In the proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2010

Many-Thread Aware Prefetching Mechanisms for GPGPU Applications

Jaekyu Lee Nagesh B. Lakshminarayana Hyesoon Kim Richard Vuduc

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{*jaekyu.lee, nageshbl, hyesoon, richie*}*@cc.gatech.edu*

***Abstract*— We consider the problem of how to improve memory latency tolerance in massively multithreaded GPGPUs when the thread-level parallelism of an application is not sufficient to hide memory latency. One solution used in conventional CPU systems is prefetching, both in hardware and software. However, we show that straightforwardly applying such mechanisms to GPGPU systems does not deliver the expected performance benefits and can in fact hurt performance when not used judiciously.**

This paper proposes new hardware and software prefetching mechanisms tailored to GPGPU systems, which we refer to as *many-thread aware prefetching* (MT-prefetching) mechanisms. Our software MT-prefetching mechanism, called *inter-thread prefetching*, exploits the existence of common memory access behavior among fine-grained threads. For hardware MT-prefetching, we describe a scalable prefetcher training algorithm along with a hardware-based inter-thread prefetching mechanism.

In some cases, blindly applying prefetching degrades performance. To reduce such negative effects, we propose an *adaptive prefetch throttling* scheme, which permits automatic GPGPU application- and hardware-specific adjustment. We show that adaptation reduces the negative effects of prefetching and can even improve performance. Overall, compared to the state-of-the-art software and hardware prefetching, our MT-prefetching improves performance on average by 16% (software pref.) / 15% (hardware pref.) on our benchmarks.

***Keywords*-prefetching; GPGPU; prefetch throttling;**

I. Introduction

Broadly speaking, multithreading, caches, and prefetching constitute the three basic techniques for hiding memory latency on conventional CPU systems. Of these, modern general-purpose GPU (GPGPU) systems employ just two: massive fine-grained multithreading and caches (including explicitly-programmed scratchpad/local-store memory). Multithreading, in particular, is the primary technique, with current GPGPU processors supporting ten to a hundred times more threads than state-of-the-art CPU systems. Consequently, when there is significant memory-level parallelism, but insufficient amounts of thread-level parallelism and data reuse, we expect poor performance on GPGPUs.

Thus, it is natural to consider prefetching for GPGPUs. For example, NVIDIA's Fermi architecture has non-binding software prefetching instructions [23], and binding software prefetching via ordinary load operations has been shown to be effective [28].[1] However, adding prefetching to GPGPUs in a straightforward way does not necessarily improve performance and may even hurt performance when not used judiciously [34].

Helpful vs. harmful prefetches, and feedback-driven mechanisms. Prefetching can be harmful in certain cases; one of the reasons for this is explained here. Prefetch requests can increase the total number of memory requests, and thereby may increase the delay in servicing demand memory requests. This problem is more severe in GPGPUs, since the number of in-flight threads and memory requests are much higher in typical GPGPU applications compared to their CPU counterparts: if each of 100 threads generates an additional prefetch request, there will suddenly be 100 outstanding memory requests. On the other hand, prefetching can be helpful if the memory system can accommodate additional memory requests without increasing demand request service time. Hence, we need a mechanism to decide when prefetching is useful and when it is harmful.

There are several recently proposed feedback-driven hardware prefetching mechanisms [8, 9, 16, 31]. However, none directly apply to GPGPUs. First, these mechanisms mainly control the aggressiveness of speculative prefetch requests, such as prefetch degree and distance, based on accuracy and timeliness of prefetching. In GPGPU applications, the accuracy of prefetching can easily be 100%, since memory access patterns in current GPGPU applications are usually regular. Moreover, late prefetch requests in GPGPUs are not as harmful as they are in CPUs since memory latency can be partially hidden by switching to other threads. Hence, we need new feedback mechanisms that specifically target GPGPU architectures. Second, prior feedback mechanisms apply mostly to hardware, with relatively little work on feedback mechanisms for software prefetching. In practice, software-based prefetching mechanisms require feedback at run-time to throttle excessive prefetch requests. These observations motivate our proposal for a new class of software *and* hardware prefetching mechanisms for GPGPUs. We refer to this new class as *many-thread aware (MT)* prefetching.

[1] Binding prefetching changes architectural state but non-binding prefetching does not [17, 20].

Our contributions. We claim the following contributions.

1) We propose new GPGPU-specific software and hardware prefetching mechanisms, which we refer to as *inter-thread prefetching*. In inter-thread prefetching, a thread prefetches data for other threads rather than for itself.
2) We enhance the hardware prefetcher training algorithms so that they scale to a large number of threads.
3) To enhance the robustness of these mechanisms, we develop a new prefetch throttling mechanism that dynamically adjusts the level of prefetching to avoid performance degradation.

Although we focus on GPGPU applications in this paper, our contributions can be applied to other many-thread architectures and SIMT (single-instruction, multiple-thread) applications.

II. Background

A. Execution Model

The GPGPU system we model follows NVIDIA's CUDA programming model [25]. In the CUDA model, each core is assigned a certain number of *thread blocks*, a group of threads that should be executed concurrently. Each thread block consists of several *warps*, which are much smaller groups of threads. A warp is the smallest unit of hardware execution. A core executes instructions from a warp in an *SIMT* (Single-Instruction Multiple-Thread) fashion. In SIMT execution, a single instruction is fetched for each warp, and all the threads in the warp execute the same instruction in lockstep, except when there is control divergence. Threads and blocks are part of the CUDA programming model, but a warp is an aspect of the microarchitectural design.

B. GPGPU Architecture and the Memory System

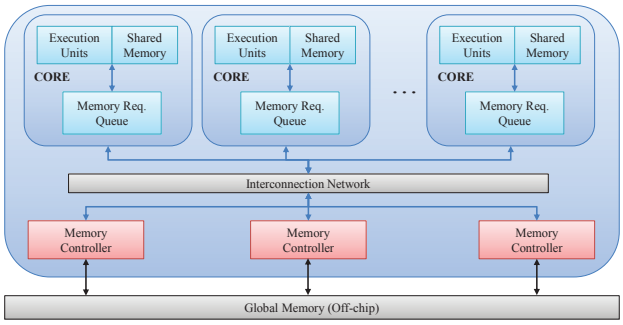
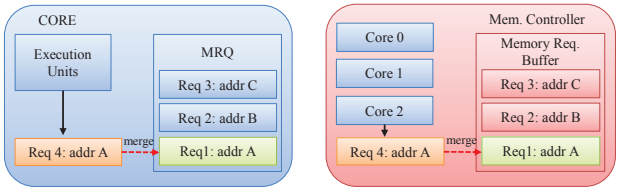


Figure 1. An overview of the baseline GPGPU architecture.

1) System Architecture: Figure 1 shows the block diagram of our baseline GPGPU architecture, which is similar to NVIDIA's 8800GT [24]. The basic design consists of several cores and an off-chip DRAM with memory controllers located inside the chip. Each core has SIMD execution units, a software-managed cache (shared memory), a memory-request queue (MRQ), and other units. The processor has an in-order scheduler; it executes instructions from one warp, switching to another warp if source operands are not ready. A warp may continue to execute new instructions in the presence of multiple prior outstanding memory requests, provided that these instructions do not depend on the prior requests.

2) Request Merging in the Memory System: The memory system may merge different memory requests at various levels in the hardware, as shown in Figure 2. First, each core maintains its own MRQ. New requests that overlap with existing MRQ requests will be merged with the existing request. This type of merging is ***intra**-core merging*. Secondly, requests from different cores are buffered in the memory-request buffer of the DRAM controller as they are being served. If a core generates a request that overlaps with a request already in the memory-request buffer, the new request is merged with the old request. This type of merging is ***inter**-core merging*.



(a) Intra-core merging (b) Inter-core merging

Figure 2. Memory-request merging.

C. Prefetching

1) Software Prefetching: There are currently two software prefetching mechanisms for GPGPUs.

Prefetching into registers. Ryoo et al. [28] describe a register prefetching mechanism for GPGPU applications. Their scheme is the same as binding prefetching as used in CPUs. This mechanism does not require any special prefetching instructions and also applies to GPGPUs that do not have a hardware managed cache. However, since it increases register usage,[2] it might decrease the number of threads that may be active, thereby reducing thread-level parallelism [34].

Prefetching into cache. NVIDIA's recently introduced Fermi architecture supports software prefetching via two explicit prefetch instructions. These instructions prefetch a cache block from either global or local memory into either the L1 or L2 cache.

2) Hardware Prefetching: Since there is no publicly available information about hardware prefetchers in GPGPUs, we assume that current GPGPUs do not include any hardware prefetchers. In this section, we briefly review some of the hardware prefetchers that have been proposed instead for CPUs.

Stream prefetchers. A stream prefetcher monitors a memory region (e.g., a few cache blocks) and detects the direction

[2]Even if data is prefetched from the global memory and stored in a shared memory, registers must be used before storing the data in the shared memory.

of access [15, 26]. Once a constant access direction is detected, the stream prefetcher launches prefetch requests in the direction of access.

Stride prefetchers. A stride prefetcher tracks the address delta between two accesses by the same PC or between two accesses that are within the same memory region [4, 11]. If a constant delta is detected, the prefetcher launches prefetch requests using this delta.

GHB prefetchers. A global history buffer prefetcher stores recent miss addresses in an n-entry FIFO table, called the global history buffer (GHB) [14, 21]. Each entry stores a miss address and a link pointer to another entry in the table. Using this pointer, a GHB prefetcher can detect stream, stride, and even irregular repeating memory address patterns.

3) Prefetching Configuration: We characterize the "aggressiveness" of a prefetcher (whether in software or hardware) by two parameters: the *prefetch distance* and the *prefetch degree*. The *prefetch distance* specifies how advance prefetch requests can be generated from the current demand request that triggered prefetching. The *prefetch degree* determines how many requests can be initiated by one prefetch trigger.

III. Prefetching Mechanisms for GPGPUs

This section describes our *many-thread aware* prefetching (MT-prefetching) schemes, which includes both hardware and software mechanisms. To support these schemes, we augment each core of the GPGPUs with a prefetch cache and a prefetch engine. The prefetch cache holds the prefetched blocks from memory and the prefetch engine is responsible for throttling prefetch requests (see Section V).

A. Software Prefetching

We refer to our software prefetching mechanism as *many-thread aware software prefetching* (MT-SWP). MT-SWP consists of two components: conventional *stride prefetching* and a newly proposed *inter-thread prefetching* (IP).

1) Stride Prefetching: This mechanism is the same as the traditional stride prefetching mechanism. The prefetch cache stores any prefetched blocks.

2) Inter-thread Prefetching (IP): One of the main differences between GPGPU applications and traditional applications is that GPGPU applications have a significantly higher number of threads. As a result, the execution length of each thread is often very short. Figure 3 shows a snippet of sequential code with prefetch instructions and the equivalent CUDA code without prefetch instructions. In the CUDA code, since the loop iterations are parallelized and each thread executes only one (or very few) iteration(s) of the sequential loop, there are no (or very few) opportunities to insert prefetch requests for subsequent iterations (if any) of the loop executed by the thread. Even if there are opportunities to insert prefetch requests, the coverage of such requests will be very low for this type of benchmark.

```
for (ii = 0; ii < 100; ++ii) {
  prefetch(A[ii+1]);
  prefetch(B[ii+1]);
  C[ii] = A[ii] + B[ii];
}
```

```
// there are 100 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

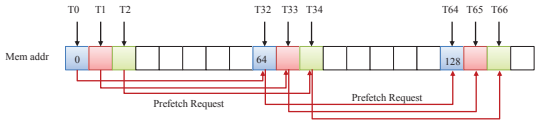
Figure 3. An example sequential loop (left) and the corresponding CUDA code (right).

Alternatively, each thread could prefetch for threads in *other* warps, instead of prefetching for itself. Figure 4 shows an example in which a given thread prefetches for the corresponding thread in the next thread warp. Threads T0, T1, and T2 in Warp 0 generate prefetch requests for T32, T33, and T34 in Warp 1. The thread IDs can be used to prefetch data for other thread warps. This mechanism is called inter-thread prefetching (IP).

```
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  // a warp consists of 32 threads
  int next_warp_id = tid + 32;

  prefetch(A[next_warp_id]);
  prefetch(B[next_warp_id]);
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

(a) Prefetch code example



(b) An example of inter-thread prefetching with memory addresses shown

Figure 4. Inter-thread prefetching example.

IP may not be useful in two cases. The first case is when demand requests corresponding to prefetch requests have already been generated. This can happen because warps are not executed in strict sequential order. For example, when T32 generates a prefetch request for T64, T64 might have already issued the demand request corresponding to the prefetch request generated by T32. These prefetch requests are usually merged in the memory system since the corresponding demand requests are likely to still be in the memory system.

The second case is when the warp that is prefetching is the last warp of a thread block and the target warp (i.e. thread block to which the target warp belongs) has been assigned to a different core. Unless inter-core merging occurs in the DRAM controller, these prefetch requests are useless. This problem is similar to the out-of-array-bounds problem encountered when prefetching in CPU systems. Nevertheless, we find that the benefits of IP far outweigh its negative effects.

B. Hardware Prefetching

We refer to our hardware prefetching mechanism as the *many-thread aware hardware prefetcher* (MT-HWP). MT-HWP has (1) enhanced prefetcher training algorithms that provide improved scalability over previously proposed stream/stride prefetchers and (2) a hardware-based inter-thread prefetching (IP) mechanism.

1) Scalable Hardware Prefetcher Training: Current GPGPU applications exhibit largely regular memory access patterns, so one might expect traditional stream or stride perfetchers to work well. However, because the number of threads is often in the hundreds, traditional training mechanisms do not scale.

Here, we describe extensions to the traditional training policies, for program counter (PC) based stride prefetchers [4, 11], that can overcome this limitation. This basic idea can be extended to other types of prefetchers as well.

PC	Warp ID	Addr	Delta
0x1a	1	0	-
0x1a	1	1000	1000
0x1a	1	2000	1000
0x1a	2	10	-
0x1a	2	1010	1000
0x1a	2	2010	1000
0x1a	3	20	-
0x1a	3	1020	1000
0x1a	3	2020	1000

Time

PC	Warp ID	Addr	Delta
0x1a	1	0	-
0x1a	2	10	10
0x1a	1	1000	990
0x1a	3	20	-980
0x1a	2	1010	990
0x1a	3	1020	10
0x1a	3	2020	1000
0x1a	1	2000	-20
0x1a	2	2010	10

Figure 5. An example of memory address with/without warp interleaving (left: accesses by warps, right: accesses seen by a hardware prefetcher).

- *Per warp training:* Stream and stride detectors must be trained on a per-warp basis, similar to those in simultaneous multithreading architectures. This aspect is critical since many requests from different warps can easily confuse pattern detectors. Figure 5 shows an example. Here, a strong stride behavior within each warp exists, but due to warp interleaving, a hardware prefetcher only sees a random pattern. Even though stream prefetchers are trained by memory address regions, finding the pattern of the direction of accesses might be difficult. In our MT-HWP, stride information trained per warp is stored in a *per warp stride (PWS)* table, which is similar to the prefetch table in traditional stride/stream prefetchers.
- *Stride promotion:* Since memory access patterns are fairly regular in GPGPU applications, we observe that when a few warps have the same access stride for a given PC, all warps will often have the same stride for the PC. Based on this observation, when at least three PWS entries for the same PC have the same stride, we promote the PC stride combination to the *global stride (GS)* table. By promoting strides, yet-to-be-trained warps can use the entry in the GS table to issue prefetch requests immediately without accessing the PWS table. Promotion also helps to minimize the space requirement for the PWS table. If stride distances among warps are not the same, the stride information cannot be promoted and stays in the PWS table. In our design, both PWS and GS tables use a LRU replacement policy.

2) Inter-thread Prefetching in Hardware: We propose a hardware-based inter-thread prefetching (IP) mechanism, in addition to our software-based IP scheme (Section III-A). The key idea behind hardware IP is that when an application exhibits a strided memory access pattern across threads at the same PC, one thread generates prefetch requests for another thread. This information is stored in a separate table called an IP table. We train the IP table until three accesses from the same PC and different warps have the same stride; thereafter, the prefetcher issues prefetch requests from the table entry.

3) Implementation: Figure 6 shows the overall design of the MT-HWP, which consists of the three tables discussed earlier: PWS, GS, and IP tables. The IP and GS tables are indexed in parallel with a PC address. When there are hits in both tables, we give a higher priority to the GS table because strides within a warp are much more common than strides across warps. Furthermore, the GS table contains only promoted strides, which means an entry in the GS table has been trained for a longer period than strides in the IP table. If there are no hits in any table, then the PWS table is indexed in the next cycle. However, if any of the tables have a hit, the prefetcher generates a request.

IV. Understanding Useful vs. Harmful Prefetching in GPGPU

In practice, overly aggressive prefetching can have a negative effect on performance. In this section, we explain when prefetching is useful, neutral (has no effect), and harmful, using a simple analytical model.

A. Useful or Neutral (No-effect) Prefetch Requests

The principal memory latency tolerance mechanism in a GPGPU is multithreading. Thus, if a sufficient number of warps and/or enough computation exist, memory latency can be easily hidden. To aid our understanding of the utility of prefetching, we define a new term, the *minimum tolerable average memory latency* (*MTAML*). *MTAML* is the minimum average number of cycles per memory request that does not lead to stalls. This value is essentially proportional to the amount of computation between memory requests and the number of active warps:

$$MTAML = \frac{\#comp\_inst}{\#mem\_inst} \times (\#warps - 1) \quad (1)$$

$\#comp\_inst$: number of non-memory warp-instructions,
$\#mem\_inst$: number of memory warp-instructions,
$\#warps$: number of active warps (warps that are concurrently running on a core)

That is, as the amount of computation per memory instruction increases, *MTAML* also increases, meaning it is easier to hide memory latency. Similarly, if there are many warps,

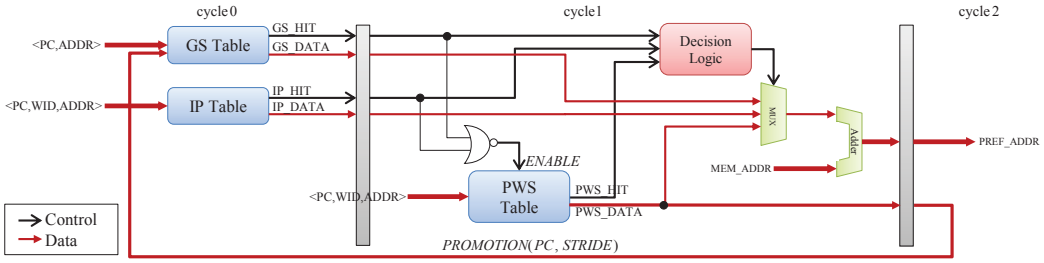


Figure 6. Many-thread aware hardware prefetcher (MT-HWP).

we can easily hide memory latency, and *MTAML* again increases. Furthermore, if the average memory latency is below *MTAML*, we do not expect any benefit from prefetching.

When there is prefetching, a prefetch cache hit will have the same latency as other computational instructions. Given the probability of a prefetch cache hit, we can compute *MTAML under prefetching* as follows:

$$MTAML\_pref = \frac{\#comp\_new}{\#memory\_new} \times (\#warps - 1) \quad (2)$$

$$\#comp\_new = \#comp\_inst + Prob(pref\_hit) \times (\#mem\_inst) \quad (3)$$

$$\#memory\_new = (1 - Prob(pref\_hit)) \times (\#mem\_inst) \quad (4)$$

$Prob(pref\_hit)$: probability of the prefetch cache hit

That is, increasing the prefetch hit probability reduces the number of instructions that have to be tolerated (denominator), thereby increasing the effective *MTAML*. Note that $\#memory\_new$ does not include prefetch instructions, because *MTAML* is calculated for demand memory instructions only.

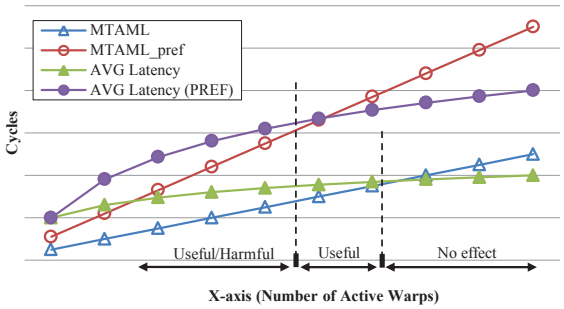


Figure 7. Minimum tolerable average memory latency vs. # of active warps.

We now explain how *MTAML* can help distinguish when prefetching has an overall beneficial (useful), harmful, or no-effect.

Consider a hypothetical computation for which the measured average latency of a memory operation for a varying number of warps is as shown by the curve *AVG Latency* in Figure 7. As we increase the number of in-flight memory instructions, the average memory latency also increases due to an increase in delay and contention in the memory system. Now, suppose we add prefetching; then, the measured average memory latency ignoring successfully prefetched memory operations will follow the curve *AVG Latency (PREF)*. In this hypothetical example, the latency has actually increased with prefetching for the reasons outlined in Section IV-B. The question is whether this increase has any impact on the actual execution time.

MTAML helps answer this question. We differentiate the following 3 cases.

1) *AVG Latency* $< MTAML$ and *AVG Latency (PREF)* $< MTAML\_pref$: multithreading is effective and prefetching has no effect. Even without prefetching, the application can tolerate memory latency therefore prefetching does not provide any additional benefit.
2) *AVG Latency* $> MTAML$: there is at least some potential for prefetching to be beneficial, *provided AVG Latency (PREF)* is less than $MTAML\_pref$. In this case, without prefetching, the application cannot tolerate memory latency (i.e., $MTAML$ is less than *AVG Latency*), but with prefetching $MTAML\_pref$ becomes greater than *AVG Latency (PREF)*, so the application can tolerate memory latency.
3) Remaining cases: prefetching might be useful or harmful. The application cannot completely tolerate memory latency with and without prefetching. However, this does not mean that all prefetch instructions are harmful because we are considering only the average case in this example.

However, distinguishing these cases is a somewhat subtle and perhaps non-obvious process because 1) the number of warps varies depending on input sizes and 2) the same static prefetch instruction can be harmful/useful/no-effect depending on the number of warps. Thus, this provides the motivation for our *adaptive* scheme in Section V.

B. Harmful Prefetch Requests

The key reason prefetch requests may hurt performance is due to the potential increase in delay in servicing memory requests. The reasons for such delays are numerous, including (a) queuing delay, (b) DRAM row-buffer conflicts, (c) wasting of off-chip bandwidth by early prefetches (prefetched block is evicted before it is used), and (d) wasting of off-chip bandwidth by inaccurate prefetches. The first three reasons

may still harm performance even when prefetch accuracy is near 100%.

We observe these cases in our experiments. Figure 8 shows the average memory latency with prefetching, normalized to the no-prefetching case (The detailed methodology and memory policies are discussed in Section VI). A circle on top of each bar displays the prefetch accuracy. The average memory latency increases significantly with software prefetching, just as in our hypothetical example. In some cases, it is more than three times the average latency without prefetching. Moreover, this average latency can still be more than two times the average without prefetching even when the accuracy of prefetching is close to 100%, as in the case of the *stream* benchmark. Thus, we need metrics other than accuracy to detect harmful prefetch requests.

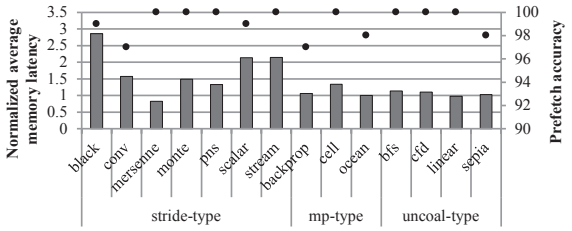


Figure 8. Normalized memory latency (bar) and prefetch accuracy (circle).

V. Adaptive Prefetch Throttling

We would like to eliminate the instances of prefetching that yield negative effects while retaining the beneficial cases. Thus, we propose a *prefetch throttling mechanism* for MT-prefetching. Figure 9 shows the overall design of our throttling framework.

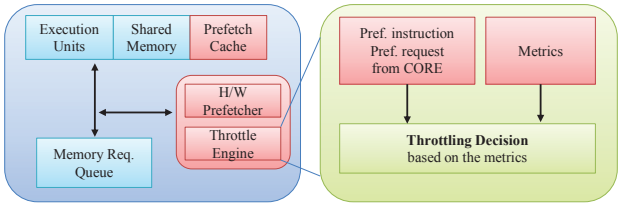


Figure 9. Adaptive prefetch throttling framework.

A. Metrics

Our ultimate throttling mechanism is based on the consideration of two metrics, *early eviction rate* and *merge ratio*.

The **early eviction rate** is the number of items (cache blocks) evicted from the prefetch cache before their first use, divided by the number of useful prefetches, as shown in Eq. 5.

$$Metric(EarlyEviction) = \frac{\#EarlyEvictions}{\#UsefulPrefetches} \quad (5)$$

The number of early evictions from the prefetch cache and useful prefetches are measured in the same way as in feedback-driven prefetch mechanisms [8, 9, 31]. We choose this as a primary metric because the number of early prefetches directly tells us about the degree of harmful prefetches. These early evicted prefetches are always harmful. They consume system bandwidth, delay other requests, and evict useful blocks from the prefetch cache without themselves being used.

The **merge ratio** is the number of *intra-core* merges that occur divided by the total number of requests:

$$Metric(Merge) = \frac{\#IntraCoreMerging}{\#TotalRequest} \quad (6)$$

To understand this metric, consider the following. With prefetching enabled, it is possible for a demand request to merge with the corresponding prefetch request and vice-versa.[3] Note that since the throttle engine is inside the core, we count only intra-core merges (Figure 2a). Such merges indicate that prefetch requests are late. However, in contrast to CPU systems where late prefetches typically result in pipeline stalls, merged requests usually do not cause pipeline stalls, since such stalls can be avoided by switching to a different warp if there are other ready warps. Thus, merging in GPGPUs indicates a performance benefit rather than harm.

B. Training and Throttling Actions

Our adaptive throttling mechanism maintains the early eviction rate and merge ratio, periodically updating them and using them to adjust the degree of throttling.[4]

At the end of each period, we update the early eviction rate and merge ratio based on the *monitored* values during the period as well as the values from the previous period, according to Eq. 7 and Eq. 8.

$$Current(EarlyEviction) = Monitored(EarlyEviction) \quad (7)$$

$$Current(Merge) = \frac{Previous(Merge) + Monitored(Merge)}{2} \quad (8)$$

Table I
Throttling Heuristics.

Condition		Action
Early Eviction Rate	Merge	
High	-	No Prefetch
Medium	-	Increase throttle (fewer prefetches)
Low	High	Decrease throttle (more prefetches)
Low	Low	No Prefetch

The throttling degree varies from 0 (0%: keep all prefetches) to 5 (100%: no prefetch). We adjust this degree using the current values of the two metrics according to the heuristics in Table I. The early eviction rate is considered high if it is greater than 0.02, low if it is less than 0.01, and medium otherwise. The merge ratio is considered high if it is greater than 15% and low otherwise.[5] The prefetch engine in a core can only throttle prefetch requests originating from that core. Initially, the degree is set to a default value (we use '2' in our evaluation).

[3] Prefetch requests can be also merged with other prefetch requests.

[4] The length of a period is adjustable, and we use a period of 100,000 cycles in our evaluations.

[5] The threshold values for high and low are decided via experimental evaluations. We do not show those experimental results due to space constraints.

VI. Methodology

A. Simulator

Table II
THE BASELINE PROCESSOR CONFIGURATION.

Number of cores	14 cores with 8-wide SIMD execution
Front End	Fetch width: 1 warp-instruction/cycle, 4KB I-cache, stall on branch, 5 cycle decode
Execution core	Frequency: 900 MHz, in-order scheduling, Latencies modeled according to the CUDA manual [25] IMUL:16-cycle/warp, FDIV:32-cycle/warp, Others:4-cycle/warp
On-chip caches	Shared memory: 16KB sw-managed cache, 16 loads/2-cycle Constant/Texture cache: 1-cycle latency, 16 loads/2-cycle Prefetch cache: 16 KB, 8-way
DRAM	2 KB page, 16 banks, 8 channel, 57.6 GB/s bandwidth, Demand has higher priority than prefetch requests 1.2 GHz memory frequency, 900 MHz bus frequency, t_{CL}=11, t_{RCD}=11, t_{RP}=13
Interconnection	20-cycle fixed latency, at most 1 req. from every 2 cores per cycle

We use an in-house cycle-accurate, trace-driven simulator for our simulations. The inputs to the simulator are traces of GPGPU applications generated using GPUOcelot [7], a binary translator framework for PTX. We use a baseline processor/SM for our simulations based on NVIDIA's 8800GT [24], whose configuration Table II summarizes. In addition to modeling the multithreaded pipeline of the SMs, the simulator models the memory hierarchy in considerable detail. In particular, the simulator handles multiple in-flight memory requests from the same warp (Section II-B).

B. Benchmarks

Table III shows the 14 *memory-intensive* benchmarks that we evaluate, taken from CUDA SDK [22], Merge [18], Rodinia [3], and Parboil [1] suites. We classify as memory-intensive the benchmarks whose baseline CPI is 50% more than the CPI with a perfect memory. We categorize the benchmarks into three groups based on their characteristics. *Stride*-type benchmarks show strong stride behavior, including multidimensional patterns. *Mp*-type benchmarks are massively parallel benchmarks. They have a significantly high number of threads, i.e., # warps * 32. Typically these threads do not contain any loops, so the execution time of each thread is very short. These benchmarks are good candidates for inter-thread prefetching. Finally, *uncoal*-type benchmarks are applications with dominant uncoalesced memory accesses.

We calculate the maximum number of thread blocks allowed per SM, also shown in Table III, using the CUDA occupancy calculator, which considers the shared memory usage, register usage, and the number of threads per thread block.

Table IV summarizes the CPI of non-memory-intensive benchmarks from the suites we use.[6] Since these benchmarks are not memory intensive, hardware prefetching and even perfect memory do not affect their performance significantly.

[6]Note that we omit the benchmarks that are not sufficiently long, for example bandwidthTest and simplegl.

VII. Software Prefetching Results

This section evaluates our proposed MT-SWP against previously proposed software prefetching mechanisms, which are explained in Section II-C1. Note that *stride* is similar to conventional stride prefetching and the prefetched block will be stored in the prefetch cache. MT-SWP is the combination of stride and IP prefetching. Only the stride-type benchmarks contain register-based prefetching because the other evaluated benchmarks (mp-type and uncoal-type) do not contain any loops.

A. Non-adaptive GPGPU Prefetching

Figure 10 shows the speedup with different software prefetching mechanisms over the baseline binary that does not have any software prefetching. The results show that stride prefetching always outperforms register prefetching except in *stream*. The reason stride prefetching performs better than register prefetching is because register prefetching uses registers to store prefetched data, while stride prefetching uses a prefetch cache. However, in the case of *stream*, the instruction overhead due to prefetching (30% increase in instruction count) and late prefetches (90% of prefetches are late) offset the benefit of prefetching.

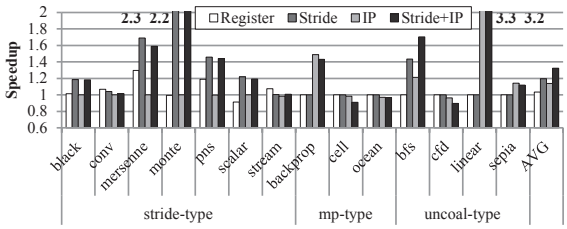


Figure 10. Performance of software GPGPU prefetching.

IP provides significant performance improvement for *backprop*, *bfs*, *linear*, and *sepia*, which are all mp-type and uncoal-type benchmarks. However, it degrades performance for *ocean* and *cfd*, for two reasons: 1) there are too many early prefetches: the instructions that source prefetched blocks are executed too late; and 2) there are too many redundant prefetches: warps generate prefetch requests for already executed warps.

Overall, static MT-SWP improves performance over stride prefetching by 12% and over register prefetching by 28%.

B. Adaptive Software Prefetching

Figure 11 shows the speedup of MT-SWP with prefetch throttling over the baseline. Throttling improves the performance of several benchmarks by reducing the number of early prefetches and bandwidth consumption, as shown in Figure 12. Prefetch throttling improves performance of *stream*, *cell*, and *cfd* by 10%, 5%, and 12%, respectively over MT-SWP (the bandwidth consumption for these three benchmarks is also reduced). With MT-SWP only, these benchmarks have

Table III
BENCHMARKS CHARACTERISTICS (BASE CPI: CPI OF BASE BINARY (CPI IS CALCULATED USING EQUATION IN [12]), PMEM CPI: CPI WITH PERFECT MEMORY SYSTEM, DEL LOADS: NUMBER OF DELINQUENT LOADS) / (BLACK:BLACKSCHOLES, CONV:CONVOLUTIONSEPARABLE, MERSENNE:MERSENNETWISTER, MONTE:MONTECARLO, SCALAR:SCALARPROD, STREAM:STREAMCLUSTER, OCEAN:OCEANFFT).

	black	conv	mersenne	monte	pns	scalar	stream	backprop	cell	ocean	bfs	cfd	linear	sepia
Suite	sdk	sdk	sdk	sdk	parboil	sdk	rodinia	rodinia	rodinia	sdk	rodinia	rodinia	merge	merge
# Total warps	1920	4128	128	2048	144	1024	2048	16384	21296	32768	2048	7272	8192	8192
# Blocks	480	688	32	256	18	128	128	2048	1331	16384	128	1212	1024	1024
Base CPI	8.86	7.98	7.09	13.69	18.87	19.25	18.93	21.47	8.81	62.63	102.02	29.01	408.9	149.46
PMEM CPI	4.15	4.21	4.99	5.36	5.25	4.19	4.21	4.16	4.19	4.19	4.19	4.37	4.18	4.19
# DEL Loads (Stride/IP)	3/0	1/0	2/0	1/0	1/1/	2/0	2/5	0/5	0/1	0/1	4/3	0/36	0/27	0/2
Type	stride	stride	stride	stride	stride	stride	stride	mp	mp	mp	uncoal	uncoal	uncoal	uncoal
# Max blocks/core	3	2	2	2	1	2	1	2	1	8	1	1	2	3

Table IV
CHARACTERISTICS OF NON-MEMORY INTENSIVE BENCHMARKS (HWP CPI: CPI WITH HARDWARE PREFETCHER).

Benchmarks	binomial	dwthaar1d	eigenvalue	gaussian	histogram	leukocyte	matrix	mri-fhd	mri-q	nbody	qusirandom	sad
Suite	sdk	sdk	sdk	rodinia	sdk	rodinia	sdk	parboil	parboil	sdk	sdk	rodinia
Base CPI	4.29	4.6	4.73	6.36	6.29	4.23	5.14	4.36	4.31	4.72	4.12	5.28
PMEM CPI	4.27	4.37	4.72	4.18	5.17	4.2	4.14	4.26	4.23	4.54	4.12	4.17
HWP CPI	4.25	4.45	4.73	5.94	6.31	4.23	4.98	4.33	4.31	4.72	4.12	5.18

many early prefetches. These harmful prefetches are successfully detected and removed by the throttling mechanism.

Overall, MT-SWP with throttling provides 4% benefit over MT-SWP without throttling, 16% benefit over stride prefetching (state-of-the-art software prefetching for GPGPUs), and 36% benefit over the baseline (no prefetching case).

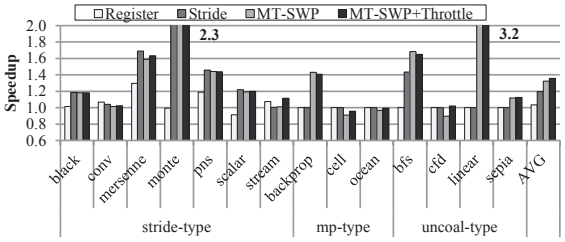
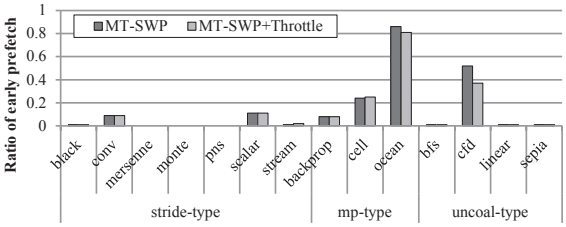
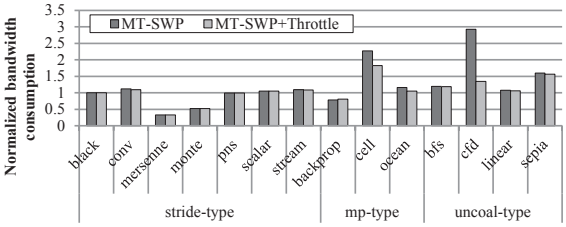


Figure 11. Performance of MT-SWP throttling.



(a) The ratio of early prefetches



(b) Bandwidth consumption of MT-SWP throttling normalized to no-prefetching case

Figure 12. Early prefetches and bandwidth consumption of MT-SWP throttling.

VIII. HARDWARE PREFETCHING RESULTS

In this section, we evaluate different hardware prefetchers, including MT-HWP. Note that for all hardware prefetchers, we use a prefetch distance of 1 and a prefetch degree of 1 as the default. Other distance configurations are evaluated in Section IX.

A. Previously Proposed Hardware Prefetchers

We first evaluate hardware prefetchers that were previously proposed for CPUs (Section II-C2). Table V summarizes the configurations of the prefetchers we evaluated.

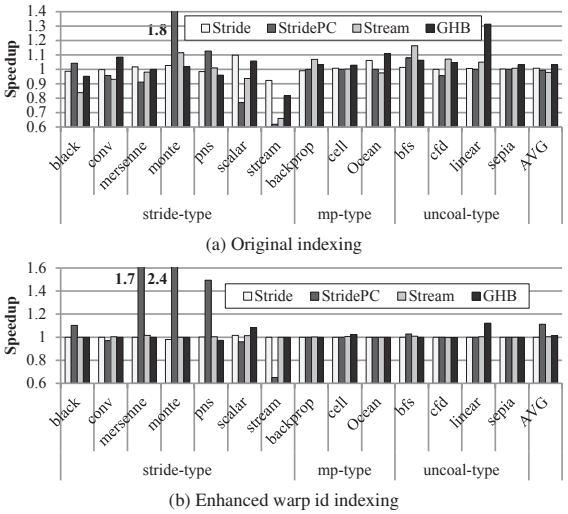
Table V
DESCRIPTION OF EVALUATED HARDWARE PREFETCHERS.

Prefetcher	Description	Configuration
Stride	RPT Stride prefetcher [13]	1024-entry, 16 region bits
StridePC	Stride prefetcher per PC [4, 11]	1024-entry
Stream	Stream prefetcher [29]	512-entry
GHB	AC/DC GHB prefetcher [14, 21]	1024-entry GHB, 12-bit CZone, 128-entry Index Table

As we argued in Section III-B1, all prefetchers should be trained using the warp id for effective prefetching. However, in this section, for each prefetcher, we evaluate both its naïve version (i.e., mechanism as it was proposed) and an enhanced version that uses warp ids.

Figure 13 shows the speedup with hardware prefetchers over the no prefetching case. Interestingly, in Figure 13a, the naïve versions show both positive and negative cases. For the positive cases, even though the prefetchers are not trained by warp id, the same warp is executed for a long enough period to train the prefetcher. The negative cases (*black* and *stream*) are due to poor training because of accesses from many warps as discussed in Section III-B.

Across the different benchmarks, the performance of the enhanced versions of the prefetchers is in general more stable than that of the naïve versions as shown in Figure 13b. Also, not many positive or negative cases are seen; in other



(a) Original indexing

(b) Enhanced warp id indexing

Figure 13. Performance of hardware prefetchers.

words, the enhanced versions of hardware prefetchers are ineffective. This is because the effective sizes of prefetch tables are reduced by the number of active warps. However, StridePC prefetcher is an exception to the general ineffectiveness of hardware prefetchers, providing 10%, 74%, 142%, and 49% improvement for *black*, *mersenne*, *monte*, and *pns*, respectively. This is due to its good prefetch accuracy and coverage for these benchmarks, especially for *mersenne* and *monte* (both accuracy and coverage are near 100%). StridePC prefetcher degrades the performance of *stream* by 35% because of too many late prefetches (93%). GHB improves performance of *scalar* and *linear* by 12% and 8%. However, GHB's low coverage (at most 10% in *scalar*) prevents further performance improvement.

Thus, we can conclude that hardware prefetchers cannot be successful for many-thread workloads without warp id indexing (to improve accuracy) and better utilization of reduced effective prefetcher size (to provide scalability). Hence, the rest of the paper uses GHB and StridePC prefetchers trained with warp ids for comparisons.

B. Many-thread Aware Hardware Prefetcher (MT-HWP)

Since MT-HWP consists of multiple tables, we evaluate different configurations of MT-HWP to show the benefit resulting from each table. Note that the enhanced version of StridePC prefetcher is essentially the same as the PWS table only configuration. In our evaluations, we use 32-entry PWS table, 8-entry GS table, and 8-entry IP table.

Figure 14 shows the performance improvement with different MT-HWP configurations. While the PWS table provides significant performance improvement for some benchmarks, the GS table does not significantly add to the improvement provided by the PWS table. However, the main benefit of the GS table comes from its power efficiency and reduction of the PWS table size. Once a trained stride is promoted to the GS table, accesses from the same PC do not need to access the PWS table unless the entry is evicted from the GS table. Also, the GS table has a much smaller number of entries and is indexed only by the PC address, whereas the PWS table requires PC address and warp id. The GS table reduces the number of PWS accesses by 97 % on average for the stride-type benchmarks.

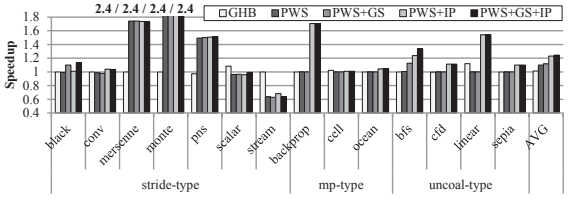


Figure 14. Performance of MT-HWP vs. GHB.

Inter-thread prefetching significantly improves the performance of *backprop*, *bfs*, *cfd*, and *linear*. However, IP does not provide any performance improvement for the stride-type benchmarks even though these benchmarks show stride behavior between warps. Since PWS has higher priority than IP, all prefetches are covered by PWS .

Table VI summarizes the hardware cost of MT-HWP. Compared with other hardware prefetchers, even when using fewer table entries, MT-HWP can be very effective. Since the GS table stores all promoted strides, it can reduce training time and also help tolerate the eviction of trained entries from the PWS table. Overall, MT-HWP provides 15%, 24%, and 25% improvement over PWS only (enhanced StridePC), GHB, and the baseline (no-prefetching), respectively.

Table VI
HARDWARE COST OF MT-HWP.

Table	Fields	Total bits
PWS	PC (4B), wid (1B), train (1b), last (4B), stride (20b)	93 bits
GS	PC (4B), stride (20b)	52 bits
IP	PC (4B), stride (20b), train (1b), 2-wid (2B), 2-addr (8B)	133 bits
Total	32×93 (PWS) + 8×52 (GS) + 8×133 (IP)	557 Bytes

C. Prefetch Throttling for MT-HWP

In this section, we evaluate MT-HWP with prefetch throttling along with the feedback-driven GHB [31] and the StridePC prefetcher with throttling. Based on prefetch accuracy, the feedback-driven GHB (GHB+F) can control the prefetch degree. In other words, it can launch more prefetches when its accuracy is higher than a certain threshold. The StridePC with throttling reduces the number of generated prefetches based on the lateness of the earlier generated prefetches. The StridePC and MT-HWP mechanisms with throttling are denoted as StridePC+T and MT-HWP+T, respectively.

Figure 15 shows the speedup over the baseline (no prefetch case) for mechanisms without and with feedback/throttling. GHB+F significantly improves the performance of *monte*,

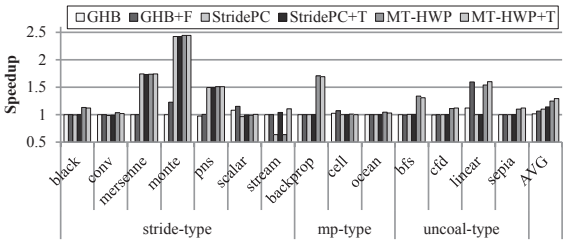


Figure 15. Performance of MT-HWP throttling.

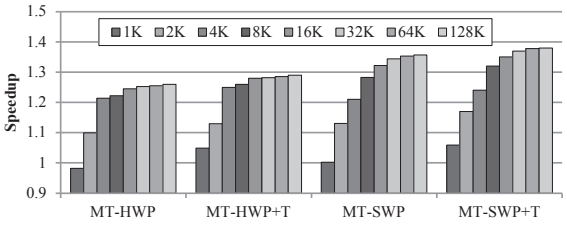


Figure 16. Sensitivity to prefetch cache size (+T: Throttling).

scalar, *cell*, and *linear*, compared to GHB. Since its accuracy for these benchmarks is fairly high (more than 50%), GHB+F can generate more useful prefetch requests. However, its benefit is still much lower than that of MT-HWP and MT-HWP+T due to its low coverage. Compared to StridePC, StridePC+T improves the performance of *stream* only. For *stream*, StridePC+T reduces the number of late prefetches resulting in a performance improvement of 40%. MT-HWP+T improves the performance of *stream* by 46% over MT-HWP for the same reason. MT-HWP+T also improves the performance of *linear* by 5% over MT-HWP. For other benchmarks, MT-HWP+T is unable to improve performance significantly over MT-HWP because prefetching already brings about enough positive effect on the performance.

Throttling MT-HWP eliminates the negative effects of MT-HWP, especially in *stream*. *cfd*, *linear*, and *sepia* also get some small performance improvement through throttling.

On average, MT-HWP with throttling provides a 22% and 15% improvement over GHB with feedback and StridePC with throttling, respectively. Overall, adaptive MT-HWP provides a 29% performance improvement over the baseline.

IX. Micro-architecture Sensitivity Study

A. Prefetch Cache Size

Figure 16 shows the speedup as we vary the size of the prefetch cache from 1KB to 128KB for MT-HWP and MT-SWP without and with throttling.

As cache size increases, the rate of early evictions decreases and performance increases. Unfortunately, with a 1KB prefetch cache, prefetching actually degrades performance. However, with throttling, even with a 1KB prefetch cache, both MT-HWP and MT-SWP provide performance improvement.

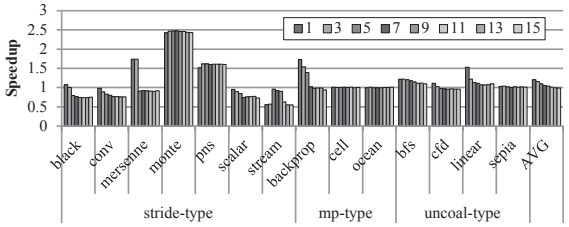


Figure 17. Sensitivity of MT-HWP to prefetch distance.

On the other hand, we also notice that the impact of throttling decreases as the cache size increases. The throttling mechanism throttles the number of prefetches based on the eviction rate from the prefetch cache. As mentioned earlier, the eviction rate decreases as the cache size increases; hence at larger cache sizes there is less scope for throttling.

B. Prefetch Distance

Figure 17 shows the speedup for MT-HWP over the baseline as its prefetch distance is varied from 1 (baseline) to 15. For most benchmarks, a prefetch distance of 1 shows the best performance. This is not surprising because GPGPU applications behave differently from CPU applications. The main reason for using aggressive (large) prefetch distances in CPU applications is to reduce late prefetches when the computation in a loop is not sufficient to hide prefetch request latency. Because of the large number of in-flight warps and the switching of execution among them, the number of late prefetches is usually low for GPGPU applications. By increasing the distance, many prefetches become early because the prefetch cache is not able to hold all prefetched blocks from many warps until they are needed. One exception to this behavior is the *stream* benchmark. At a distance of 1, 93% of prefetches are late, but by increasing the aggressiveness of prefetching, we see significant performance improvement. However, the performance benefit decreases as we continue to increase distance (after prefetch distance 5) as more prefetches become early.

C. Number of Cores

In order to see whether prefetching continues to provide a benefit as the number of cores increases, we conduct experiments with the number of cores varying from 8 cores to 20 cores (baseline has 14 cores). Note that the DRAM bandwidth was kept the same in all the experiments. As shown in Figure 18, the performance improvement decreases slightly as the number of cores increases. For example, the performance of MT-SWP decreases from 137% over no-prefetching for 8 cores to 131% over no-prefetching for 20 cores. The drop in performance can be attributed to the increasing contention in the interconnect and DRAM due to the increase in the number of memory requests (active warps) in the system. However, we expect that as the number of cores increase, the DRAM bandwidth (and interconnect bandwidth) will also increase;

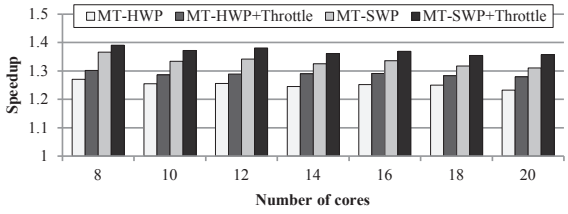


Figure 18. Sensitivity to number of cores.

thus contention will not increase significantly and prefetching will be viable even if the number of cores is increased.

X. Related Work

We summarize prior work on prefetching for GPGPU applications and dynamic adaptation techniques for prefetching.

A. Prefetching for GPGPU Applications

Ryoo et al. [28] showed that prefetching into registers can yield benefits through binding operations. Recently, Yang et al. [34] proposed a compiler algorithm for register based prefetching along with several other compiler optimizations for GPGPU applications. Tarjan et al. [32] proposed the "diverge on miss" mechanism for improving memory access latency tolerance of SIMD cores with caches. Meng et al. [19] proposed dynamic warp subdivision to reduce performance degradation due to divergent warps in SIMD cores with caches. Although prefetching is not the main component of these two mechanisms, to some extent, both rely on the prefetching effect provided by the early execution of some threads in a warp to achieve performance improvement.

B. Dynamic Adaptation of Prefetching Policy

Adaptive prefetching has been studied by several researchers. Much of prior work uses prefetching accuracy as the main feedback metric. For example, Dahlgren et al. [5] measured the prefetching accuracy and adjusted the prefetch distance based on the accuracy. Srinath et al. [31] proposed a more sophisticated feedback mechanism. In their work, not only the prefetching accuracy but also the timeliness and cache pollution effect are used to reduce the negative effects of prefetching.

Some research has focused on throttling prefetch requests instead of controlling the aggressiveness of prefetching. A throttling mechanism controls only the priority of requests (or even drops requests) but does not control the aggressiveness of prefetching. Zhuang and Lee [35] proposed hardware based cache pollution filters for processors employing aggressive prefetching (both hardware and software). They classified useful prefetches as good prefetches while early and wrong prefetches were classified as bad prefetches. A history table (based on PC or request address) stores information as to whether a prefetch is good or bad. If a prefetch is considered as bad, it is dropped immediately. Ebrahimi et al. [9] proposed a coordinated throttling of multiple prefetchers in a core for prefetching linked data structures. They monitor accuracy and coverage of each prefetcher and control the aggressiveness. Ebrahimi et al. [8] also proposed a prefetch throttling mechanism among multiple prefetchers in a CMP system. In a CMP system, very aggressive prefetching (even though its accuracy is very high) from one application may hurt other applications running together with it because of inter-core interference with prefetch and demand accesses from other cores. Thus, they propose a hierarchical prefetch coordination mechanism that combines per-core (local) and prefetch-caused inter-core (global) interference feedback to maximize the benefits of prefetching on each core while maintaining overall system performance.

Lee et al. [16] proposed Prefetch-Aware DRAM Controllers that dynamically prioritize between prefetch and demand requests based on the accuracy of the prefetcher. The proposed DRAM controllers also drop prefetch requests that have been resident in the DRAM request buffers for longer than an adaptive threshold (based on the accuracy of the prefetcher) since such requests are likely to be useless. Caragea et al. [2] proposed a software prefetching algorithm called resource-aware prefetching (RAP). The resource under consideration is the number of MSHR entries. Depending on the number of MSHR entries in the core, the distance of prefetch requests is adjusted (reduced) so that prefetches are issued for all loop references.

There are three major differences between these previous feedback mechanisms and our work: 1) these feedback algorithms use accuracy as one of the main feedback metrics, so they do not work well when prefetch accuracy is almost 100%. These mechanisms control the aggressiveness of prefetching and when there is 100% accuracy, they consider all prefetch requests as useful. In GPGPUs, even if prefetch requests are 100% accurate, they can be harmful because of resource contentions. 2) These feedback mechanisms are not designed for supporting hundreds of threads. 3) All these feedback mechanisms, except Caragea et al., are only for hardware prefetching and not for software prefetching.

In addition to these differences, our prefetching mechanisms contain a new prefetch algorithm (inter-thread prefetching) that is specifically designed for GPGPUs. None of the previous work exploited these characteristics.

C. Memory System Optimizations in Vector Architectures

In classical vector architectures, a single instruction operates on arrays of data elements. Thus, similar to SIMT GPUs, vector architectures can generate several memory requests due to a single instruction. Accesses in vector processors can be either strided (unit stride and non-unit stride) or scatter-gather accesses. Non-unit stride and scatter-gather accesses are equivalent to uncoalesced accesses in SIMT machines. Vector architectures typically did not use caches due to large working sets, low locality of access, and the use of highly interleaved memory. The granularity of memory accesses

in such vector architectures without caches was individual data elements. The strategy adopted by vector architectures to reduce memory access latencies was to develop more effective mechanisms for interleaving [30] and to reorder memory requests to concurrently service memory accesses as much as possible [6, 27].

Some vector processors even adopted caches [33]. Fu and Patel specifically discussed the benefits of using caches for uncoalesced loads (exploiting spatial locality) and also employed stream/stride prefetchers [10]. However, their proposed mechanisms were also very simple and did not provide any scalability.

XI. Conclusion

The key ideas behind our MT-prefetching schemes are (a) per-warp-training and stride promotion, (b) inter-thread prefetching, and (c) adaptive throttling. The first two introduce the idea of *cooperative* prefetching among the threads, which exploits a fundamental property of current GPGPU applications, namely, the presence of many threads in-flight. The third idea, adaptive throttling, solves the observed problem that even with 100% accuracy, prefetching can still degrade performance in GPGPUs, again precisely because of the large numbers of concurrent threads. Putting these ideas together, our evaluation shows that basic MT-prefetching improves performance while throttling reduces or eliminates any possible negative effects.

These ideas represent just the beginning of the study of MT-prefetching schemes for GPGPUs and other many-core SIMT architectures. For instance, our baseline assumes a relatively simple memory hierarchy, whose designs continue to evolve. In future work, we will extend our mechanisms for more complex hierarchies.

More broadly, MT-prefetching schemes extend GPGPU designs with additional latency-tolerance mechanisms that can, in principle, serve much more diverse applications. Thus, a major question moving forward is what additional or new applications might now become GPGPU applications in an GPU + MT-prefetching system.

Acknowledgements

Many thanks to Dilan Manatunga, Pranith D. Kumar, Sangho Lee, Changhee Jung, Chang Joo Lee, Sunpyo Hong, Eiman Ebrahimi, and other HParch members and the anonymous reviewers for their suggestions and feedback on improving the paper. We gratefully acknowledge the support of the National Science Foundation (NSF) CCF-0903447, NSF/SRC task 1981, NSF CAREER award 0953100, Georgia Tech Raytheon Faculty Fellowship, the U.S. Department of Energy including Sandia National Laboratories, the Defense Advanced Research Projects Agency, Intel Corporation, Microsoft Research, and the equipment donations from NVIDIA.

References

[1] Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.
[2] G. C. Caragea, A. Tzannes, F. Keceli, R. Barua, and U. Vishkin. Resource-aware compiler prefetching for many-cores. In *ISPDC-9*, 2010.
[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*, 2009.
[4] T.-F. Chen and J.-L. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Trans. Computers*, 44(5):609–623, 1995.
[5] F. Dahlgren, M. Dubois, and P. Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.
[6] A. Dal Corral and J. Llaberia. Minimizing conflicts between vector streams in interleaved memory systems. *IEEE Transactions on Computers*, 48(4):449–456, 1999.
[7] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT-19*, 2010.
[8] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO-42*, 2009.
[9] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA-15*, 2009.
[10] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *ISCA-18*, 1991.
[11] W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO-25*, 1992.
[12] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.
[13] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS-18*, 2004.
[14] K. J.Nesbit and J. E.Smith. Data cache prefetching using a global history buffer. In *HPCA-10*, 2004.
[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.
[16] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware dram controllers. In *MICRO-41*, 2008.
[17] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *ICPP-16*, 1987.
[18] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, 2008.
[19] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA-37*, 2010.
[20] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, 1991.
[21] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT-13*, 2004.
[22] NVIDIA. CUDA SDK 3.0. http://developer.download.nvidia.com/object/cuda_3_1_downloads.html.
[23] NVIDIA. Fermi: Nvidia's next generation cuda compute architecture. http://www.nvidia.com/fermi.
[24] NVIDIA. Geforce 8800 graphics processors. http://www.nvidia.com/page/geforce_8800.html.
[25] NVIDIA Corporation. *CUDA Programming Guide, V3.0*.
[26] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA-21*, 1994.
[27] M. Peiron, M. Valero, E. Ayguade, and T. Lang. Vector multiprocessors with arbitrated memory access. In *ISCA-22*, 1995.
[28] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO*, 2008.
[29] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4-5):505–522, 2005.
[30] G. Sohi. High-bandwidth interleaved memories for vector processors-a simulation study. *IEEE Transactions on Computers*, 42(1):34 –44, jan 1993.
[31] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, 2007.
[32] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.
[33] Q. Yang and L. W. Yang. A novel cache design for vector processing. In *ISCA-19*, 1992.
[34] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI-10*, 2010.
[35] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP-32*, 2003.