array as well as an arbiter for each vertex/texture fetch unit. Furthermore, the ultra-threaded dispatch processor has a *shader instruction cache* and a *shader constant cache* to supply instructions and constant values.

The next pipeline stage of the baseline GPU executes a given shader code. As shown, the GPU consists of several SIMD arrays (four SIMD arrays in the figure), each of which consists of 16 shader cores forming a 16-way SIMD array. Each shader core is a five-way VLIW machine, each execution unit of which is referred to as a streaming processing unit (SPU). An additional branch execution unit of each shader core handles flow control and conditional operations.

In the right-most column of the SIMD array, a specialized vertex/texture unit (labeled as a TEX Unit in the figure) is connected to a vertex and a texture cache, each of which supplies requested memory values to the SIMD array. (In the figure, only a texture cache is shown for brevity.) Furthermore, 16KB local data share (LDS) is placed between the 16 shader cores and the vertex/texture unit. LDS enables efficient data sharing between threads mapped to the same SIMD array. In addition to LDS, another 16KB global data share (not shown in the figure) is present to allow data shared among different SIMD arrays, but we will not use it in this paper. Additionally, the baseline GPU has other hardware units such as a render back-end unit for color blending, alpha blending, depth testing, and stencil testing, but we do not elaborate them here as they are not essential to the main idea of this paper.

With these hardware resources, the baseline GPU is able to tolerate long cache miss latency (often in hundreds of cycles) by executing many threads alternately. Upon a cache miss of a thread, the ultra-threaded dispatch processor suspends the execution of the thread and schedules another thread to not throttle the throughput of shaders due to miss latency. To achieve this, the ultra-threaded dispatch processor forms a group of 64 threads and uses this group as a *thread scheduling unit*. It essentially dispatches a group of 64 threads to the SIMD array simultaneously and later dispatches another group of 64 threads upon a cache miss of the previous group. This group is referred to as a *wavefront* (or a *warp* in NVIDIA terminology [14]). A wavefront (64 threads) executes one VLIW bundle on a 16-way SIMD array over four cycles as shown in Figure 1(b). In other words, one VLIW bundle of the first 16 threads is dispatched to the SIMD array at cycle $4n$, the same bundle of the next 16 threads is dispatched to the SIMD array at cycle $4n+1$, and so on. As the ultra-threaded dispatch processor has two arbiters per SIMD array, two wavefronts compete to dispatch their instructions to the SIMD array. In this paper, we assume that the same wavefront can be dispatched only after another wavefront is dispatched as shown in Figure 1(b).

To support such a large number of threads on four 16-way SIMD arrays, the GPU requires large register files. Following a speech from AMD [34], we assume that the capacity of the register file of each SIMD array is split into 256 sets, each consisting of 64 128-bit registers. It amounts to a total of 256KB ($256 \times 64 \times 16$) register space evenly split across 16 shader cores of each SIMD array. The total capacity of register files can vary according to the target market or over different GPU generations. For different market segments, the GPU industry used to design its products with a different number of shader cores. For example, ATI Radeon HD 4890 (for the enthusiast gamer market) consists of ten 16-way SIMD arrays while ATI Radeon HD 4600 (for the mainstream market) contains only four 16-way SIMD arrays. In this paper, we employ a baseline GPU of four 16-way SIMD arrays for the integrated chip unless otherwise mentioned. Considering the number of SIMD arrays will continue to soar in the future GPU, our results based on this assumption of four SIMD arrays can be considered conservative.

Such a large pool of registers is shared by threads executed on the same SIMD array. It also implies that more registers each thread uses, less threads can be simultaneously active. Such register partitioning is managed by GPU itself using a relative indexing scheme [1]. Because the details on register partitioning is not disclosed, we assume a simple indexing scheme— a global register index is the sum of a base register of a thread and a relative register index within the thread. For example, if we have a pool of 32 registers and if each thread uses four registers, only eight threads can be active simultaneously. In this example, the global index of a physical register is calculated by adding the register identifier to the base register index of the thread, which can be calculated by simply shifting the thread ID to the right by two.

Lastly, in this paper, we assume that the integrated CPU cores and the GPU share four banks of the L2 cache as shown in Figure 1(c). Such a proposal has been considered in an early, failed integrated solution (e.g., Intel's Timna processor) and, to our firm belief, will re-emerge in the future integrated chips for the following reasons: (1) saving the overall cost, (2) providing efficient and coherent communication between the host and the accelerator, and (3) giving more flexibility in cache space consumption, similar to the rationale of the Advanced Smart Cache in Intel's multi-core processors[1]. In addition, we also assume that memory controllers

[1]Even in the case of a separate L2 cache, our proposed mechanism can still be used if small circuits that forward a prefetch address from the GPU to the private L2 cache of CPUs is implemented.

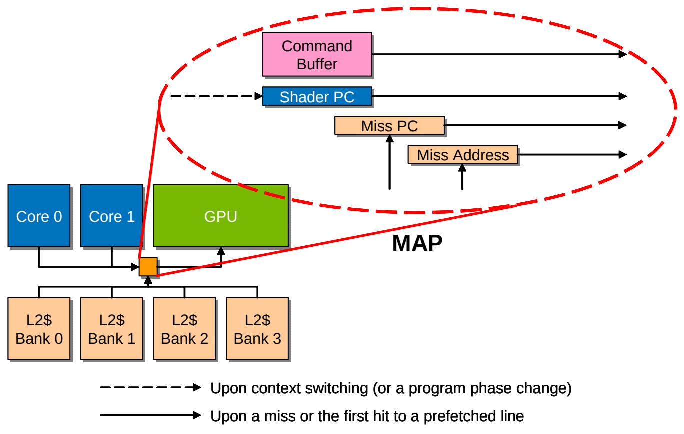

Figure 2: Miss Address Provider

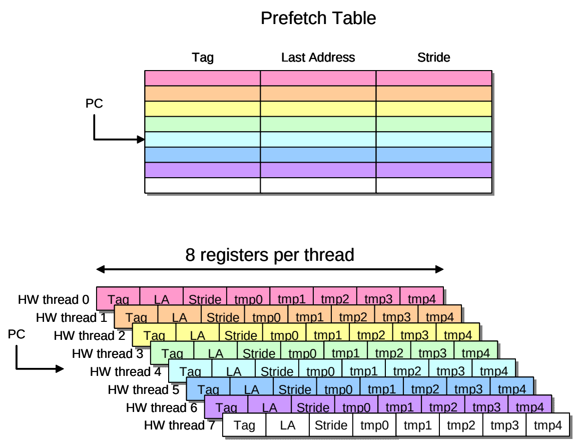

Figure 3: Hardware Prefetcher vs. COMPASS

are integrated on-die as they are already in a current discrete GPU product.

3. COMPASS

Although data prefetching techniques have been widely researched, state-of-the-art commercial processors only either employ simple schemes such as a next-line prefetcher [3] or a stride prefetcher [39], or push it to the software side by providing prefetch instructions and let the compilers or programmers insert them at appropriate locations inside the code. Most of the advanced complex hardware schemes remain in literature due to their prohibitively expensive hardware cost [17, 19, 30, 38]. Furthermore, different applications may favor different types of prefetchers. Thus, a one-size-fits-all hardware-based prefetcher may not be in the best of interests of all applications. More importantly, the GPU consumes enormous memory bandwidth when active, which competes the same bandwidth shared by a dedicated hardware prefetcher for the CPU. Such scenarios, in effect, will deteriorate the performance of both the CPU and the GPU, diminishing the entire purpose of GPU acceleration.

To ameliorate the shortcomings of prior art, in this paper, we propose COMPASS, a compute-shader assisted prefetching scheme, which uses the idle GPU to achieve the functionality of hardware prefetchers for improving the single-thread performance of an integrated single-chip system. The rationale behind our design is as follows: (1) An on-chip GPU has large register files and rich computational logic, so it can emulate the behavior of hardware prefetchers, (2) While a GPU is idle, much of memory bandwidth originally designed to meet the requirement of a GPU, will be left unused and can be re-harnessed to assist the CPU cores, in particular, for data prefetching, and (3) The tight coupling of the CPU and the GPU on a single die facilitates efficient, prompt communication leading to synergistic outcome.

Note that one of COMPASS design goals is to reuse existing hardware as much as possible, minimizing the overall hardware overheads. Also, the overheads incurred by COMPASS should not affect and compromise the massive parallel computation capability in a GPU originally designed for 3D rendering and high-performance computing. Toward these objectives, we describe how to emulate different types of data prefetchers in subsequent sections.

3.1 Miss Address Provider

Because COMPASS is based on a compute shader, which is programmable, it provides much more flexibility than conventional hardware prefetchers. As such, instead of implementing COMPASS with a fully automatic hardware mechanism, we opted for an OS or application vendors to offer and enable COMPASS prefetch capabilities. In this respect, we propose to add the *Miss Address Provider* (MAP), a hardware/software interface bridging the L2 cache, the GPU, and the OS. MAP is located between the L2 cache and the GPU as shown in Figure 2.

The operation of the MAP is described step-by-step in the following. (1) Once the OS has no job to run on the GPU, the OS provides MAP a pointer to a compute shader for prefetching, referred to as a COMPASS shader. (2) Upon an L2 miss or the first hit to a prefetched line[2], a program counter (PC) that has generated this memory request and the physical address of this request are forwarded to MAP for prefetching. (3) Upon receiving these two values, MAP sends a GPU command that triggers the execution of a compute shader. These two values, a PC and a physical address, are stored in the constant cache and read by the COMPASS shader. Furthermore, the command also indicates which value, the PC or the physical address, should be used to index a thread. (More details will be explained in Section 3.2.) (4) The role of the COMPASS shader is to read miss history from GPU's register files to predict the subsequent miss addresses, to update history information, and to execute prefetch instructions that bring data back to the L2. (5) Prior to a new job to be scheduled onto the GPU by the OS, MAP will be disabled. Note that the OS intervenes in COMPASS only when it needs to enable or disable a COMPASS shader upon context switching. The OS can change a COMPASS shader more often, e.g., upon a program phase change, but such fine-grained execution is outside the scope of this paper.

3.2 Threads and Register Files

A COMPASS shader that is triggered by MAP emulates a hardware prefetcher as follows. Basically, we emulate each entry of a prefetch table with one GPU thread (or with a set of GPU threads when using a multi-threaded COMPASS shader). For example, as shown in Figure 3, GPU thread4 emulates the behavior of a prefetch operation that accesses the 4th entry of the prefetch table. In other words, registers of a GPU thread (or a set of GPU threads) are used to record miss history information to emulate the same recording in a hardware prefetch table.

To allow such mapping, we assign a hardware thread ID using the same index function as the index function of a hardware

[2]Here, as in many previous prefetch studies, we assume that an L2 cache has a tag bit per cache line for prefetch. It is set when a line is prefetched in but not consumed.

prefetcher. Such thread mapping circuit is not currently implemented in the setup engine of the GPU architecture. Clearly, this is an additional hardware overhead, but this overhead is rather insignificant for the indexing requires only simple masking.

Because a typical hardware prefetcher table has a power-of-two entries to facilitate PC indexing, we also force the number of concurrently active hardware threads to be in power-of-two. Also, we wrote our COMPASS shader to always use power-of-two registers. With this simple trick, we do not need to modify the register partitioning hardware of the baseline GPU, but the requirement of using power-of-two registers is clearly one source of storage inefficiency, preventing us from emulating a larger prefetch table.

Those registers allocated to a thread (or a set of threads in a multi-threaded COMPASS shader) are used for two different purposes. One group of registers store miss history information of each table entry, and the others are used as temporary registers for calculating next prefetch addresses and for updating miss history. For example, to store miss history of a stride prefetcher, each thread uses three registers to keep a tag value, a last address value, and a stride value, corresponding to an entry of a conventional stride prefetcher (Figure 3). To calculate the following prefetch addresses or to update the emulated table, the thread needs several additional temporary registers.

3.3 An Example of a COMPASS Shader

Figure 4 shows an example shader code for stride prefetching. This shader code uses three general-purpose registers to store a tag, a last miss address, and a stride value while using one more register as a temporary register. Two other values, a current PC and a current miss address, are provided by the MAP in the form of constant values stored in a constant cache.

```
if $tag == currentPC$ then
    $tmp \leftarrow lastAddr + stride$
    if $currentAddr == tmp$ then
        $tmp \leftarrow currentAddr + stride$
        $prefetch \quad tmp$
    end if
end if
$tag \leftarrow currentPC$
$stride \leftarrow currentAddr - lastAddr$
$lastAddr \leftarrow currentAddr$
```

Figure 4: Stride Prefetching

As shown in the code, one run of a COMPASS shader emulates one lookup of a conventional hardware prefetcher. Such design necessitates the registers keeping the miss history information to remain unmodified until the thread that emulates the same prefetch table entry is dispatched. Such integrity of register files is enforced by the setup engine, which uses the same index function of a hardware prefetcher to allocate a hardware thread ID to each COMPASS lookup as we explained previously.

3.4 A Prefetch Instruction

To perform a prefetching operation, we can use a conventional load instruction of a GPU. However, the load instruction requires a destination register, which consumes one more register per prefetch request. As explained previously, more registers each thread uses, a less number of threads can be executed simultaneously. In other words, using a load instruction reduces the capacity of a prefetch table being emulated by a COMPASS shader. Hence, to reduce the number of required registers in each thread, we propose to add a prefetch instruction in the GPU. (Note that a GPU is unlikely to support a prefetch instruction given cache misses can always be hidden by the execution of a plethora of independent threads.) Implementing a prefetch instruction does not incur much hardware overhead because the prefetch instruction can be supported by disabling the write-back path of a load instruction.

3.5 Usage Model

We design COMPASS to be executed by an OS. Through profiling, the OS can select the best matched COMPASS shader and enables it by setting a pointer to the selected COMPASS shader. On the other hand, the OS can also provide an API call to allow an individual application to select an appropriate COMPASS shader from a COMPASS library provided by the OS. Furthermore, an application vendor can provide an application-specific COMPASS shader through another API call. This API call provides the OS a pointer to their custom COMPASS shader. When the OS schedules the target application, the OS can read this pointer value and store the pointer in the shader PC register of the MAP (Figure 2). This execution model implies that even if the application has nothing to do with graphics rendering, the application developers can still use the GPU to enhance its performance by writing a prefetcher shader and having it loaded by the runtime system.

4. DIFFERENT COMPASS SHADER DESIGNS

In this section, we describe different COMPASS shader designs and analyze their design trade-offs. First of all, we evaluate three table-based prefetchers: a PC-indexed stride prefetcher [6], a Markov prefetcher [19], and a PC-indexed delta correlation prefetcher [20, 30]. On the other hand, we also evaluate a region prefetching technique [23]. In addition to these generic prefetchers, to demonstrate the unique feature of COMPASS, we also design and evaluate a custom-designed prefetcher for 429.mcf, which is known to be memory-bound. As mentioned, an application provider can write its own custom COMPASS prefetcher and pass it to the OS via an API call to optimize performance.

One issue to be addressed is the long latency of instructions in a GPU shader. Notice that the design principle of GPUs is to optimize for high throughput. They exploit a large amount of thread-level parallelism to hide the instruction latency. As illustrated in Figure 1(b), if read-after-write dependency exists between two successive instructions of a COMPASS shader, the dependent instruction will be dispatched eight GPU cycles after its producer instruction is dispatched. This is certainly not a performance bottleneck for throughput-oriented graphics rendering algorithms, it is, however, clearly a performance issue for a data prefetcher to bring in missing cache lines in advance. On the other hand, a GPU clock can be slower than a CPU clock ($\frac{1}{2}$ CPU frequency assumed in this paper), albeit the integrated design may be able to bring the GPU up to the CPU's speed. In our more pessimistic assumption, we could waste 16 CPU cycles between two successive VLIW bundles of a COMPASS shader.

On the other hand, one limitation of COMPASS is its throughput. COMPASS is based on a compute-shader, which needs tens of instructions to emulate a hardware prefetcher. Before a shader completes its execution, another shader mapped to the same hardware thread ID cannot be dispatched. Furthermore, a GPU may not have enough command queue to have many different shaders. (Note that this is not a queue for threads spawned from a single shader execution command, but a queue for different shader execution commands.) The maximal number of shaders that can be queued in our baseline GPU is 16. Once this queue is full, the shader execution command from MAP is ignored until the queue releases a slot for a new shader execution. These two issues, if left unaddressed, will make the idea of COMPASS less useful. We will focus on them in describing our shaders.

4.1 Stride COMPASS

In the implementation of PC-indexed stride COMPASS, we use three registers to keep a PC tag, its associated last miss address and stride. Upon a cache miss, the setup engine uses a PC given by MAP to generate an index, which is used as a hardware thread ID for our COMPASS shader. Once a thread is selected, it compares whether a requested PC matches to the tag value and whether a current miss address is equivalent to a previously predicted miss address. If both conditions are met, the shader generates next d prefetch addresses where d is the prefetch depth[3]. To avoid branch instructions, we use predication for condition checking and unroll the loop to generate d prefetch addresses.

We have two different types of stride COMPASS shaders to trigger d prefetches. One is a single-threaded stride COMPASS, which activates only one thread upon a miss. In this case, each entry of the prefetch table is modeled with one thread. This thread consumes three registers to maintain miss history and generates d prefetches itself. The utilization of the GPU that runs this shader is very low since we only enable one out of 64 threads in a wavefront. The second design is a multi-threaded stride COMPASS, which activates d threads within a wavefront. Although these d threads emulate just one prefetch table entry, the same miss history (e.g., tag, last miss address and stride stored in three registers) needs to be duplicated for each thread. Since each thread only generates one prefetch request, it requires only one temporary register for the prefetch address of the thread compared to d temporary registers of a single-threaded stride COMPASS shader. Thus, the number of temporary registers required per thread is smaller, and the multi-threaded stride COMPASS shader can be completed sooner, thereby reducing the latency and increasing the throughput. When d is not in power-of-two, we made a group of D threads to emulate a prefetch table entry where D is the smallest number in power-of-two greater than d. For example, to emulate a stride prefetcher with a prefetch depth of five, we group eight threads into one group so that this group can emulate a prefetch table entry. In this case, three remaining threads are never activated. Such inactive threads are found to be another source of storage inefficiency.

4.2 Markov COMPASS

In contrast to a PC-indexed stride prefetcher, a Markov prefetcher [19] uses a miss address to index a prefetch table. Thus, the setup engine uses a current miss address to index a thread (for prefetching) and the last miss address to index another thread (for updating the table) (Figure 5). Here, the command processor stores the miss address of the last execution and provide it as the last miss address to the setup engine for updating the table upon receiving a new miss address from the MAP. Each thread of our Markov COMPASS shader maintains w next addresses, where w denotes the prefetch width. Our Markov COMPASS maintains these w next addresses in a FIFO manner. One advantage of Markov COMPASS over a hardware Markov prefetcher is its programmability for using different prefetch width. For example, an application that heavily uses a binary tree favors a Markov prefetcher with a prefetch width of two (or three if a pointer to a parent node is required). On the other hand, another application that heavily uses a singly-linked list favors a Markov prefetcher with a prefetch width of one because reducing the size of each entry allows more entries to be emulated. Therefore, an application vendor can configure (even dynamically) the appropriate prefetch width of their own COMPASS shader according to the behavior of their specific algorithms.

```
if $tag == currentAddr$ then                         ▷ Prefetch shader
    $prefetch \quad nextAddr0$
    $prefetch \quad nextAddr1$
    $prefetch \quad nextAddr2$
end if
$tag \leftarrow currentAddr$
```

```
$nextAddr2 \leftarrow nextAddr1$                     ▷ State update shader
$nextAddr1 \leftarrow nextAddr0$
$nextAddr0 \leftarrow currentAddr$
```

Figure 5: Markov Prefetching (Prefetch Width: 3) (The prefetch shader is indexed with a current miss address while the state update shader is indexed with the last miss address.)

4.3 Delta COMPASS

The third prefetcher we evaluated is a PC-indexed delta prefetcher [20, 30]. The pseudo code is depicted in Figure 6. As shown, implementing delta COMPASS is found to be more challenging because the process of delta correlation matching is complicated and we have to accumulate delta values to calculate prefetch addresses. Considering that a minimum interval for dispatching two successive VLIW bundles of a thread is eight GPU clock cycles, the latency of delta COMPASS implemented with a single thread will be very high.

```
for $i \leftarrow depth - 1$ to 1 do                          ▷ State update
    $\delta_i \leftarrow \delta_{i-1}$
end for
$\delta_0 \leftarrow currentAddr - lastAddr$
$lastAddr \leftarrow currentAddr$

for $i \leftarrow depth - 1$ to 2 do              ▷ Delta correlation matching
    if $(\delta_0 == \delta_i) \&\& (\delta_1 == \delta_{i+1})$ then
        $break$
    end if
end for

$prefAddr \leftarrow currentAddr$            ▷ Prefetch address calculation
for $j \leftarrow i - 1$ to 0 do
    $prefAddr \leftarrow prefAddr + \delta_j$
    $prefetch \quad prefAddr$
end for
```

Figure 6: Delta Prefetching (δ_i: i^{th} entry of a delta buffer)

To improve its efficiency, we implemented multi-threaded delta COMPASS. For example, to emulate a delta prefetcher with eight delta buffers, we use eight threads; each has a part of the delta buffer and performs delta correlation matching in parallel. In particular, thread i ($0 \leq i \leq 7$) keeps δ_0, δ_1, δ_i, and δ_{i+1} where δ_n is the n^{th} entry of the delta buffer. To update the state, thread i passes δ_{i+1} to thread $(i+1)$ through LDS (Figure 1(a)), so that the delta buffer can be synchronized globally. To perform correlation matching, thread i compares a pair of δ_0 and δ_1 against a pair of δ_i and δ_{i+1}. Once a thread finds a match, it broadcasts its thread ID[4].

However, calculating a prefetch address in each thread is still challenging because thread j ($j < i$) requires to perform a reduction of $\sum_{n=j}^{i-1} \delta_n$, which should be accumulated after finding a match. This accumulation requires sequential scanning among threads, leading to elongated latency and lowered throughput. To

[3]In this paper, we follow the prefetching terminology used in [30]. For example, a prefetcher with a prefetch depth of four prefetches four cache lines that will likely be requested by four consecutive misses in the future. On the other hand, a prefetcher with a prefetch width of four prefetches four potential cache line candidates that will likely be requested by the next miss.

[4]Multiple threads may find a match when the length of a correlation sequence is short, but here we use a special broadcast instruction that broadcasts a value from the first valid thread in a wavefront to all threads of the wavefront [1].

Table 1: A Modified Prefetch Address Calculation Algorithm (Miss Address = 1024)

	thread 0	thread 1	thread 2	thread 3	thread 4	thread 5	thread 6	thread 7
Delta (δ_j)	1	2	1	2	1	2	3	4
Delta Sum ($\sum_{n=0}^{j-1} \delta_n$)	0	1	3	4	6	7	9	12
Match?	O	X	O	X	O ($i = 4$)	X	X	X
$\sum_{n=j}^{i-1} \delta_n$	6	5	3	2	disabled	disabled	disabled	disabled
Prefetch Address	1030	1029	1027	1026	disabled	disabled	disabled	disabled

Table 2: A Sample Stride Pattern of 429.mcf (Cache Line Size: 64B)

Miss Address	0x4b19ba40	0x4b19e080	0x4b1a2d40	0x4b1ac680	0x4b1bf900	0x4b1e5e40	0x4b2328c0	0x4b2cbd80
Stride (δ_j)		9792	19648	39232	78464	156992	313984	627904
$\delta_j - 2 \times \delta_{j-1}$			64	-64	0	64	0	-64

avoid this loop, we contrive the delta COMPASS shader so that thread j also maintain another value, $\sum_{n=0}^{j-1} \delta_n$, which we call delta sum. Basically, delta sum is a sum of the last j delta values. Thread j can easily maintain this value by accumulating $(\delta_0 - \delta_j)$ upon a miss or upon the first hit to the prefetched line. Once thread i finds a correlation match, this thread broadcasts its own delta sum, $\sum_{n=0}^{i-1} \delta_n$, along with its thread ID. Once thread j receives this value, it can calculate $\sum_{n=j}^{i-1} \delta_n$ by subtracting its own delta sum, $\sum_{n=0}^{j-1} \delta_n$, from broadcast thread i's delta sum, $\sum_{n=0}^{i-1} \delta_n$. With this modified algorithm, we can eliminate the iterative accumulation process from each shader execution.

A detailed example is shown in Table 1. If we use the original algorithm (Figure 6), thread4 finds a match and broadcasts its thread ID to all other threads. After receiving the thread ID of four, thread1, for example, needs to accumulate the delta value of thread3 (= 2), that of thread2 (= 1), and that of thread1 (= 2). The sum of these values, 5 (= 2 + 1 + 2), are added to a current miss address, 1024, to calculate a prefetch address, 1029. On the other hand, in our modified algorithm, each thread keeps updating the delta sum when it updates its local delta values. When thread4 finds a match, it broadcasts its thread ID and its own delta sum (= 6) to all other threads. Upon receiving this value, thread1 subtracts its own delta sum (= 1) from the receive delta sum (= 6) and adds this difference (= 5) to the miss address, 1024, to calculate the prefetch address, 1029.

4.4 A Simplified Region Prefetcher

```
1: $pageAddr \leftarrow currentAddr$ & 0xfffff000
2: for $i \leftarrow 0$ to 63 do
3:     $prefAddr \leftarrow pageAddr + 64 \times i$
4:     $prefetch \quad prefeAddr$
5: end for
```

Figure 7: Region Prefetching

Additionally, we also evaluated a simplified version of a region prefetcher [23]. Although the original region prefetching technique monitors the utilization of a memory channel and prefetches an entire page while the channel is idle, we cannot perform such fine-grained monitoring and prefetching because we do not want to heavily modify the GPU design. Thus, in this paper, we simplified the design by fetching a page upon the first touch of a certain page. Basically, the setup engine uses a miss address to index a group of 64 threads or a wavefront. In particular, as shown in Figure 7, thread i of a wavefront prefetches $P + 64 \times i$ where P denotes the base address of a page to which a requested miss belongs. In other words, 64 threads of the wavefront prefetches 64 cache lines that belongs to the same page. (In this paper, the size of the L2 cache line is 64B, and that of a page is 4KB.) Such a brute-force, non-controlled prefetching technique cannot be used in a conventional prefetcher, which will affect the performance of all applications. However, due to its flexibility, COMPASS shader enables such an aggressive technique whenever an algorithm has a demand for.

4.5 Custom COMPASS Design for 429.mcf

To demonstrate and evaluate a custom COMPASS shader, we selected 429.mcf from SPEC2006 for our case study. It is known that 429.mcf demonstrated a peculiar memory access pattern [10, 35]. The address strides of several PCs missing the L2 cache in 429.mcf are increased exponentially as shown in Table 2 making them hard to be recognized by most of the hardware prefetchers. To capture the exponential stride pattern, we designed a COMPASS shader that performs exponential stride prefetching. The pseudo code is shown in Figure 8. Our exponential stride prefetcher maintains the last miss address and the last stride value. If a current stride value meets the condition shown in line 1 of the figure, we prefetches multiple cache lines using exponential strides. We multi-threaded this COMPASS shader so that thread i can fetch three lines that are fetched by iteration i of a loop shown in Figure 8. To reduce the latency of this computation, we avoid the accumulation process in this loop by designing thread i to compute the accumulated sum of exponential strides by itself. Basically, instead of calculating $\Sigma_{n=1}^{i} 2^n \delta_0$, each thread calculates $2 \times (2^i - 1)\delta_0$ which is equivalent to the previous equation, where δ_0 is a current stride value.

Note that, prior research [35] had attempted to capture such access patterns in a hardware prefetcher, yet often resulting in prohibitively large hardware structure, thereby rendering impractical. Again, the programmable COMPASS shader will enable such prefetching in an economical manner.

```
1: if $(\delta_0 \geq 2\delta_1 - 64)\&\&(\delta_0 \leq 2\delta_1 + 64)$ then
2:     $stride \leftarrow \delta_0$
3:     $prefAddr \leftarrow missAddr$
4:     for $i \leftarrow 0$ to $depth - 1$ do
5:         $stride \leftarrow stride \times 2$
6:         $prefAddr \leftarrow prefAddr + stride$
7:
8:         $prefetch \quad prefAddr$
9:         $prefetch \quad prefAddr + 64$
10:        $prefetch \quad prefAddr - 64$
11:    end for
12: end if
```

Figure 8: Exponential Stride Prefetching (δ_0: a Current Stride, δ_1: the Last Stride)

5. EXPERIMENTAL RESULTS

In this section, we will first describe our simulation framework and evaluate each COMPASS design. For each design, we will

Table 3: Platform Configurations

CPUs	Processor model	Two 3.0GHz, 14-stage, out-of-order, 4-wide fetch/issue/retire superscalar processors 192 ROB entries, 128 (INT) + 128 (FP) physical register file
	Branch predictor	Hybrid branch predictor (16K global / local / meta tables), 2K BTB, 32-entry RAS
	L1 instruction cache	dual-port 2-way set-associative, 64B-line, 32KB cache, LRU policy, 1 cycle latency, 1 cycle throughput, 8-entry MSHR
	L1 data cache	dual-port 4-way set-associative, 64B-line, 32KB write-back cache, LRU policy, 2 cycle latency, 1 cycle throughput, 8-entry MSHR
GPU	Clock frequency	1.5 GHz
	Front-End	Command Processor (4 GPU cycle latency) + Setup Engine (3 GPU cycle latency)
	Ultra-Threaded Dispatch Processor	2 arbiters per SIMD array, 1 arbiter per TEX unit, minimum 1 GPU cycle latency, FIFO scheduling policy
	Array Size	4 SIMD arrays, 16 shader cores per SIMD array, 5 SPUs per shader core
	Register file	4-banked, 16KB per SIMD array
	TEX unit	One per SIMD array, 4 address processors per TEX unit
	TEX L1 Cache	1 cycle latency, 4-way set-associative, 64B-line 32KB (tightly coupled with a TEX unit)
Memory Back-End	Shared L2 cache	dual-port, 4-banked, 8-way set-associative, 64B-line, 2MB inclusive write-back cache, LRU policy, 6 cycle latency, 1 cycle throughput, 8-entry MSHR per bank
	Baseline prefetcher	Next-line prefetcher in the L2 cache
	Memory	350-cycle minimum latency, four 8B-wide buses, 800 MHz clock, double-data-rate

employ one or two representative applications for an in-depth analysis in the design trade-offs. After that, we will show overall results with our benchmark applications. Lastly, we will discuss hardware, software, and power overhead of our COMPASS.

5.1 Simulation Framework

We evaluate COMPASS by extending the SESC simulator [33]. In particular, we use an existing CPU model of SESC as well as its memory back-end. In addition, we integrated a GPU pipeline to perform cycle-level simulation. Figure 1(c) illustrates the overall system architecture of our CPU-GPU integrated platform and its details are listed in Table 3. To quantify the performance advantage of COMPASS, we use memory-intensive applications from SPEC2006. We define a memory-intensive application as an application whose speedup with a perfect L2 cache is greater than 1.1x compared to a baseline CPU with a next-line prefetcher. For computation-intensive applications, as COMPASS is fully programmable, an OS can opt for not using it. As such, COMPASS will not adversely affect the performance for these applications. We also excluded 434.zeusmp, 465.tonto, and 470.lbm due to cross-compilation issue or unsupported syscalls in SESC. For each experiment, we fast-forwarded the first 10 billion instructions and measured the performance for the next billion instructions unless otherwise mentioned. Throughout this paper, the baseline has a next-line prefetcher associated with its L2 cache.

5.2 Evaluation of Stride COMPASS

To evaluate the stride COMPASS, we performed a vast amount of simulations with different types of shaders and different prefetch depth. First of all, Figure 9(a) and Figure 9(b) show the number of required registers per thread and the number of threads required for simulating one prefetch table entry, respectively. As shown, as the prefetching depth increases, a single-threaded (ST) stride COMPASS shader requires more registers per thread. On the other hand, a multi-threaded (MT) stride COMPASS shader requires more threads to emulate one entry while the number of required registers per thread remains constant. This trade-off results in different numbers of table entries that can be emulated with COMPASS as shown in Figure 9(c). The figure suggests that, in general, an ST stride COMPASS shader can emulate more table entries than an MT stride COMPASS shader. Note that both stride COMPASS shaders can emulate at least 4096 entries. Assuming that an entry of a hardware-based stride prefetcher table is 12B wide (including a tag, a last address, and a stride value), our stride COMPASS with four SIMD arrays can emulate the behavior of a 48KB hardware stride prefetch table.

Although both of stride COMPASS shaders, ST and MT, emulate a larger stride prefetcher than a typical hardware-based counterpart, their performance may not be as high due to the longer latency and lower throughput. To observe this effect, we performed another set of simulations with a zero-latency, infinite-throughput (ZI) model that has the same number of table entries as the MT stride COMPASS. Figure 9(d) shows the speedup of these three models running 459.GemsFDTD. As can be seen, both ST and MT models underperform slightly but will catch up with the ZI model if their prefetching depth is deep enough. This can be explained by analyzing Figure 9(e), which shows the average latencies of ST and MT models, and Figure 9(f), which shows the number of discarded requests due to queue overflow (explained in Section 4). Here, we do not show the latency and the number of discarded requests for the ZI model since both of them are zero by the definition of the ZI model. As shown in these figures, when a GPU emulates a stride prefetcher for 459.GemsFDTD, the latency of the COMPASS shader is reasonably low so that an ST or an MT COMPASS shader that fetches more deeply can catch up with the ZI model. Besides, the queue occupied by pending miss requests hardly overflows, thus lost miss history due to discarded requests does not affect the overall performance of 459.GemsFDTD significantly.

In contrast, when running 462.libquantum, we found that the ST and the MT stride COMPASS shader cannot achieve the performance of the ZI model as illustrated in Figure 9(g). Moreover, the ST stride COMPASS shader performs worse than the MT stride COMPASS shader because of its long latency even though there is no hurdle for the ST COMPASS shader to emulate a large prefetch table (Figure 9(h)). More interestingly, once the prefetch depth of the MT stride COMPASS shader exceeds nine, its improvement levels off (Figure 9(g)) with a continuing increased latency due to a longer queuing delay within the GPU (Figure 9(h)). Furthermore, as shown in Figure 9(i), not a single miss request was discarded due to overflow. After analyzing the simulation trace, we found that a *single PC* of 462.libquantum constantly generates a cache miss. Consequently, our PC-indexed stride COMPASS shader is serialized, failing to utilize four SIMD arrays or failing to utilize other threads available in the same SIMD array. The following

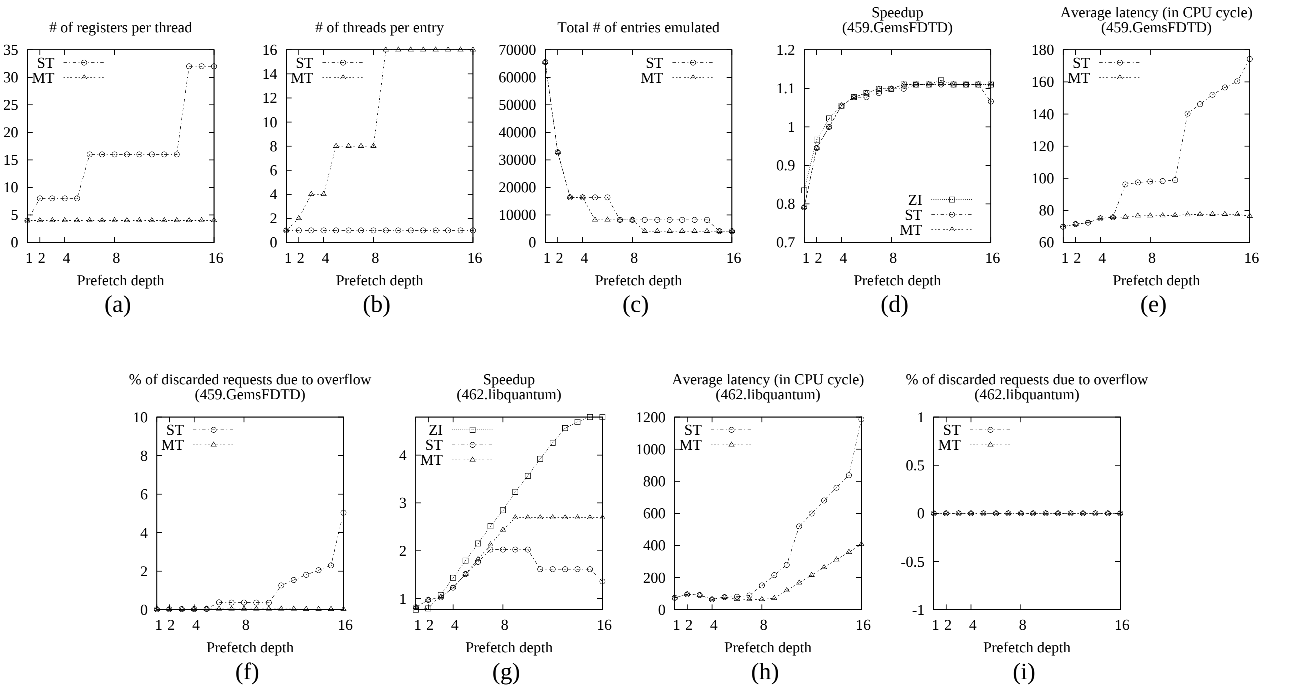

Figure 9: Evaluation of Stride COMPASS (ST: single-threaded, MT: multi-threaded, ZI: zero-latency, infinite-throughput)

analysis quantitatively explains this phenomenon well. First of all, we found that the saturated IPC is 1.05, and the corresponding L2 misses per kilo instructions (MPKI) is 19.77. In other words, it takes 952.38 cycles to execute 1000 instructions that generate 19.77 misses. Thus, the average time interval between two successive misses is 48.17 cycles while it takes 48 cycles to execute three VLIW bundles for our stride COMPASS shader. In short, our COMPASS shader cannot prefetch cache lines timely enough once the IPC of 462.libquantum reaches 1.05.

To mitigate this problem, we may want to use a different indexing function, although this is not evaluated in this paper. For example, instead of using a PC, we can concatenate a PC with two bits from a cache miss address (bit 7 and bit 6 of the miss address while six LSBs are cache line offset bits.) to make a new indexing function. As use four different entries for a single PC, the stride of each entry will be 256, instead of 64 (when we used the conventional indexing function). Instead of designing a prefetcher with the prefetch depth of 16, we can make these four entries to prefetch four cache lines in advance only resulting in a functionally equivalent prefetcher.

Another noteworthy function is that an L1 TEX cache attached to each SIMD array can combine or collapse redundant prefetch requests issued by COMPASS. For the MT stride COMPASS with a prefetch depth of 16 for 462.libquantum, the total number of L1 TEX cache accesses is 316.3 million while the total number of actual prefetches that reached the L2 is 19.8 million.

5.3 Evaluation of Markov COMPASS

In contrast to the stride COMPASS which we varied its prefetch depth, we evaluate the Markov COMPASS shaders by varying their prefetching width. Because a Markov COMPASS shader is very simple, we only evaluate a single-threaded Markov prefetcher. The cost of emulating those different Markov prefetchers are shown in Figure 10(a) and Figure 10(b), in which the number of required registers and the capacity of the emulated Markov prefetch table are plotted with the prefetch width. Due to the fact that the number of registers or the number of entries must be in power-of-two for indexibility, a Markov prefetcher with a prefetch width of two will contain unused resources while consuming the same resources with the one with a prefetch width of three as shown in Figure 10(b). In our experiments, a Markov prefetcher with a prefetch width of three turns out to be the most effective for two applications: 471.omnetpp and 483.xalacbmk that benefit from a Markov prefetcher. For 471.omnetpp shown in Figure 10(c), a Markov COMPASS shader with a prefetch width of one did not outperform the cases with wider prefetch width although it can emulate more table entries. On the other hand, a Markov COMPASS shader with a prefetch width of four reduces the speedup since the number of table entries that can be emulated has significantly dropped. The average latencies of these Markov COMPASS with different prefetch width are found to be similar as shown in Figure 10(d).

When comparing the capability of our Markov COMPASS to a conventional hardware-based prefetcher, our scheme has a huge advantage. For example, a Markov COMPASS with a prefetch width of three emulates 65,536 table entries, each containing 16B (a tag plus three next miss addresses.) That amounts to a 1MB prefetch table if implemented in a hardware-based prefetcher, too overwhelmingly large for a prefetch table.

5.4 Evaluation of Delta COMPASS

As explained in Section 4.3, we evaluate only a multi-threaded delta COMPASS shader due to the computation complexity of a delta prefetcher. In our delta COMPASS implementation, each thread uses 16 registers. The number of required registers to emulate a delta prefetcher and the number of table entries that can be emulated with delta COMPASS are shown in Figure 11(a) and Figure 11(b), respectively. As shown in Figure 11(b), the capacity of emulated prefetch tables is much larger (at least 1024 entries)

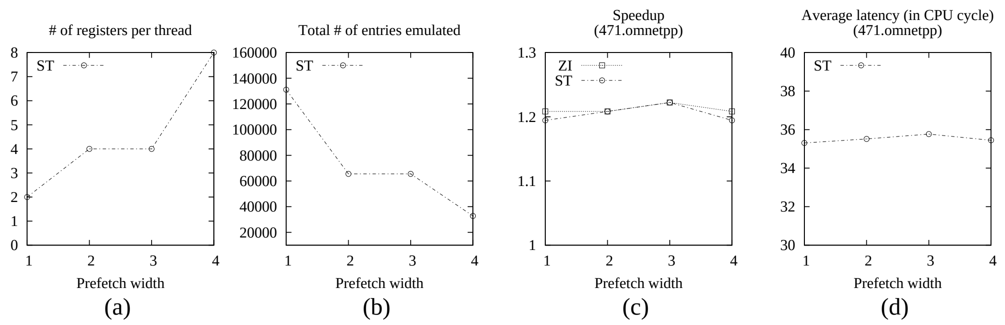

Figure 10: Evaluation of Markov COMPASS (ST: single-threaded, ZI: zero-latency, infinite-throughput)

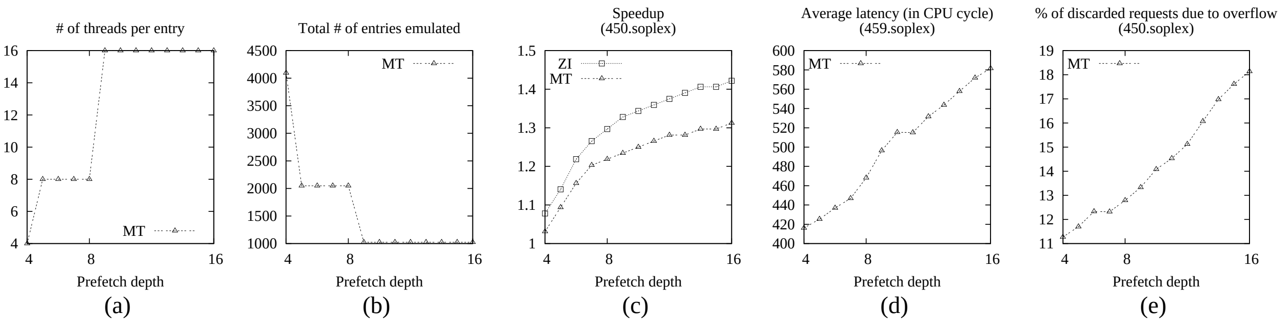

Figure 11: Evaluation of Delta COMPASS (MT: multi-threaded, ZI: zero-latency, infinite-throughput)

than that of a conventional hardware delta prefetcher. Figure 11(c) shows the speedup of 450.soplex using delta COMPASS shaders with varying prefetch depth. Like a stride COMPASS shader, as the prefetch depth increases, the MT delta COMPASS shader can catch up with the ZI delta COMPASS. (Figure 11(d)). Meanwhile, because a delta COMPASS shader executes a rather large shader code, its throughput is low. Consequently, a number of miss requests are discarded due to overflow (Figure 11(e)). Notwithstanding the discarded miss history, a delta prefetcher is found to train itself pretty rapidly resulting in high performance (Figure 11(c)).

5.5 Evaluation of Region COMPASS

Unlike previous COMPASS designs, a simplified version of a region prefetcher requires a small number of registers with a lightweight COMPASS shader code. To fetch 64 cache lines mapped to the same page, our region COMPASS uses 64 threads of a wavefront, each of which calculates a corresponding cache line using its own thread ID. This computation is extremely simple and requires only four registers per thread. Similar to a Markov COMPASS, we use the miss address to index a thread. This decision is to efficiently prefetch cache lines even if a single PC constantly generates cache misses very frequently. Because there is not much trade-off in this design, we do not show further in-depth sensitivity study for it. Its overall benefit will be discussed in Section 5.7.

5.6 Evaluation of Custom COMPASS

We also evaluated a custom COMPASS shader for 429.mcf. This shader is PC-indexed and uses 16 registers per thread. We varied the prefetch depth for this multi-threaded COMPASS shader. The number of required threads for each shader is shown in Figure 12(a). As shown in Figure 12(d) and Figure 12(e), the latency of this shader is around 130 cycles and the number of discarded requests accounts for around 2% of L2 cache misses. The performance of 429.mcf is improved by 64% when the prefetching depth of this custom shader is four.

5.7 COMPASS vs. GHB Stride Prefetchers

Table 4 summarizes our simulation results. Instead of showing all results with different prefetching depth or width, we selected the best performing COMPASS shader in each type of COMPASS. Here, we also show the performance of a conventional hardware-based stride GHB prefetcher that has a 256-entry global history buffer, the same size used in [30]. We use three stride GHB prefetchers with prefetch depth of 4, 8, and 16. Note that a stride GHB prefetcher was reported the most efficient prefetcher among many other contemporaries in [31].

As shown in the table, the stride GHB prefetchers work pretty well with 462.libquantum and 410.bwaves, however, they may also degrade performance, e.g., 429.mcf. This is where the strength comes from by employing our programmable COMPASS. As mentioned earlier, COMPASS allows an application vendor or the OS to select the best performing prefetcher by customizing the GPU prefetch shader code according to the peculiar behavior of an application. Even with the small benchmark program sample shown in Table 4, there exists no one-size-fits-all best prefetcher for them. Flexibility of a programmable COMPASS facilitates the choice of the best prefetching strategy. As shown in the table, for the majority of the benchmark applications, flexibility allows COMPASS to perform better than a GHB prefetcher. Three exceptions are 462.libquantum, 410.bwaves, and 437.leslie3d, all of which prefer a stride prefetching technique. As explained in Section 5.2, for 462.libquantum, a stride COMPASS fails to catch up with a stride GHB prefetcher with prefetching depth 16 due to its low throughput. Unfortunately, even region COMPASS fails to catch up

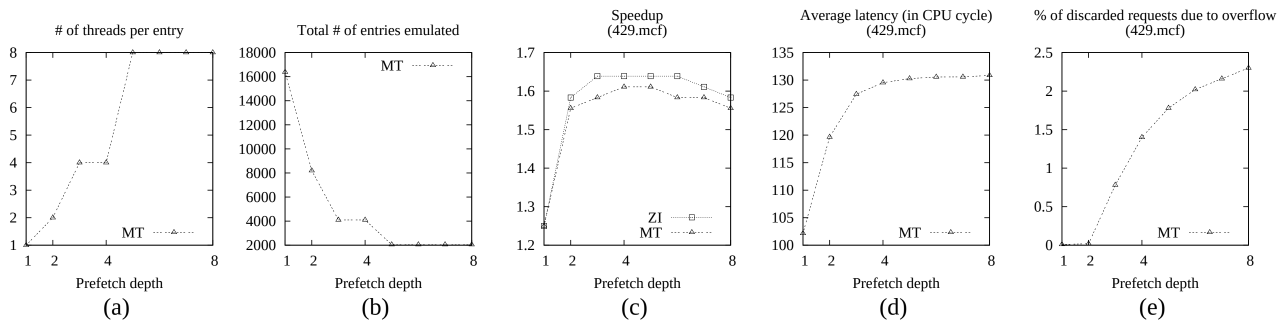

Figure 12: Evaluation of Custom COMPASS for 429.mcf (MT: multi-threaded, ZI: zero-latency, infinite-throughput)

Table 4: Overall Results (d: prefetch depth, w: prefetch width)

Speedup		SPECint06					SPECfp06						Geomean
		429. mcf	462. libquantum	471. omnetpp	473. astar	483. xalancbmk	410. bwaves	433. milc	437. leslie3d	450. soplex	454. calculix	459. GemsFDTD	
HW GHB	Stride (d=4)	0.89	1.44	1.15	1.00	**0.96**	1.80	1.11	1.27	1.06	1.11	1.05	1.15
	Stride (d=8)	0.89	2.85	**1.17**	1.04	0.93	**1.89**	**1.11**	**1.28**	1.11	1.11	1.10	1.24
	Stride (d=16)	**0.89**	**4.79**	1.15	**1.05**	0.93	1.80	1.09	1.27	**1.14**	**1.11**	**1.10**	**1.29**
COMPASS	Stride (d=16)	1.06	2.69	1.15	1.07	0.93	**1.84**	1.15	**1.27**	1.34	**1.12**	1.11	1.28
	Markov (w=3)	0.97	0.74	**1.22**	0.82	**1.18**	0.73	0.87	0.69	0.86	0.91	0.60	0.85
	Delta (d=16)	1.33	1.13	1.15	1.05	0.93	1.62	**4.22**	1.26	1.31	1.10	**1.70**	1.38
	Region	**2.00**	**3.49**	0.25	**1.09**	0.82	1.42	1.59	1.23	**1.45**	1.11	0.29	1.06
	Custom (429.mcf) (d=4)	1.61	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Perfect	Perfect L2	2.56	4.82	1.72	1.52	4.07	1.93	4.61	1.30	2.08	1.12	2.08	2.25
COMPASS Geomean (when selected properly)													**1.68**

with the GHB prefetcher. For 410.bwaves and 437.leslie3d, the performance difference is negligible. On the other hand, our custom COMPASS shader based on exponential increased stride for 429.mcf outperforms all other table-based prefetchers, although a brute-force, region COMPASS is found to perform even better than the custom COMPASS shader.

On average (geometric mean), stride GHB prefetchers with the prefetching depth of 4, 8, and 16 improve performance by 15%, 24%, and 29%, respectively. On the contrary, if an application vendor or an OS choose the application-specific COMPASS shader, we can improve performance by 68% on average (geometric mean).

5.8 Multi-Core Effects

The majority of data prefetching techniques have been focused on improving the performance of single programs. It is not well understood whether we need to partition a prefetch table into several sub-tables or we should allow different processes to corrupt the miss history of each other. Clearly, this study is a separate piece of work. Thus, in this section, we only briefly discuss how the OS can use COMPASS in a platform with two CPU cores and a GPU.

The best way to handle this problem is to co-schedule a computation-intensive application and a memory-intensive application [40]. This is not only good for reducing pollution in a prefetch table but also good for reducing contention in the L2. Such shared cache-aware scheduling methods has been investigated recently [12, 13, 32, 21]. Another way to address this problem is prioritizing one application. In this case, the highest priority application will run COMPASS alone while other concurrent applications will not benefit.

On the other hand, we can schedule two memory-intensive applications that benefit from the same type of COMPASS. For example, we can schedule 433.milc and 459.GemsFDTD, the performance of which is significantly improved by delta COMPASS. Even though these two applications share COMPASS, their performance can still be improved because the capacity of a prefetch table that delta COMPASS emulates is a lot larger than that of a conventional delta prefetcher. Table 5 shows results using this policy. For these simulations, we fast-forwarded the first 20 billion instructions and measured the performance of the next two billion instructions. Here again, we also show simulation results with the hardware-based stride GHB prefetchers for comparison. The first set of applications, 410.bwaves and 462.libquantum, in the table represents two applications that benefit a lot from a stride COMPASS shader. As shown in the table, a stride COMPASS can improve performance significantly. As explained previously, because a PC of 462.libquantum constantly generates cache misses very frequently, the speedup of COMPASS is not as high as that of the GHB prefetchers. However, when we simulate another two applications, 410.bwaves and 450.soplex, the performance of a stride COMPASS is as good as that of GHB prefetchers. In contrast, when we schedule two applications, 433.milc and 459.GemsFDTD, that benefit from a delta COMPASS, our COMPASS clearly outperforms GHB prefetchers by simply programming the GPU.

On the other hand, when 471.omnetpp and 483.xalancbmk, which benefit from the Markov COMPASS, are scheduled together, the Markov COMPASS outperforms the GHB prefetchers. However, the speedup of these two applications are actually lower than the speedup of each application measured in a single-core simulation. This outcome is not surprising because a Markov prefetcher usually requires a large table and because these applications share the same capacity as the previous single-core simulations. Lastly, when 410.bwaves and 433.milc, which benefit most from the stride COMPASS and delta COMPASS, respectively, are scheduled

Table 5: Dual-Core Simulation Results

		410.bwaves 462.libquantum	410.bwaves 450.soplex	433.milc 459.GemsFDTD	471.omnetpp 483.xalancbmk	410.bwaves 433.milc	Geo-mean
HW GHB	stride(d=4)	2.29	1.96	1.43	1.04	1.89	1.66
	stride(d=8)	2.71	2.08	1.49	1.05	2.00	1.78
	stride(d=16)	3.12	2.03	1.43	1.04	1.95	1.79
COMPASS		2.52 (stride) (d=16)	2.09 (stride) (d=16)	3.05 (delta) (d=16)	1.10 (markov) (w=3)	2.67 (delta) (d=16)	2.16
Perfect L2		3.81	2.83	4.26	2.54	3.67	3.36

together, the delta COMPASS can still improve performance significantly. This is because a typical delta prefetcher can do the same job as a stride prefetcher by simply consuming larger table space.

In summary, COMPASS has potentials to improve the performance of multiple cores if the OS schedules applications wisely. As mentioned, an OS-level scheduling policy is out of scope of this paper, and it remains as our future work. Note that as the number of CPU cores scales in the future, the number of SIMD arrays will scale as well, thus a GPU will be able to provide enough throughput for prefetching and to emulate a larger prefetch table.

5.9 Hardware, Software, and Power Overhead

The hardware overhead of COMPASS includes the following. (1) The MAP requires three registers for pipelining. (2) It needs a command buffer, which stores the GPU execution command. Note that the command buffer stores the command itself, not the code for COMPASS shader. Thus, this buffer is extremely small. (3) The command processor of a GPU needs to be modified to understand the new GPU execution command. This overhead is basically a microcode patch because the command processor is a programmable processor. (4) The setup engine should be able to index a thread based on a value retrieved from the command. The overhead of this index function is very minimal because it is basically a simple masking operation. (5) Each TEX unit should support a prefetching operation. This is also a very negligible change because we just need to disable the write-back path of a conventional GPU load operation. (6) The address translation process through a TLB should be disabled because MAP is forwarding a physical address from the L2 cache and we can simply use it without address translation.

On the other hand, to provide a COMPASS shader to MAP, an OS needs to have one or multiple COMPASS shaders built-in so that it can use one of these shaders along with CPU processes. Optionally, the OS can provide an API function that allows an individual application vendor to provide its custom-designed COMPASS shaders. Second, the task scheduler of the OS should be able to enable the MAP.

Because we are largely reusing existing GPU hardware for COMPASS and the TDP (thermal design power) of our integrated chip already considers fully active GPU and CPU cores, the use of COMPASS should not aggravate the thermal and power dissipation. The additional power consumed by our lightweight supporting hardware described above will be negligible. Furthermore, in most cases, our COMPASS does not fully utilize the GPU, e.g., using 16 threads out of 64 threads of a wavefront. Lastly, our COMPASS shader is executed only upon an L2 cache miss, which occurs not frequently in common cases.

6. RELATED WORK

Hardware prefetching is widely investigated by prior studies [6, 19, 30, 20, 30, 23, 9]. These hardware prefetchers use a dedicated hardware to predict and bring in the cache lines in advance whereas COMPASS is designed to reuse the existing GPU hardware and leverage their programmability and flexibility so that one (e.g., software vendors) can write or select the most appropriate COMPASS shader to improve the performance of their applications. On the other hand, software prefetching was also investigated by prior studies [5, 7] and various data prefetch instructions were added in almost every commercial microprocessor. These techniques basically insert software instructions into a code so that a future memory fetch can be initiated in advance. COMPASS is different from a conventional software prefetching technique because of the following. First, COMPASS does not consume any compute bandwidth of the main processor, rather, it leverages the idle GPU to perform the task. Second, it emulates a hardware prefetcher and its associated hardware (e.g., prefetch table). As we demonstrated, the emulated table oftentimes is much larger than the practical size of a real hardware table. Third, COMPASS does not require recompilation of a code.

Helper thread techniques [11, 8, 25, 4, 22, 29] were proposed to prefetch cache lines by precomputing load addresses. They often relied on another thread running on the same processor (e.g., SMT or Multicore) to achieve their goal. As such, they could diminish the return if they compete the same resources needed by their master thread. Similar to software prefetching, these techniques usually run a stripped-down slice of the original program to precompute miss addresses and bring in those data in time. In contrast, COMPASS emulates the behavior of a hardware prefetcher, and it can be enabled at runtime through the OS. The closest work is an event-driven helper thread [15] that emulates a hardware prefetcher on an idle CPU. COMPASS is the first approach to use an idle on-chip GPU and implement shader codes to enable a programmable prefetching scheme. Furthermore, emulating prefetchers on a GPU is far more challenging because a conventional GPU is optimized for throughput sacrificing the latency of a thread and because a GPU runs at lower clock frequency than a CPU. This paper investigates such challenges regarding the latency and the throughput of emulating a hardware prefetcher on a GPU.

7. CONCLUSION

In this paper, we proposed COMPASS, a compute shader-assisted prefetching scheme, to improve the memory performance of an integrated chip while the on-chip GPU is idle. With very lightweight architectural support, COMPASS shaders can emulate various hardware prefetchers for improving performance of single-thread applications. Moreover, due to its programmability and flexibility, one can implement or select the best performing prefetching algorithm specially tailored for a specific application to exploit its particular memory access behavior. We designed, evaluated, and analyzed different COMPASS designs and also performed a case study to demonstrate a custom-designed COMPASS. Our simulation results showed that COMPASS can improve the single-thread performance of memory-intensive applications by 68% on average. With COMPASS, we envision that application vendors can supply an application-specific COMPASS shader bundled with their software in the future, and have it loaded at runtime to improve memory performance.

8. REFERENCES

[1] Advanced Micro Devices Inc., R700-Family Instruction Set Architecture, http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf.

[2] Intel Corporation, Intel Shifts Future Core^{TM} Processor into Turbo Mode, http://www.intel.com/pressroom/archive/releases/20080819comp.htm.

[3] Intel Corporation, Optimizing Application Performance on Intel® Core^{TM} Microarchitecture Using Hardware-Implemented Prefetchers, http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers.

[4] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[6] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[7] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the International Symposium on Microarchitecture*, 1991.

[8] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: long-range prefetching of delinquentloads. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[9] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[10] M. Dimitrov and H. Zhou. Combining Local and Global History for High Performance Data Prefetching. In *The Journal of Instruction-Level Parallelism Data Prefetching Championship*, 2009.

[11] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*, 1997.

[12] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[13] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the International Symposium on Microarchitecture*, 2007.

[15] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[16] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[17] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[18] R. Huddy. ATI Radeond^{TM} HD 2000 SeriesTechnology Overview. In *AMD Technical Day, The Develop Conference & Expo*, 2007.

[19] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the International Symposium on Computer Architecture*, 1997.

[20] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[22] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[23] W. Lin, S. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[24] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: general purpose computation on graphics hardware. In *Proceedings of the conference on SIGGRAPH 2004 course notes*, 2004.

[25] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[26] M. Mantor. Radeon R600, a 2nd Generation Unified Shader Architecture. In *Proceedings of the 19th Hot Chips Conference, August*, 2007.

[27] M. Mantor. Entering the Golden Age of Heterogeneous Computing. In *Performance Enhancement on Emerging Parallel Processing Platforms*, 2008.

[28] C. Moore. The Role of Accelerated Computing in the Multi-core Era. In *Workshop on Manycore and Multicore Computing: Architectures, Applications And Directions*, 2007.

[29] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2003.

[30] K. Nesbit and J. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2004.

[31] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the International Symposium on Microarchitecture*, 2004.

[32] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[33] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[34] N. Rubin. Issues And Challenges In Compiling for Graphics Processors (Keynote speech). In *Proceedings of the International Symposium on Code Generation and Optimization*, 2008.

[35] A. Sharif and H.-H. S. Lee. Data Prefetching Mechanism by Exploiting Global and Local Access Patterns. In *The Journal of Instruction-Level Parallelism Data Prefetching Championship*, 2009.

[36] S. L. Smith. Intel Roadmap Overview. In *Intel Developer Forum*, 2008.

[37] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[38] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[39] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. IBM Technical White Paper, October 2001.

[40] N. Tuck and D. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003.