APOGEE: Adaptive Prefetching On GPUs for Energy Efficiency

Ankit Sethia[1] Ganesh Dasika[2] Mehrzad Samadi[1] Scott Mahlke[1]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
Email: {asethia, mehrzads, mahlke}@umich.edu

[2]ARM R&D
Austin, TX
Email: ganesh.dasika@arm.com

***Abstract*—Modern graphics processing units (GPUs) combine large amounts of parallel hardware with fast context switching among thousands of active threads to achieve high performance. However, such designs do not translate well to mobile environments where power constraints often limit the amount of hardware. In this work, we investigate the use of prefetching as a means to increase the energy efficiency of GPUs. Classically, CPU prefetching results in higher performance but worse energy efficiency due to unnecessary data being brought on chip. Our approach, called APOGEE, uses an adaptive mechanism to dynamically detect and adapt to the memory access patterns found in both graphics and scientific applications that are run on modern GPUs to achieve prefetching efficiencies of over 90%. Rather than examining threads in isolation, APOGEE uses adjacent threads to more efficiently identify address patterns and dynamically adapt the timeliness of prefetching. The net effect of APOGEE is that fewer thread contexts are necessary to hide memory latency and thus sustain performance. This reduction in thread contexts and related hardware translates to simplification of hardware and leads to a reduction in power. For Graphics and GPGPU applications, APOGEE enables an 8X reduction in multi-threading hardware, while providing a performance benefit of 19%. This translates to a 52% increase in performance per watt over systems with high multi-threading and 33% over existing GPU prefetching techniques.**

***Keywords*—GPU, Energy Efficiency, Prefetching, Throughput Processing**

I. Introduction

The demand for rendering increasingly real world scenes is leading to a surge in computational capability of GPUs. The combination of programmability and high computational power have made GPUs the processor of choice even for throughput oriented scientific applications. High throughput is achieved in GPUs by having hundreds of processing units for floating point computations. To keep all these processing units busy, GPUs use high degrees of multi-threading to hide latency of global memory accesses and long latency floating point instructions. For example,the NVIDIA GTX 580 has 512 processing units and can use over 20,000 threads to maintain high utilization of the compute resources [19]. The basic approach that GPUs use is to divide the given work into several chunks of small, independent tasks and use fine-grained multi-threading to hide any stall in the pipeline. They can hide several hundreds of cycles of latency if they are provided with enough work.

The support for such levels of multi-threading and fast context switching requires deployment of considerable hardware resources such as large register files. For example, the GTX-580 has 2 MB of on-chip register file. In addition to the register files, more resources are required to orchestrate the fine-grained multi-threading and maintain thread state including divergence stacks and warp schedulers. An unwanted cost of this design style is high power consumption. While providing teraflops of compute, modern graphics cards can consume 100W or more of power. As modern systems are becoming power limited [12], such a trajectory of power consumption is a difficult challenge to meet on future desktop devices.

Furthermore, the increasing trend of using scientific style compute, visually rich applications and games on mobile GPUs, will continue to push the performance needs of GPUs in mobile domain as well. Straight-forward down scaling of desktop GPUs for mobile systems often results in insufficient performance as the power budget for mobile systems is a small fraction of desktop systems. Thus, improvements in the inherent energy efficiency are required so that performance can be scaled faster than energy consumption in both the domains.

To attack the energy efficiency problem of GPUs, we propose APOGEE (Adaptive Prefetching On GPUs for Energy Efficiency), that leverages prefetching to overlap computation with off-chip memory accesses. Successful prefetching reduces the exposed memory latency and thereby reduces the degree of multi-threading hardware support necessary to sustain utilization of the datapath and thus provide high performance. As the degree of multi-threading support is reduced, power efficiency is increased as datapath elements can be correspondingly reduced in size (e.g, register file, divergence stacks, etc.).

Prefetching is not a novel concept, it has been around for years and is used in commercial CPUs. Conventional wisdom suggests that prefetching is more power hungry due to unnecessary off-chip accesses and cache pollution. We postulate that this wisdom is untrue for GPUs. On the contrary, prefetching improves the energy efficiency of GPUs for two reasons. First, the reductions in memory latency provided by prefetching can directly lead to reductions in hardware size and complexity that is not true for CPUs. Second, prefetching efficiency for the GPU can be increased well beyond those seen on the CPU because the address patterns for graphics and scientific applications commonly run on GPUs are highly regular.

In order to reduce the degree of necessary multi-threading in an energy efficient way, APOGEE has to address three chal-

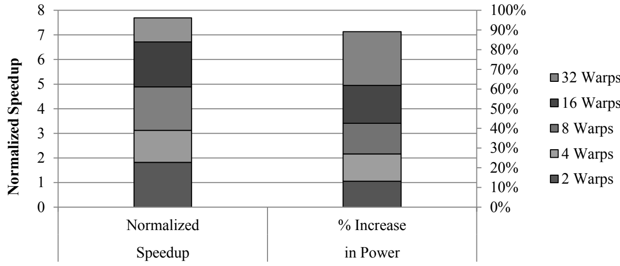

Fig. 1: Increase in speedup and increase in power as number of warps are increased

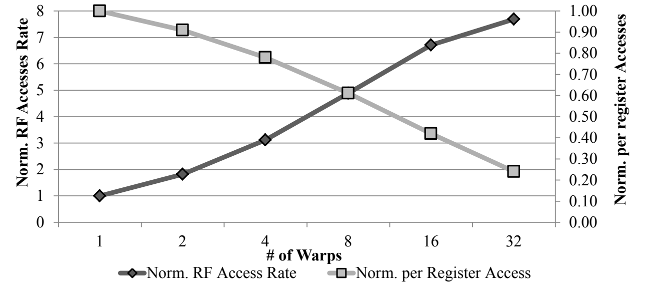

Fig. 2: Increase in number of register file access rate and degradation in per register accesses. This shows that with SIMT more hardware is required, but the utilization of the hardware reduces significantly.

lenges. Firstly, it should be able to predict the address patterns of GPU applications - both graphics and scientific. Prefetching for scientific applications on CPUs is a well studied area, where stride prefetching is known to be effective [3], [5], [7]. In the GPU domain, such a technique for scientific applications will not be efficient for thousands of threads as explained in Section II-B. On the other hand, for graphics applications, strided patterns are not the only common access patterns [15]. Secondly, it should be able to prefetch in a timely fashion, such that the data arrives neither too early nor too late to the core. For energy-efficiency, timeliness of prefetching is an important factor, as it reduces the need for storing correct but untimely prefetched data. Thirdly, the prefetcher should not over burden the bandwidth to memory, as aggressive prefetching can cause actual read requests to be delayed.

APOGEE exploits the fact that threads should not be considered in isolation for identifying data access patterns for prefetching. Instead, adjacent threads have similar data access patterns and this synergy can be used to quickly and accurately determine the access patterns for all the executing threads. APOGEE exploits this characteristic by using address streams of a few threads to predict the addresses of all the threads leading to a low hardware overhead design as well as a reduced number of prefetch requests to accomplish the necessary prefetching. The key insight that APOGEE uses to maintain the timeliness of prefetching is by dynamically controlling the distance of prefetching on a per warp basis. The same technique can also be used to control the aggressiveness of prefetching.

In order to solve the challenges mentioned above, this paper makes the following contributions:

- We propose a low overhead prefetcher which *adapts* to the address patterns found in both graphics and scientific applications. By utilizing an intra-warp collaboration approach, the prefetcher learns faster and by sharing this information amongst warps, the area/power overhead of the prefetcher is lower than traditional prefetchers.
- We show that APOGEE can *dynamically adjusts the distance* of prefetching depending on the kernel for a GPU architecture. This provides timely and controlled prefetching for variable memory latency.
- We show that the required degree of multi-threading can be significantly reduced and thus energy efficiency is increased without any loss in performance for both Graphics and GPGPU (scientific) applications.

II. Techniques to Hide Large Memory Latency

This section discusses the shortcomings of the two major prevalent techniques used for hiding/reducing memory access latency: i) High-degree of multi-threading as in Single Instruction Multiple threads (SIMT) execution model and ii) Traditional prefetching.

A. High degree of multi-threading

GPU Performance and Power. Figure 1 illustrates the change in speedup and power consumed with increasing the number of warps from two to 32 on a GPU like architecture. The methodology for obtaining these results is explained in Section IV. The baseline is a system with 1 warp implying no multi-threading. As the number of maximum concurrent warps is increased, performance increases because computation by concurrent warps hides the latency of access to memory. With 32 warps, around 85% of the total cycles are reduced as compared to the baseline. Adding more than 4 warps reduces a smaller fraction of cycles as many of the stalls are already hidden, leading to diminishing returns. The right bar in Figure 1 shows the increase in power as the number of warps are increased. Even though 16 and 32 warps provide around one-third of the speedup, the corresponding increase in power is half of the total increase in power. This increase in power is due to the large number of resources added to support the high degree of multi-threading.

GPU Underutilization. Figure 2 shows the variation in normalized access rate of the register file on the primary vertical axis and normalized per register accesses of the register file in the secondary vertical axis of an SM (Streaming Multiprocessor in NVIDIA terminology) as the number of maximum active warps per SM are increased. All values are normalized to the case when the SM has one warp. As more warps are added, the access rate of the register file also increases as the number of accesses is same but the total time is reduced. However, as more warps are added to the SM, the size of the register file is also increased. The size of the register file with 32 maximum warps is 32 times the baseline case of 1 warp. While the access rate of the register file increases to 7.69 times for 32 warps, the overall number of access per register access is down to 0.24 times of the 1 warp case. With the decreased number of accesses per register, several thousand registers are underutilized and leaking power.

Similary, when a warp is added to the system, register files, branch divergence book-keeping support, warp schedulers, scoreboarding, etc. need extra capacity. While the strategy

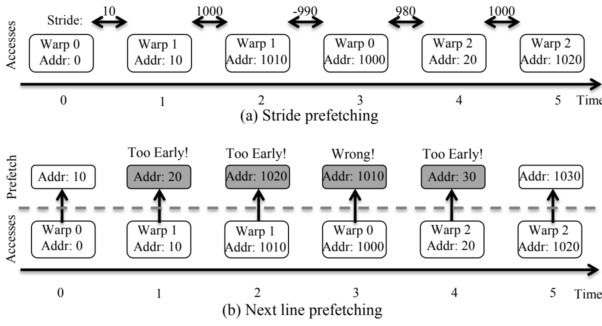

Fig. 3: Examples to demonstrate the ineffectiveness of common CPU prefetching techniques on GPU (a) stride prefetching and (b) next-line prefetching. These techniques do not work well due to the unordered execution of warps.

of high multi-threading keeps the compute units busy, the additional hardware mentioned above has lower utilization and hence increased power consumption. This demonstrates that the design of GPUs with heavy multi-threading has underutilized hardware which can lead to a significant increase in power consumption due to leakage.

B. Traditional CPU Prefetching

Prefetching has been studied extensively for multi-processors. However, although common prefetching techniques work well for CPUs with limited numbers of threads, for prefetching memory accesses of thousands of threads, those techniques cannot predict the memory access pattern correctly. The rest of this section shows why common CPU prefetching techniques such as stride prefetching and next-line prefetching do not work efficiently for GPUs.

Stride Prefetching. One of the most common CPU prefetching techniques is stride prefetching which focuses on array-like structures, computes the stride between accesses, and uses that to predict the next accesses. However, this technique is not efficient on GPUs due to interleaved memory accesses from different warps. Figure 3(a) shows an example of memory addresses of different warps. The difference between addresses of two consecutive memory requests is shown in the figure. Although there is a constant stride between two accesses within a warp, a stride prefetcher only sees a random pattern and as a result, it cannot compute the stride correctly. Therefore, stride prefetching technique is not a good choice for GPU architecture due to interleaved memory accesses.

Next-line Prefetching. Another common CPU prefetching technique is next-line prefetching which fetches the next sequential cache line on a cache miss. Figure 3(b) shows an example of next line prefetching on a GPU. In this example, cache line size is equal to 10B. There are two problems with this scheme for GPUs. First, next-line prefetcher may fetch an address after the actual memory access. For example, after the fourth memory access, the prefetcher brings address 1010 to the cache. However, this address is already accessed by the third access so prefetching does not help. Secondly, the prefetcher might prefetch an address far ahead of actual memory request. This may evict needed data from the cache before it was used. For example, the second memory access prefetches the address 20, but this address will be accessed in the future. Overall, the two prevalent CPU prefetching techniques will be inefficient on a GPU.

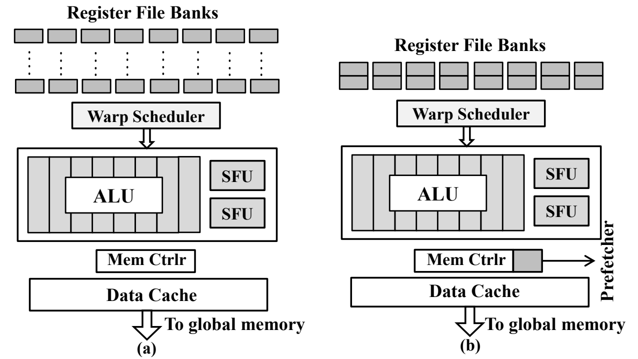

Fig. 4: (a) Traditional SM (b) APOGEE SM with prefetching in data cache and reduced register bank size.

III. APOGEE

To hide memory access latency in an energy efficient way, APOGEE adopts new prefetching techniques that are tailored specifically for GPU architectures in order to improve the prefetching efficiency. With APOGEE these prefetching techniques, underutilized hardware required for multi-threading can be scaled back which results in improved energy efficiency.

Figure 4 compares the design of a traditional SM with an APOGEE SM. APOGEE interacts with the memory system and looks at the memory access pattern between the threads of a warp. If it can detect a consistent pattern in the addresses accessed by adjacent threads, it stores various characteristics of that load. These characteristics are the Program Counter (PC), the address accessed by one thread, its offset with the adjacent thread and how consistent the offset is across adjacent threads. Once APOGEE is confident of the access pattern, it prefetches data into the data cache as shown in Figure 4(b). The requests are sent from the data cache to the global memory.

In GPUs, APOGEE exploits two important features: Firstly, the SIMD nature of the pipeline forces the same load instruction to be executed by all threads in the warp. This helps in the characterization of the access pattern of the load, as every thread in the warp provides a sample point for the detection of the pattern. Secondly, adjacent threads in a warp have adjacent thread indices, so the difference between the memory addresses accessed by these threads can provide a regular offset.

For APOGEE to utilize prefetching for reduction of hardware, it should be able to predict the access pattern found in Graphics and GPGPU applications. The major memory access pattern found in these applications are generally a linear function of the thread index of the thread accessing memory. This occurs because distribution of work in a kernel is done primarily using thread index. Since, adjacent threads differ in thread index by 1, the offset between the addresses accessed by adjacent threads is fixed. This constant difference is valid for all adjacent threads in a SIMT load instruction. We call such an access pattern as *Fixed Offset Address(FOA)*. The prefetcher for FOA is explained in Section III-A.

Apart from predicting the correct address accessed by a warp in the next access, a major factor in prefetching performance is the timeliness of the prefetch. If a prefetch is made too late, then the next access of that load would be a miss in

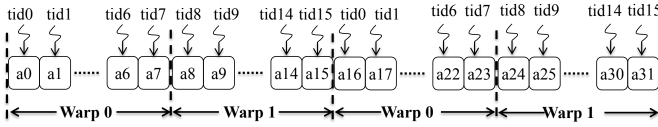

Fig. 5: Example of processing an array of elements in SIMT model with 2 warps

the cache even though a prefetch request for the same address has been made. On the other hand if the prefetch is made too early, the data will be in cache while it is not needed and may evict useful data that has more immediate reuse. APOGEE maintains the state of prefetching and adjusts the distance of prefetching dynamically as explained in Section III-A. A static software/hardware prefetching solution which uses a constant distance of prefetching for timeliness, will not be a good solution for GPUs. The problem of timeliness of prefetching in GPUs is exacerbated by the fact that the look-ahead distance is not only a function of the kernel size, but also of the number of warps and scheduling. APOGEE dynamically adapts to individual memory accesses and adjusts how far ahead data should be prefetched based on that instruction only, so it can prefetch in a timely fashion.

Since APOGEE reduces hardware support for high degree multi-threading, the latency hiding capacity of the system through SIMT execution is reduced. Therefore, acccess patterns that are not frequent, but miss in cache, can also have significant impact on performance, unless, the prefetcher is able to prefetch those access patterns. For graphics applications, moderate number of accesses are made to the same address by all threads. We call these memory accesses *Thread Invariant Access(TIA)*. As the addresses in TIA are same, predicting these addresses is trivial. In prefetching for TIA accesses, APOGEE focuses on timeliness of prefetching rather than prediction accuracy. APOGEE uses a novel prefetching method which finds an earlier load at which to prefetch the thread invariant address. Details of TIA prefetching is explained in Section III-B.

A. FOA Prefetching

Predictability. FOA accesses are generally predictable as they are based on thread index. APOGEE exploits this predictablility of streaming accesses. Figure 5 shows an example of accesses to various regions of an array by two warps in the SIMT model. It also shows the accesses made by each thread in a warp. In this example there are eight threads per warp, with a total of 16 threads. Using a unique thread index, a thread can access some element of the array. Programs in SIMT execution are modelled such that adjacent threads access adjacent memory locations. So after processing the first 16 elements, tid 0, moves to processing the 17th element and other threads follow the same pattern. In this example, during its lifetime, a thread with tid n processses $n+i*(total_threads)$ elements, where i is a factor which represents the current section of data being processed. This demonstartes that the addresses accessed by a thread are predictable for FOA accesses.

In this example, each thread accesses data elements as a linear function of the thread index. However, the access can be a complex function of the thread index as well. Generally, the function is rarely complicated because if the addresses are not

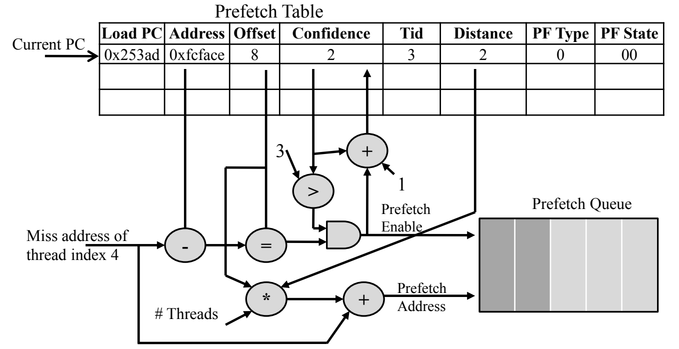

Fig. 6: Fixed Offset Address prefetcher design

contiguous, uncoalesced memory access can lead to a heavy performance penalty. Therefore, for performance reasons, most programs access addresses that are usually simple functions of the thread index. For the FOA pattern, the offset between the memory accesses of adjacent threads in a warp is confirmed by using all the threads in the warp. After that, it can predict the addresses accessed in the future iterations of a warp by using the number of active threads in the warp, the number of active warps, and the validated offset. Doing this for all the warps will result in the ability to prefetch all the data required for the next iteration of all of the threads.

Prefetcher Implementation. APOGEE uses FOA prefetching where access patterns between threads in a warp for a load are used to decide the address that will be accessed by the next access by these threads. The structure of the FOA prefetcher is shown in Figure 6. It consists of a Prefetch Table and other logic to check the offset between addresses of different threads in a warp. When the warp makes a load request, the access requests of all the threads in the warp are visible to the prefetcher. The load address of the first active thread is stored in the address column and the thread index is updated. Then, with the address and thread index of the second active thread, APOGEE computes the difference in addresses and difference in thread indices between the two threads. The offset is then calculated to be the ratio of the two quantities. We also update the address column and thread index column with the address of the second thread and its thread index respectively.

For the next active thread, APOGEE computes the offset and compares it with the previous offset. If they are equal, APOGEE increases the confidence of that entry. APOGEE does similar operation for the remaining threads in the warp. At every step a ratio is taken between the difference in address and difference in thread index to calculate the offset. This is done because a thread might be inactive due to branch divergence and no load request will be made from that thread. In such a case, the difference of the next active thread, with the address stored in the table will be two times the offset. So a ratio is taken between the quantities instead. If the confidence of the load is fewer than two less than the number of active threads in the warp, then we conclude that the offset found is indeed the characteristic of the load.

For a system with one warp, and N threads per warp, once the offset has been fixed, we can prefetch the data of the next access. By multiplying the offset with i (where i varies from 0 to $N-1$) and adding to the last stored address in the prefetch

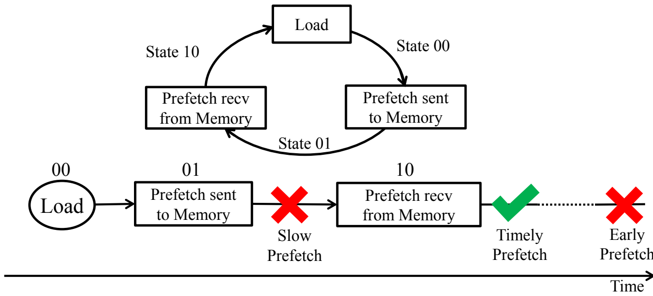

Fig. 7: (a) The three stages in the lifetime of a prefetch entry. (b) Timeline of a prefetch and related load requests. If next load happens between 01 and 10, the prefetching is slow. If it happens just after 10 it is timely, else it is too late. Once the next load happens the status is reset to 00

table for that load. This address is then put in the prefetch queue which sends the address to be prefetched to the global memory. Prefetch requests for all addresses that are in the same cache line are merged. With more than one warp prefetching, we must ensure that APOGEE is not confused by the addresses accessed by the loads of different warps. To this effect, to calculate the data to be prefetched for a warp accessing the cache, we further multiply the offset by an additional factor of $(n-N+1)$ to get the new offset. n is the number of threads in the SM. This will ensure that only that data is prefetched which will be used by the warp in the next access. The prefetcher is designed to ensure that requests to the same cache-line are coalesced.

During any access, if the offset of addresses do not match the offset stored in the table, APOGEE resets the confidence of that entry to 0. New offset calculations are done for the next access. Loads that do not have a fixed offset, always have a confidence of 0. If an entry needs to be evicted from the prefetch table, the entry with the lowest confidence is removed.

1) **Dynamic Distance Adjustment***:* To adjust the distance of prefetching, every entry in the prefetch table, has a 2-bit entry per warp to store 3 states: 00, 01 and 10. The lifetime of a prefetch entry is shown in Figure 7. Once a load of that entry happens, its state is set to 00. After the computation of its future address, if a prefetch request is sent, the state is changed to 01. Transition from state 01 to 10 occurs when the data comes back from memory to the cache. Whenever a new load of the same entry occurs, the state is reset to 00.

When an entry is in the 01 state, it indicates that the prefetch request has been sent to the memory, but the data has not come back. If the next load for that entry happens when the entry is in the 01 state, this means the prefetched data has not returned from memory and hence, prefetching is slow and The distance of prefetching for that entry is incremented and stored in the table. Future load instructions of that PC will use the updated distance of prefetching and prefetch farther ahead. This will reduce the slowness of prefetching. The same mechanism will continue to increase the distance till the distance is sufficiently ahead, such that when the next load instruction accesses the cache, the data is already prefetched.

Sometimes it may happen that the distance of prefetching is higher than required. In such cases, prefetched data will have to stay in the limited cache for a longer period of time.

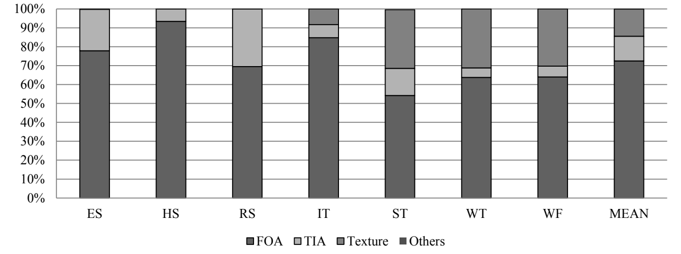

Fig. 8: Breakdown of access patterns of loads that missed in cache.

It may get evicted even before its use occurs. These cases of early prefetching are handled by APOGEE in the following way: When a prefetch request comes back from memory to the cache, the status of that entry is updated to 10. Whenever an entry is 10 and the next load instruction of that entry is a miss in the cache, we can assume that the data prefetched was either wrong or too early. APOGEE recomputes the address sent for the prefetching from the entry in the table and if the addresses match, then APOGEE is certain that the data was missed because it was prefetched too early. In such cases the distance of prefetching is decremented and stored back in the table. For future iterations, the new reduced distance is used for prefetching which will decrease the earliness of prefetching.

B. Thread Invariant Prefetching

Figure 8 shows the three major memory access patterns found in graphics applications. FOA accesses have alread been discussed in the earlier section. The remaining two access patterns are described below:

Thread Invariant Addresses (TIA): Apart from streaming data coming in to the various stages of the graphics pipeline, a significant number of accesses are to values that are used to set up the graphics pipeline before the innermost loop in a warp starts. These variables specify the various kinds of rendering, transformations, lighting, and related information. While they are at the same address for all the threads, their values change between invocations of the kernel.

Texture Accesses: Texturing operations are always handled by hardware texturing units as these units are designed for performance of specific texturing algorithms. Performing texture operations on generic shader cores is fundamentally different from what is done by the hardware texturing units and results in unreal memory access patterns. Since texturing operations will be handled by these hardware units, the accesses made for texturing in Figure 8 are not considered for this work.

TIA Timeliness. APOGEE dynamically detects and adapts to FOA and TIA access patterns at runtime. TIA accesses can be detected when the offset detected for an FOA is found to be zero. The data accessed by TIA accesses is 16% of the total data as shown in Figure 8 and to reduce multi-threading, misses to these addresses can become a performance bottleneck as per Amdahl's law. Therefore, APOGEE is extended to dynamically adapt to these kinds of accesses as well. We modify the prefetcher shown in Section III-A for TIA. For an entry in the prefetch table in Figure 6, if the offset is found to be zero for all the accesses in that warp, we assume that the address will be constant. The thread index of that entry is set to all ones to show that the access pattern of this PC is scalar.

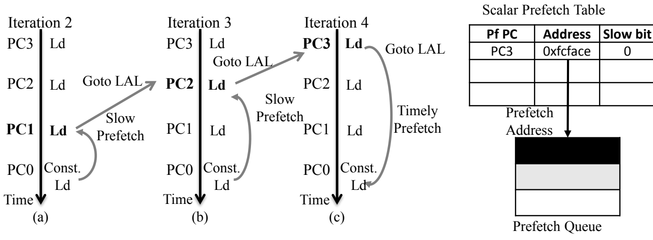

Fig. 9: PC0 always access same address. APOGEE keeps shifting the load from which the prefetch requests should be issued till it can prefetch early enough and stores that information in the Prefetch Table.

The main focus of TIA prefetching is not on predicting addresses, which are same anyways, but to issue the prefetch request in a timely fashion to bring evicted values back into the cache. Figure 9 shows an example to illustrate how the TIA prefetcher works. For a load accessing a thread invariant address (PC0) across all threads in a warp, the prefetcher tries to find a load instruction which is executed earlier than PC0 as shown in Figure 9(a). This earlier load is called a prefetching load instruction and behaves like a triggering mechanism. Whenever the prefetching load instruction accesses the cache, it sends a prefetch request for the constant load for which it has been designated as a prefetching load instruction.

TIA Implementation. To find the prefetching load instruction, the prefetcher keeps the PC of the load which accessed the cache most recently. This load is known as Last Accessed Load (LAL). For every load instruction, LAL is the most recently executed load before the current load instruction. When we see a miss to a load (PC0) and if the offset entry for that PC in the prefetch table is zero, we add an entry to the table shown in Figure 9. The LAL (PC1) is set to be the Load PC in the entry. The address in that entry is set to be the address (AD0) which was needed by PC0. In this way the prefetch request for PC0 is made before PC0 is executed as PC1 is the LAL for PC0. This table is part of the Prefetch Table and the fields of the entry are modified on the basis of the access type. Next time when PC1, which was the LAL for PC0, does a load request, an entry for it is checked in the prefetching table and the address AD0 is found. This address (AD0) is added to the prefetch queue and sent to global memory for prefetching. PC1 may not be sufficiently ahead of PC0 such that by the time the prefetching completes, PC0 might have already sent out a request to global memory as shown in Figure 9(b). In such a case the entry in the prefetch table is marked as slow by using the slow bit. When the next iteration is executed and the PC of the warp is at PC1, since the entry was marked as slow, we update the Load PC of that entry to the current LAL. This LAL will be the Load that was executed before PC1. In this way we prefetch 2 load instructions ahead of PC0. This is like traversing in a linked list. We continue this till a load has been reached which is sufficiently ahead such that prefetching at that load's execution results in a hit for the TIA load as shown in Figure 9(c).

C. Other Stalls

Apart from hiding the latency to memory the high degree of multi-threading in GPUs can hide many kinds of stalls, such as long latency floating point operations. If we completely remove multi-threading, then the stalls due to these operations cannot be hidden. Hiding the access to global memory is the majority of the work for the multi-threading. With prefetching we can remove this work and the remaining stalls that the multi-threading has to hide mostly include truly dependent floating point instruction. We observe that with scoreboarding, 4 warps are sufficient to hide the latency of other stalls such as datapath stalls.

IV. Experimental Evaluation

A. Simulation Setup:

The benchmarks used for Graphics and GPGPU applications are mentioned in Table Ia. We use viewsets from the SPEC-Viewperf version 9 benchmark suite which is a SPEC suite to evaluate graphics performance. To evaluate the OpenGL applications, we use Mesa3D version 7.11 as our driver, which is OpenGL version 2.1 compliant. We parallelized the main stages of the graphics pipeline using the Fractal APIs [18]. The MV5 reconfigurable simulator [18] understands these modifications in the source code and creates as many threads as mentioned in the Mesa source code whenever a kernel is launched. These threads are grouped into warps and are run in a SIMT fashion. We use the round-robin policy for scheduling of warps [2]. We modified the simulator to maintain a scoreboard and stall for true and output dependencies between instructions. We use five benchmarks for analysis of GPGPU applications.

We compare APOGEE with a previous prefetching technique for GPGPU applications, Many Thread Aware(MTA) [15]. In MTA prefetching, once the prefetcher is trained, it prefetches data of all the warps into a prefetch cache. Section VI explains MTA in further detail.

B. Performance and Power Evaluation:

The simulation parameters for the architecture are shown in Table Ib. The simulator models an in-order SIMT processor, scoreboarding, timing accurate cache and all resources such as MSHRs and uses LRU replacement policy. As the finer details of a GPU are not open knowledge, we calculate the power consumed by the GPU by calculating the dynamic and static power separately. We use the analytical power model of Hong et al [10] to calculate the dynamic power consumed by the various portions of the GPU. For our work, we focus only on the power consumed by one SM with and without our solution. We do not consider the power consumed by the global memory. The results show that on average only 2.2% extraneous memory requests are due to the prefetcher and, so, the difference due to the power consumption of these extra requests will not be significant.

For computation of leakage power one third of the SM total power is considered to be leakage power, based on prior studies [14], [9], [21]. We classify the leakage power of an SM into core and cache power. Leakage power of the cache is measured using CACTI [20]. The remaining leakage power is distributed amongst the various modules of the SM "core" on the basis of area. Areas of different modules are measured from EDA design tools and published figures. Table Ic shows component-wise source for the SM area used in the static power calculation. The most relevant leakage power

TABLE I: Experimental Parameters

Graphics	
HandShaded (HS)	5.8M vertices, Maya 6.5
Wolftextured (WT)	16.2M vertices texturing
WolfWireFrame (WF)	19.6M vertices wireframe
InsectTextured (IT)	5.5M vertices
SquidTextured (ST)	2.4M vertices with texturing
EngineShaded (ES)	1.2M vertices, state changes
RoomSolid (RS)	Lightscape Virtualization System
GPGPU	
FFT (FFT)	Fast Fourier Transform algorithm
Filter (FT)	bilinear filter detects edges
Hotspot (HP)	Evaluate temperature across grid
Mergesort(MS)	Parallel merge sort algorithm
Shortestpath (SP)	Dynamic programming algorithm

(a) Benchmark Description. (M = million)

Parameter	Value
Frequency	1 GHz
SIMD Width	8
Warp Size	32 threads
Number of Warps	1, 2, 4, 8, 16, 32
I-Cache	16kB, 4 way, 32B per line
D-Cache	64kB, 8 way, 32B per line
D-Cache Access Latency	4 cycles
Memory latency	400 cycle
Integer Instruction	1 cycle
FP Instruction	24 cycles [24]
Scoreboarding	True
Prefetch Table	64 Entry, 76 bits
Prefetch Issue Latency	10 cycles
Memory Bandwidth	12 GBps/SM

(b) Simulation Parameters

Technology	**TSMC 65 nm**		
Component	**Source for Area**	**GPU**	**APOGEE**
Fetch/Decode	McPAT, RISC in-order model	Default	Default
FPU	Published Numbers [8]	8	8
SFU	4 FPUs [17]	2	2
Register File	Artisan RF Compiler	16k, 32-bit	2k, 32-bit
Shared Memory	CACTI	64kB	64kB
Prefetcher	Artisan SRAM Compiler	None	64, 73-bit entries

(c) Details of Area Calculation, and component wise description for the major components of baseline GPU and APOGEE

Prefetcher Entry	
PC	10 bits
Address	28 bits
Offset	8 bits
Confidence	8 bits
Thread Idx	6 bits
Distance	6 bits
PF Type	2 bits
PF State	8 bits

(d) Prefetcher Entry Area

number for this work, that of the register file, were obtained through the ARM Artisan Register File Compiler tool. All the measurements are at 65 nm using TSMC technology.

The prefetcher is considered to be an SRAM structure integrated with the cache controller. The dynamic and leakage power of this structure is calculated from the Artisan SRAM Compiler. The data from the global memory is brought directly into the cache and no stream buffers are used. Table Id shows the breakup of the prefetcher area. For all the prefetching types there is only one table. The interpretation of the entries of the table vary depending on the Prefetch Type bit. FOA prefetching needs the most amount of information. TIA prefetching needs less bits. Hence, Table Id shows the number of bits required by one entry of FOA prefetching.

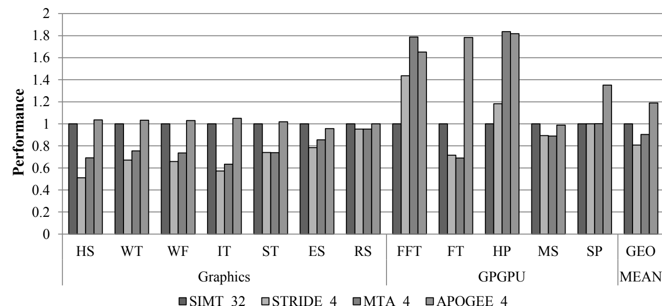

Fig. 10: Comparison of overall speedup of SIMT with 32 warps, stride prefetcher per warp with 4 warps, MTA with 4 warps and APOGEE.

V. RESULTS

A. Performance and Power

Performance of Prefetching with Low multi-threading. Figure 10 shows the speedup achieved by stride prefetching per warp, MTA (MTA_4) and APOGEE_4, all with four warps when compared to a baseline configuration of SIMT with 32 warps (shown as SIMT_32, and as always having a value of '1'). In graphics benchmarks, APOGEE_4 is similar in performance to SIMT with 32 warps. The stride prefetching and MTA techniques are designed for use in GPGPU applications and, therefore, do not provide performance benefits for graphics applications. APOGEE_4, however, performs better than the baseline and the two compared techniques as it has the ability to prefetch in a timely manner and prefetch for TIA accesses. APOGEE_4 is at par with, or better than, massively-multithreaded SIMT for six out of the seven benchmarks, providing a 1.5% speedup over the baseline 32-warp design.

For GPGPU benchmarks, stride prefetching per warp and MTA perform considerably better. However, APOGEE_4 performs significantly better than these techniques in FT, MS and SP. For FFT and HP benchmarks, it is within a few percent of other techniques. The performance of APOGEE_4 varies from a 4% loss to 82% better performance than the baseline. Overall Stride prefetching and MTA with 4 warps have 19% and 10% performance degradation as compared to baseline, respectively, whereas, APOGEE_4 with 4 warps performs 19% better than SIMT with 32 warps across all benchmarks.

Comparison with MTA_32. The performance of MTA with 32 warps(MTA_32) and APOGEE_4 with four warps is shown in Figure 11. The baseline configuration is SIMT with 32 warps. On average the performance of MTA varies from 1% degradataion in performance to 79% improvement

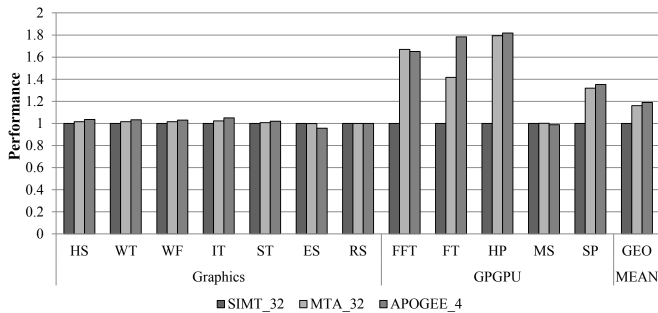

Fig. 11: Comparison of overall speedup of SIMT with 32 warps, MTA with 32 warps and APOGEE.

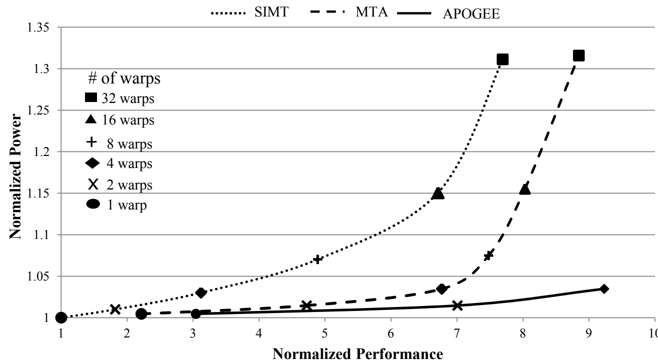

Fig. 12: Comparison of Performance/Watt of high-degree of multi-threading and multi-threading with prefetching

over baseline with an average speedup of 16% over SIMT with 32 warps. Overall, MTA_4 warps provides only 90% of the performance of SIMT_32 and MTA_32 provides a 16% performance improvement. To achieve the 26% performance difference between MTA_4 and MTA_32, MTA requires an eight fold increase in the number of concurrent warps from 4 to 32 whereasAPOGEE_4 is 3% faster than MTA_32 across all the benchmarks. MTA_32 can hide the latency of accesses that it cannot prefetch for, with the 32 warps that it has at its disposal. But with the TIA prefetcher and timely prefetching, APOGEE_4 can prefetch the data and does not need the additional warps.

Efficiency. Figure 12 shows the power consumption on the y-axis and performance on the x-axis for SIMT, MTA and APOGEE as the number of warps are varied. The baseline for power and performance is a configuration with one warp and no prefetching. As more warps are added to SIMT, the performance increases as long latency stalls are hidden by multi-threading. Initially with 2, 4, and 8 warps, some stalls are hidden by the additional multi-threading and the performance gain is almost linear. Due to the addition of multi-threading hardware, the power also increases. However, the change in power for the corresponding change in performance is much higher for SIMT when it goes from 16 warps to 32 warps. For the maximum performance in SIMT, the power overhead is 31.5% over baseline. MTA provides higher performance compared to SIMT for 2, 4 and 8 warps. At 8 warps it provides 52% speedup over SIMT at similar levels of power consumption. But to get the maximum performance of MTA which is 18% more than its performance with 8 warps, it needs 32 warps which consumes 28% higher power as compared to 8 warps. The addition of 24 warps provide diminishing returns over the first 8 warps.

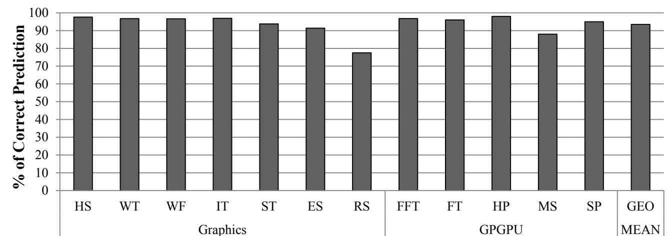

Fig. 13: Accuracy of the overall prefetching technique.

APOGEE_4 provides performance benefits with 4 warps, even over 32-warp SIMT and MTA, while having the power consumption of 4-warp SIMT and MTA. Due to the high performance of the dynamic prefetching techniques, the performance-per-watt of APOGEE_4 in the best-performance case is 51% higher than that of SIMT and 33% higher than that of MTA. The performance-per-watt efficiency is much higher for APOGEE_4 as fewer warps and a relatively small prefetch table provide 19% speedup and having 8 times less multi-threading hardware provides significant power benefits. In this work, moving from 32 warps to 4 warps results in the removal of 14K 32-bit registers. As shown in Figure 2, these hardware structures are underutilized and leak a significant amount of power. The overall improvement in power-efficiency by prefetching, therefore, is due to both performance increases and reduced power consumption.

B. Analysis of Performance:

This section analyzes the reasons for the difference in performance of APOGEE as compared to MTA and SIMT. We do not analyze the same for stride prefetching on a per warp basis due its poor performance as shown in Figure 10.

Accuracy. The two major criterion for successful prefetching are correctly predicting addresses and their timely prefetching. Figure 13 shows the accuracy of the overall prefetching technique for all the benchmarks. On average, 93.5% of the addresses that were predicted by the prefetcher were correct and accessed by the core in the future. The high correctness of the prefetcher is due to the regular access patterns of the programs. Sometimes, due to certain conditional execution of instructions leading to a change in memory access patterns, the prefetcher may reset itself for a particular entry. This leads to low correctness in RS and MS benchmark. The overall access time to global memory for all loads for APOGEE is reduced to 43 cycles from 400 cycles which does not need high degree of multi-threading.

Miss Rate. We show the effectiveness of prefetching by looking at the variation in the data cache miss rate. A reduction in the data cache miss rate indicates that there are fewer requests that have to be brought in from L2 and memory. While prefetching can help in reducing the dcache miss rate, if the prefetching predicts wrong addresses or brings in data that evicts useful data, the overall miss rate will increase. So, it is a good metric to determine the effectiveness of prefetching. Figure 14 shows the reduction in dcache miss rate when compared to 32 warp SIMT with no prefetching. MTA_4 is able to reduce the dcache miss rate only for 4

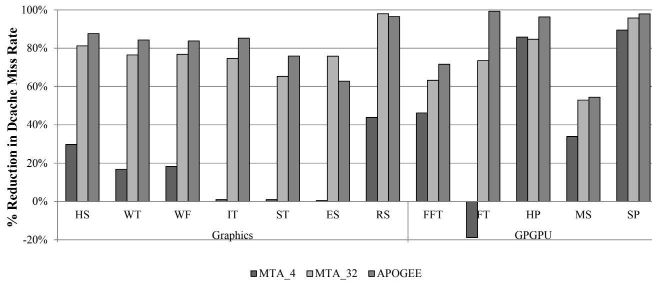

Fig. 14: Variation in Dcache Miss Rate.

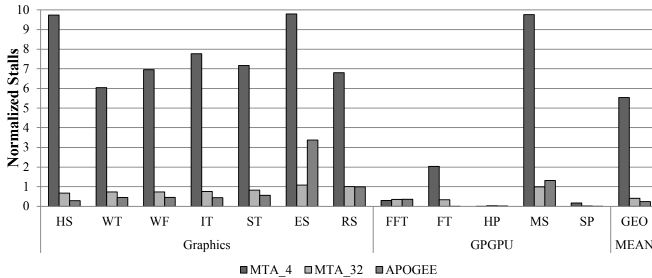

Fig. 15: Variation in stall cycles of the SMs, normalized to SIMT with 32 warps.

out of the 7 graphics benchmarks. It does better for GPGPU benchmarks except for FT where it actually increases the miss rate. MTA_32 and APOGEE perform considerably better than SIMT with 32 warps. They are able to reduce the baseline dcache miss rate by close to 80% in 8 benchmarks. Due to the effectiveness in prefetching, access to the main memory needs to be hidden only for the remaining 20% of the times. A low reduction in miss-rate actually transfers to low performance for ES and MS for both MTA and APOGEE as shown in Figure 11.

Reduction in Stalls. The reduction in stall cycles of the pipeline of the core is shown in Figure 15. The y-axis shows the reduction in stall time normalized to SIMT with 32 warps. Due to the reduction in the dcache miss rate in MTA_32 and APOGEE, there is a significant reduction in stalls. For MTA_4, since the low reduction in dcache rate, the number of stall cycles of the shader pipeline increases significantly due to the lack of multi-threading. MTA_32's reduction in cycles is achieved through prefetching as well as high degree of multi-threading. In APOGEE there is only a fraction of multi-threading present and, yet, due to the high degree of correct prefetching, as shown in Figure 13, and higher reduction in dcache miss rate, as shown in Figure 14, it is able to eliminate more stalls.

Bandwidth. Data that is wrongly or untimely prefetched will not be used by the core and unnecessarily strains the bandwidth of the system. Figure 16 shows that on average around 2.2% extra requests are sent to the global memory by APOGEE. These extra requests include prefetch and regular read requests. Graphics benchmarks have low bandwidth wastage due to prefetching as compared to the GPGPU benchmarks. Even though the accuracy of the prefetcher is around 92%, the extraneous data overhead is 2.2%. There are times when the data accessed by different threads overlap and, hence, prefetcher may predict the address that is already in the cache

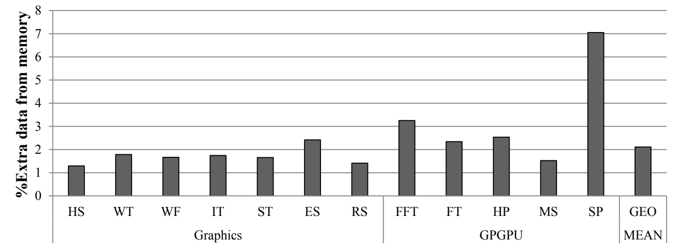

Fig. 16: Extra data transferred from memory to the cache including prefetching requests.

In such cases, no request is sent to the memory, reducing the prefetcher overhead. Overall, due to the high accuracy and timeliness of the prefetcher, prefetching has a very small impact on an application's bandwidth consumption.

VI. Related Work

Prefetching on GPUs has been studied in the past. Lee et al. [15] proposed MTA, Many Thread Aware prefetching for GPGPU applications. This work compares MTA with APOGEE in detail in Section V. Their solution primarily addresses accuracy of prefetching with a complex throttle for wasteful prefetching which involves finding data that was prefetched and evicted before being utilized. Such counters can be complex to implement. APOGEE focuses on timely prefetching by adjusting the distance of prefetching using two bits in a table. Furthermore, MTA addresses only one kind of access pattern for graphics application. Arnau et al. [1] shows the inefficiency of using MTA for Graphics applications. In this work we show the effectiveness of our prefetching for two major access pattern to overcome the deficiency of GPGPU prefetching techniques for graphics applications. In MTA prefetching the prediction of the next warp's data is based on the currently trained prefetcher. Once the training is complete, the data for all the warps is prefetched and stored in the prefetch cache. This approach is not very cordial for energy efficiency as show in Figure 12.

Arnau et al. [1] show the ineffectiveness of standard prefetching techniques for graphics applications and use a decoupled access/execute mechanism for graphics processing. Apart from dynamically adapting to the access patterns found in these applications, APOGEE also maintains the timeliness of prefetching. So, our technique is more effective than the prefetching techniques compared in [1]. They change the graphics core architecture substantially, whereas our work has less invasive changes to the core itself. We change the cache-controller and provide efficiency through prefetching. While their technique is effective for complex access pattern, their system is analyzed with 100 cycle memory access latency, we show that our technique is effective for 400 cycle latency for a graphics memory access. Prefetching for texture caches was studied by Igehy et al. [11]. They take advantage of distinct memory access pattern of mip-mapping. In APOGEE only operations done at the shader core are considered and texturing is considered to be handled by separate hardware structures.

Energy efficiency on GPUs has been addressed from various directions. Gebhart et al. [9] provide a register file cache to access a smaller structure rather than the big register files. They have a two-level scheduler to select from a small set of active

threads to reduce the amount of register caching required. Lin et al. [16] propose software prefetching for GPU Programs to optimize power. Their work adds prefetch instructions and vary voltage to address high power consumption. Dasika et al. [4] create a fused FPU design to reduce the number of register file accesses in a SIMT-like architecture. Yu et al. [13] increase the efficiency of the register files for GPU with a SRAM-DRAM memory design. Zhao et al. [25] propose an energy efficient mechanism to reduce the memory power consumption for GPUs via a reconfigurable in-package wide interface graphics memory on a silicon interposer. Tarjan et al. [23] proposed "diverge on miss" and Meng et al. [18] proposed Dynamic Warp Subdivision where performance improves due to the advantage of prefetching effects by early execution of some threads in a warp. All these approaches are orthogonal to the approach used by APOGEE for energy efficiency.

Adaptive prefetching has been studied for CPU architectures. Srinath et al. [22] use a feedback mechanism to adjust the prefetch accuracy, timeliness and cache pollution. Ebrahimi et al. [6] coordinate multiple prefetcher for a multicore architecture to address prefetcher caused inter-core interference. APOGEE focuses on adapting to different access patterns for throughput oriented applications in GPUs.

VII. Conclusion

GPUs use multi-threading to hide memory access latency. The associated cost of thread contexts and register state leads to an increase in power consumption and a reduction in performance-per-watt efficiency. In this work, we demonstrate that dynamically adaptive prefetching technique is a more power-efficient mechanism to reduce memory access latency in GPUs with comparable performance. Even though conventional wisdom suggests that prefetching is more energy hungry, when used in the context of GPUs to reduce the high-degree of multi-threading and related hardware, it can be used to reduce the overall power consumption.Through analysis of the memory access patterns of GPU applications, we found that fixed-offset address access and thread invariant access can be prefetched for GPU applications. We demonstrate that a dynamically adaptive prefetcher can correctly predict these addresses in 93.5% of the cases. Furthermore, adaptive prefetching with dynamic distance adjustment and multi-threading of 4 warps per SM, reduce the requests that go to memory by 80%. Overall, APOGEE prefetching with 4 warps of multi-threading can provide a performance increase of 19% over a GPU which has 32 warps per SM leading to a 52% increase in performance per watt, with 8 times reduction in multi-threading support.

Acknowledgments

We thank the anonymous referees who provided excellent feedback. We also thank Gaurav Chadha for providing feedback on this work. This research was supported by National Science Foundation grant CNS-0964478, SHF-1217917 and ARM Ltd.

References

[1] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile GPU performance with a decoupled access/execute fragment processor," in *Proc. of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 84–93.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 163–174.

[3] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[4] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, "PEPSC: A power-efficient processor for scientific computing," in *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 101–110.

[5] P. Diaz and M. Cintra, "Stream chaining: exploiting multiple levels of correlation in data prefetching," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 81–92.

[6] E. Ebrahimi, O. Mutlu, J. L. Chang, and Y. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 316–326.

[7] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 102–110.

[8] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 913 –922, july 2011.

[9] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 235–246.

[10] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. Proc. of the 37th Annual International Symposium on Computer Architecture, pp. 280–289.

[11] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 1998, pp. 133–ff.

[12] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[13] W. kei S.Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Kan, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 247–258.

[14] P. Kogge, *et al.*, *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. Technical Report TR-2008-13, University of Notre Dame, 2008.

[15] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *Proc. of the 43rd Annual International Symposium on Microarchitecture*, 2010, pp. 213–224.

[16] Y. Lin, T. Tang, and G. Wang, "Power optimization for GPU programs based on software prefetching," in *Trust, Security and Privacy in Computing and Communications*, 2011, pp. 1339–1346.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, march 2008.

[18] J. Meng, "The MV5 simulator: An event-driven, cycle-accurate simulator for heterogeneous manycore architectures," 2010, https://sites.google.com/site/mv5sim/.

[19] NVIDIA, "Fermi: NVIDIAs next generation CUDA compute architecture," 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architect - ure_Whitepaper.pdf.

[20] G. Reinman and N. P. Jouppi, "CACTI 2.0: An integrated cache timing and power model," Hewlett-Packard Laboratories, Tech. Rep. WRL-2000-7, Feb. 2000.

[21] G. Sery., S. Borkar, and V. De, "Life is CMOS: why chase the life after?" in *Proc. of the 39th Design Automation Conference*, 2002, pp. 78 – 83.

[22] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007, pp. 63–74.

[23] D. Tarjan, J. Meng, and K. Skadron, "Increasing memory miss tolerance for SIMD cores," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 22:1–22:11.

[24] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, 2010, pp. 235–246.

[25] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Energy-efficient gpu design with reconfigurable in-package graphics memory," in *Proc. of the 2012 International Symposium on Low Power Electronics and Design*, 2012, pp. 403–408.