Snake: A Variable-length Chain-based Prefetching for GPUs

Saba Mostofi
Sharif University of Technology
Department of Computer Engineering
Iran
saba.mostofi98@gmail.com

Hajar Falahati
Institute for Research in Fundamental Sciences (IPM)
School of Computer Science
Iran
hfalahati@ipm.ir

Negin Mahani*
Shahid Bahonar University
Department of Computer Engineering
Iran
negin.mahani@uk.ac.ir

Pejman Lotfi-Kamran
Institute for Research in Fundamental Sciences (IPM)
School of Computer Science
Iran
plotfi@ipm.ir

Hamid Sarbazi-Azad†
Sharif University of Technolgy
Department of Computer Engineering
Iran
azad@sharif.edu

ABSTRACT

Graphics Processing Units (GPUs) utilize memory hierarchy and Thread-Level Parallelism (TLP) to tolerate off-chip memory latency, which is a significant bottleneck for memory-bound applications. However, parallel threads generate a large number of memory requests, which increases the average memory latency and degrades cache performance due to high contention. Prefetching is an effective technique to reduce memory access latency, and prior research shows the positive impact of stride-based prefetching on GPU performance. However, existing prefetching methods only rely on fixed strides. To address this limitation, this paper proposes a new prefetching technique, Snake, which is built upon chains of variable strides, using throttling and memory decoupling strategies. Snake achieves 80% coverage and 75% accuracy in prefetching demand memory requests, resulting in a 17% improvement in total GPU performance and energy consumption for memory-bound General-Purpose Graphics Processing Unit (GPGPU) applications.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

GPU, On-Chip Memory, Prefetching, Performance.

*Also with Institute for Research in Fundamental Sciences (IPM), School of Computer Science.
†Also with Institute for Research in Fundamental Sciences (IPM), School of Computer Science.

ACM Reference Format:
Saba Mostofi, Hajar Falahati, Negin Mahani, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2023. Snake: A Variable-length Chain-based Prefetching for GPUs. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, Toronto, TO, Canada, 14 pages. https://doi.org/10.1145/3613424.3623782

1 INTRODUCTION

Modern General-Purpose Graphics Processing Units (GPGPUs), simply called GPUs in the rest of this paper, are extensively used in many essential memory-bound applications (i.e., applications whose overall performance is restricted due to memory transactions), including applications ranging from machine learning to scientific computing [14, 18, 35]. To service a bunch of memory requests and capture the on-chip memory locality of the memory-bound GPGPU applications, a CPU-like hierarchical memory model (i.e., L1 data cache, L2 cache, and off-chip DRAM) is exploited in (NVIDIA) GPUs [20]. The L1 data cache is dedicated to a Streaming Multiprocessor (SM) and supports fast local data access. The L2 cache is shared by all SMs to bypass off-chip DRAM latency [36]. However, the average memory latency continues to be a crucial bottleneck for memory-bound GPGPU applications [9, 24, 26].

To tolerate memory latency, thousands of parallel threads are executed concurrently on the GPUs, i.e., Thread-Level Parallelism (TLP). However, the large number of memory requests generated by running parallel threads increases average memory latency and degrades cache performance due to high contention [47, 49]. Therefore, GPUs spend a large number of cycles when all available warps (a group of 32 threads) are waiting for data from memory (a.k.a., memory wall) [3, 46]. The memory wall results in crucial challenges in GPU, such as resource under-utilization, overall performance degradation, and increased overall power consumption [40].

To tackle the memory wall problem, prefetching is one of the well-studied mechanisms [15, 25, 27, 29, 37]. Prior research shows that stride-based prefetching [25, 27, 29, 37] reduces the average memory latency and improves GPU performance, due to two main facts: (1) there is a fixed stride among memory accesses in GPUs, and (2) a large number of concurrent threads can use prefetched data in

GPUs [17, 29]. Well-acknowledged stride-based prefetching mechanisms in GPUs are Intra-warp [27, 29], Inter-warp [29, 37], and Inter-CTA (Cooperative Thread Array or Thread block) [25] mechanisms. Intra-warp prefetcher enables each thread to prefetch for the next iteration of the same load instruction executed by the same thread in the same warp [27, 29]. Inter-warp prefetcher enables each thread to prefetch for future warps [29]. Inter-CTA prefetcher enables each thread to prefetch for future CTAs [25]. Some work prefetches a sequence of addresses based on fixed chunks of memory spaces[15]. All these mechanisms limit their potential gain (i.e., the number of correct prefetched data) due to the fact that they focus on a fixed stride between accesses of a load instruction or fixed chunks [27, 37], see Section 2 for more details.

In this paper, by analyzing various memory access patterns of memory-bound GPGPU applications, we observe that there is a list of variable strides among the consecutive load instructions (referred to as chains of strides in this paper) at different levels, e.g., inter-warp and intra-warp. Note that these chains are composed of various lengths of various strides, instead of a fixed stride for each load instruction used in prior work. Prefetching techniques that utilize chains of strides are well-studied in CPUs [23, 39, 43]. However, directly applying them to GPUs creates new challenges, including identifying patterns across different levels (threads, warps, and CTAs), managing out-of-order warp scheduling, architectural constraints (memory and bandwidth limitations), and handling divergent memory access patterns.

As an intuition, we compare an Ideal prefetcher that supports all possible (fixed/variable) strides under the optimal characteristics (i.e., infinite storage and zero latency for the prefetching requests) with two state-of-the-are prefetcher mechanisms: CTA-aware [25] that prefetches data for next CTAs, and Many–Thread–Aware (MTA) prefetcher that combines both Inter-warp and Intra-warp mechanisms and provides the best coverage among the prior work [29]. We observe that the Ideal prefetcher outperforms CTA-aware and MTA in terms of coverage by 70% and 25%, respectively, for memory-bound GPGPU applications. This result clearly shows that GPU-specific chain-based prefetching has the potential to improve GPU performance and energy efficiency, where multiple future warps utilize patterns detected in a single warp.

To exploit the benefits behind the chains of strides, we propose Snake, an efficient inter-thread prefetching mechanism based on chains of strides working on various levels (i.e., Intra-warp and Inter-warp). Figure 1 provides a clearer illustration of three types of strides (i.e., Intra-warp, Inter-warp, and Inter-thread).

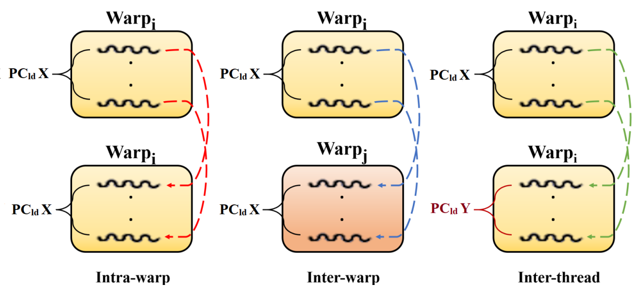


Figure 1: Three types of strides.

Snake (1) detects the chains of strides between consecutive program counter (PC) of load instructions (called PC$_{ld}$ in this paper) by using two tables, (2) issues prefetch requests based on the sufficiently trained strides, leading to high coverage (around 80%), and (3) exploits store decoupling and prefetch throttling mechanisms to alleviate cache contention. The store decoupling mechanism keeps prefetched data and L1 data separately. The prefetch throttling mechanism stops the prefetching, triggered by the amount of unused memory or bandwidth. As a result, Snake prefetches 75% of memory requests accurately, which is significantly higher (around 55%) than the most accurate state-of-the-art mechanism (CTA-aware) [25]. It is worth noting that the proposed method is primarily designed to facilitate the execution of a single application. However, it can be extended to support multiple applications where the chains of strides are detected within each application.

This paper makes the following contributions:

- We demonstrate the suboptimality of state-of-the-art prefetching mechanisms in providing accurate prefetched data for memory-bound GPGPU applications.
- We propose Snake, an efficient inter-thread prefetching mechanism based on chains of strides working on various levels (i.e., Intra-warp and Inter-warp).
- We demonstrate that Snake improves memory-bound GPGPU application performance by an average of 17% while reducing energy consumption by up to 17%.
- We show that Snake has significantly better coverage and accuracy (up to 60% and 55% respectively) as compared to the state-of-the-art prefetching mechanism (CTA-aware [25]).
- We show that Snake's overheads are less than 1% on Volta V100 GPU power/die area.

2 BACKGROUND AND MOTIVATION

The GPU architecture consists of several SMs, as shown in Figure 2. Each SM contains execution cores called Streaming Processors (SPs) that execute identical instructions in groups of 32 threads (i.e., warps) concurrently. Multiple warps together form a thread block or a cooperative thread array (CTA). CTAs are assigned to SMs based on their on-chip resources (e.g., register file and shared memory) [11, 36, 42]. The SMs start to execute the warps of the assigned CTAs. Whenever a warp's thread is waiting for data from memory, the warp stalls until its memory request is satisfied and the scheduler switches to execute another ready warp [35]. Each SM has a private memory subsystem (register file, L1 data cache, shared memory, constant cache, and texture cache). If accesses miss in the on-chip caches, they are sent to a shared L2 cache via an interconnection network at the expense of higher latency (around 100 cycles). The L2 cache banks are connected to memory controllers that govern the global memory. Fetching data from global memory takes hundreds to thousands of cycles, given the traffic [34].

GPUs cannot efficiently use the memory system due to their inherent high TLP. A large number of memory requests generated by concurrent threads are sent to the L1 data cache simultaneously. Limited on-chip resources and a large number of accesses lead to two main challenges. First, the L1 data cache contention increases; Threads can interfere with each other and evict useful data being

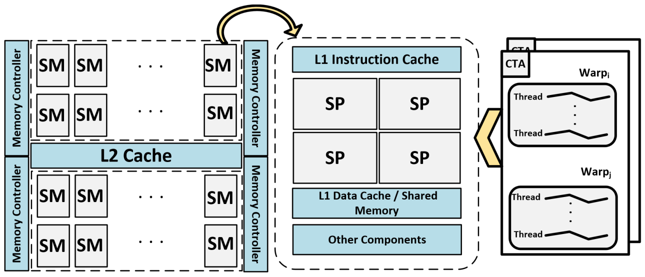


Figure 2: Baseline GPU architecture.

used by other threads. Second, after a while, there is not enough resource to handle the new accesses generated by the latter threads (i.e., reservation fail). Therefore, a high TLP degree compromises memory system efficiency and causes warps to stall for long periods in memory-bound GPGPU applications [4].

To analyze the aforementioned challenges and provide motivational results, we use Accel-Sim v1.2.0 [21] and execute various memory-bound GPGPU applications from Rodinia [31], Parboil [44], and ISPASS [5] benchmark suites. For more information on the simulation configuration and benchmarks, see Section 4. To study the intensity of the memory wall, we examine the number of reservation fails,[1] bandwidth utilization, and memory stalls as key factors affecting average memory latency and memory throughput in memory-bound GPGPU applications.

To study the quantity of the reservation fails, we measure the number of reservation fails normalized to the total L1 data cache accesses (i.e., hit, miss, reserved, and reservation fail) in memory-bound GPGPU applications in Figure 3. We observe that reservation fails occur, on average, 30%. A reservation fail request arises due to factors like insufficient free spaces in miss queues, reserving all entries in the L1 data cache that are waiting for data from lower memory levels, and insufficient free spaces in MSHR. Prior work shows that reservation fails in GPUs are primarily due to miss queue congestion in recent generation [19, 21, 22], which confirms our motivational analysis. These requests continue to be sent to the L1 data cache until they are accepted. Therefore, more reservation fails waste more execution cycles, harming the system performance, power consumption, and resource utilization [12, 13, 41].

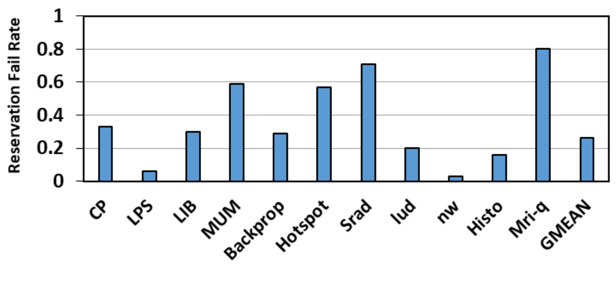


Figure 3: The percentage of reservation fails.

To study the rate of bandwidth utilization, we measure the amount of transferred data between the L1 data cache and the L2 cache, normalized to the theoretical bandwidth of the baseline GPU, across various memory-bound GPGPU applications in Figure 4. We observe that around 33% of the total bandwidth (between the L1 data cache and the L2 cache) is utilized. Therefore, on-chip resources are not used efficiently in memory-bound GPGPU applications due to severe cache contention (resulting in a large number of reservation fails and bandwidth under-utilization) [10].

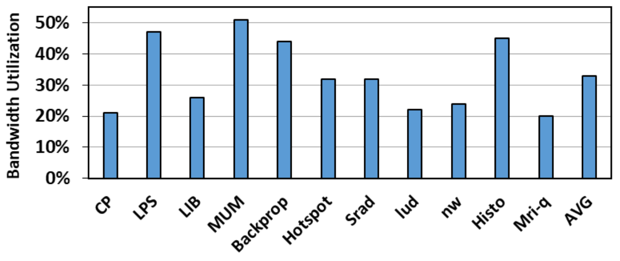


Figure 4: Interconnect bandwidth utilization.

To further analyze the impact of the memory wall on overall performance, we examine the number of cycles when all available warps are waiting for memory data in memory-bound GPGPU applications. Figure 5 shows the number of stalls due to the memory latency. We observe that around 55% of the run-time stalls are caused by memory stalls.

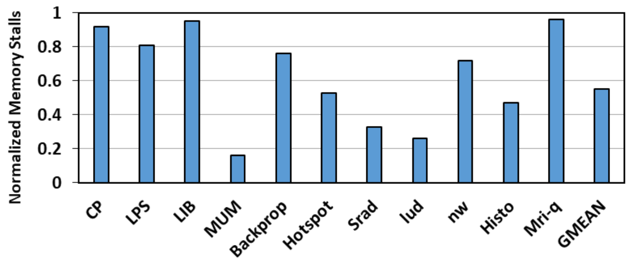


Figure 5: The number of cycles when all available warps are waiting for data from memory normalized to the total number of stalls in memory-bound GPGPU applications.

To address the memory wall problem, data prefetching (i.e., fetching data from a slower original memory to a faster local memory before it is actually needed [33]) is one of the promising techniques in GPUs [45]. Considering the spatial locality, one approach is to prefetch a sequence of addresses based on fixed chunks of memory spaces [15]. However, this method can impose excessive pressure on the memory system as it brings in a large amount of data that may not even be used, leading to cache under-utilization. Prior research shows that stride-based prefetching reduces the average memory latency and improves GPU performance, due to two main facts. First, the load addresses of each thread are typically defined through its assigned built-in variables (e.g., thread index and block index). In other words, the data index is determined based on the thread index, and hence, it is possible to have a fixed stride between addresses requested by different threads. Second, a large number of concurrent threads in GPUs use the accurate prefetched data which reduces the pressure on the resources, reservation fails,

[1]Each time the L1 data cache is accessed, the existence of data is verified and the status (hit, miss, reserved, and reservation fail) is determined.

execution cycles, and power consumption [29, 37]. However, inaccurate prefetching evicts useful data which increases memory traffic, wastes bandwidth resources, and ultimately decreases the overall performance [25].

Prior research introduces various stride-based prefetching mechanisms (e.g., Intra-warp [29], Inter-warp [29], and Inter-CTA [25]) tailored to GPU characteristics. Intra-warp prefetcher enables each thread to prefetch for the next iteration of the same load instruction executed by the same thread in the same warp [29]. This method achieves high coverage and accuracy in the existence of deep loop iterations. However, to exploit concurrent execution, memory-bound GPGPU applications replace deep loops with parallel programming [25, 29]. Inter-warp prefetcher enables each thread to prefetch for future warps [29]. As there is a fixed number of threads within a warp, the stride between data accessed from different warps can be constant. However, this method suffers from timeliness and accuracy trade-offs. Warps within a CTA often have the same stride, but they schedule closely in time so that the memory access latency cannot be hidden. Many–Thread–Aware (MTA) prefetcher, combines both Inter-warp and Intra-warp mechanisms together to exploit both prefetching opportunities. However, MTA suffers from Intra/Inter-warp drawbacks (e.g, lack of opportunity for accurate prefetching) and does not solve the timeliness problem. Inter-CTA prefetcher, e.g., the CTA-aware [25], enables each thread to prefetch for future CTAs and provides a higher timeliness opportunity. All these mechanisms limit their potential gain (i.e., the number of accurate prefetched data) due to the fact that they are trained to find a fixed stride between accesses of a load instruction [25, 29].

To measure the effectiveness of the prior solutions, we compare the coverage of Intra-warp [29], Inter-warp [29], MTA [29], and CTA-aware [25] prefetchers to the Ideal prefetcher. Ideal prefetcher supports all possible (fixed/variable) strides with infinite storage and zero latency for the prefetching requests. Figure 6 shows the coverage of Intra-warp [29], Inter-warp [29], MTA [29], and CTA-aware [25] prefetchers compared to the Ideal prefetcher. We make a key observation that the coverage of the Ideal prefetcher is around 25% and 70% higher than the state-of-the-art mechanisms (i.e., MTA [29] and CTA-aware [25]), showing the gap between the ideal case and the state-of-the-art mechanisms.

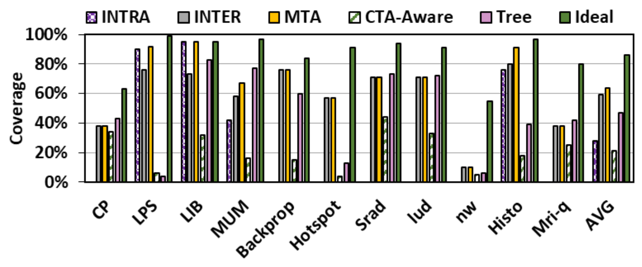


Figure 6: Coverage of Intra-warp [29], Inter-warp [29], MTA [29], CTA-aware [25], and Ideal prefetchers.

To better illustrate a comprehensive example of various strides in the GPU programming model, we analyze LPS [5] application that includes four types of different strides (Intra-warp [29], Inter-warp [29], Inter-CTA [25] fixed strides, and Inter-thread chains of strides). The presented code in Figure 7 demonstrates these four types of strides in LPS. We make four key observations. First, this code has opportunities for Intra-warp prefetching because the value of ind is determined by fixed variables, i.e., $\text{thread}_{\text{id}}$ and block_{id} (line 6). The loop iteration, which reads from an array at a fixed stride, is covered by Intra-warp prefetching (line 10). Second, warps within a CTA have a fixed stride which is covered by Inter-warp prefetching. Considering the same block_{id}, and a definite number of threads within each warp, a fixed stride is observed between the corresponding threads of each warp in a CTA. Third, to ensure accurate and timely prefetching, Inter-CTA prefetching considers the base addresses of each CTA and enables a warp in the current CTA to prefetch for warps in other CTAs (lines 2 and 3). Fourth, we observe Inter-thread chains of strides that have never been considered in prior work [25, 29] (e.g., in lines 12 and 13). Whenever a thread accesses the value of u1[ind], the next instruction accesses the value of u1[ind+KOFF]; as the value of KOFF is predictable, we can prefetch the next load instruction for each thread.

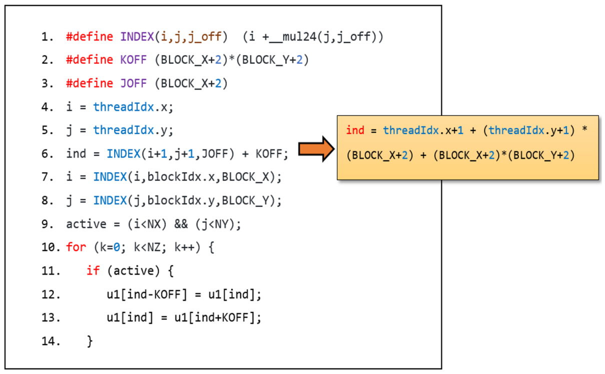

```
1.  #define INDEX(i,j,j_off)  (i +__mul24(j,j_off))
2.  #define KOFF (BLOCK_X+2)*(BLOCK_Y+2)
3.  #define JOFF (BLOCK_X+2)
4.  i = threadIdx.x;
5.  j = threadIdx.y;
6.  ind = INDEX(i+1,j+1,JOFF) + KOFF;
7.  i = INDEX(i,blockIdx.x,BLOCK_X);
8.  j = INDEX(j,blockIdx.y,BLOCK_Y);
9.  active = (i<NX) && (j<NY);
10. for (k=0; k<NZ; k++) {
11.    if (active) {
12.       u1[ind-KOFF] = u1[ind];
13.       u1[ind] = u1[ind+KOFF];
14.    }
```
```
ind = threadIdx.x+1 + (threadIdx.y+1) *
(BLOCK_X+2) + (BLOCK_X+2)*(BLOCK_Y+2)
```

Figure 7: Load address calculation and various strides in the source code of LPS [5] application.

Figure 8 illustrates an example of the variable chains of strides discovered between a group of PC_{ld}s executed by all warps in the LPS code traces. For instance, we identified a chain of inter-thread strides (-400, 40400, -400) between four PC_{ld}s (PC_1-PC_4). Notably, two of the PC_{ld}s (PC_1 and PC_3) exhibit fixed inter-warp strides of -400 and 400, respectively. Furthermore, since PC_3 and PC_4 occur in a loop, an intra-warp stride (40000) can be calculated for them. Considering the above observations, we conclude that (1) using intra-warp stride opportunities is unlikely, (2) inter-warp stride opportunities may change due to variations in CTAs, and (3) various chains of strides can be identified among a group of warps due to certain statements, such as branch.

To further study the prefetching opportunity based on the chains of strides, we perform a trace-based analysis on the memory accesses of memory-bound GPGPU applications. To this end, we first show the number of PC_{ld}s in the chains, normalized to the total number of PC_{ld}s in a representative warp (i.e., a warp that executes the most load instruction count in the SM) of memory-bound GPGPU applications. Figure 9 shows that the chains cover around 65% of the total PC_{ld}s in a representative warp of memory-bound GPGPU applications. Second, we show the maximum repetition

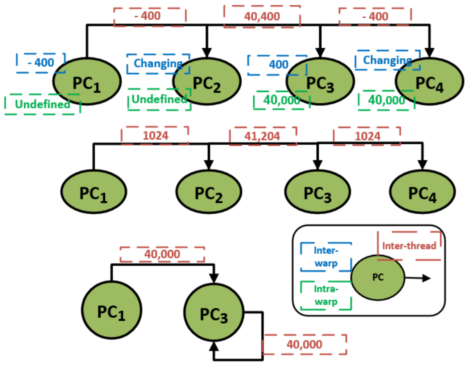


Figure 8: A graph representation of the founded chain between PC_{ld}s in a part of LPS [5] application code.

count of the available chains of strides within a representative warp. Figure 10 indicates that these chains repeat an average of 35 times per warp, highlighting potential opportunities for exploiting them at an intra-warp level. Third, we show the number of memory accesses that can be prefetched using the chains of strides and compare it against the MTA (i.e., Many-Thread-Aware prefetcher [29]) prefetcher in Figure 11.

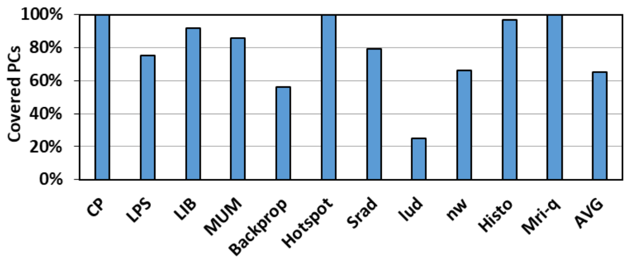


Figure 9: The number of PC_{ld} in a chain normalized to the total number of PC_{ld} in a warp.

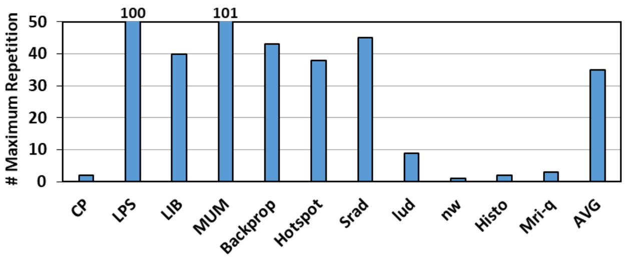


Figure 10: The maximum number of the repetition of a chain of strides within a warp.

We make three key observations: (1) a significant portion (around 70%) of the memory accesses can be prefetched using the chains of strides in memory-bound GPGPU applications, (2) These chains of strides repeat frequently in an intra-warp level, and (3) the coverage resulting from chains of strides is around 15% higher than that of MTA that provides the best coverage among prior solutions (e.g., Intra-warp [27], Inter-warp [29], and Inter-CTA [25]), due to extra prefetching opportunities provided by chains of strides. Applications with low chain repetition achieve acceptable coverage due to the availability of the inter-warp chains of strides.

To conclude, on-chip resources are not used efficiently in memory-bound GPGPU applications due to severe cache contention resulting in reservation fails and bandwidth under-utilization (i.e., the memory wall). To tackle this problem, stride prefetching reduces the average memory latency and improves total GPU performance [25]. Well-acknowledged stride prefetching variants in GPUs limit their potential gain (i.e., the number of accurate prefetched data) due to the fact that they are trained to find a fixed stride between accesses of a load instruction [25, 29]. In this paper, we observe inter-thread chains of strides opportunity that have never been considered in prior work [25, 29]. We observe that a significant part (around 70%) of memory accesses can be prefetched using the chains of strides in memory-bound GPGPU applications, which is 15% higher than that of the MTA (i.e., Many-Thread-Aware prefetcher [29]) prefetcher, which provides the best prefetching opportunities compared to prior solutions. Therefore, a prefetcher based on frequent chains of strides enhances prefetching opportunities.

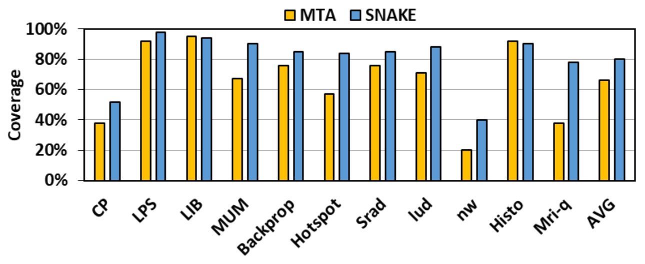


Figure 11: The percentage of memory accesses that can be prefetched using the chains of strides and the MTA (i.e., Many-Thread-Aware prefetcher [29]) prefetcher.

3 SNAKE PREFETCHER

We propose a new prefetching mechanism called Snake that uses chains of strides to effectively prefetch data in three main steps (i.e., detection, prefetching, and throttling). First, in the detection step, Snake extracts the possible chains of strides in the memory-bound GPGPU applications, stores the most frequent effective chains of strides per warp, and promotes the discovered chains of strides to prefetch for other warps if they are detected at least for three warps. To save the required information of the chains, Snake relies on two tables (called Head table and Tail table, indexed with Warp IDs and PC_{ld}s, respectively). Snake trains (fills) the Tail table in the detection step using the information stored in the Head table. Second, in the prefetching step, Snake starts data prefetching using the stored chains of strides in the Tail table. Third, in the throttling step, Snake utilizes a throttling mechanism to control prefetch depth efficiently. For more information on the detection, prefetching, and throttling steps see Section 3.1, Section 3.2, and Section 3.3, respectively.

3.1 Detection Step

To prefetch data using the most frequent chains of strides, Snake relies on two tables (i.e., Head table and Tail table). To store the last executed PC_{ld} of each warp and the corresponding requested address, Snake uses Head table. Whenever a warp executes a load instruction, the warp entry in the Head table is updated to the value of the last executed PC_{ld} and its requested address. Each entry of this table consists of two key fields: (1) a PC_{ld} and (2) the corresponding requested address which is indexed by warp IDs. To avoid losing inter-warp strides in the presence of an aggressive warp scheduler like GTO, Snake stores information from two different warps per PC_{ld}. This involves doubling the warp ID and base address columns. The Tail table is used to store a variety of strides per PC_{ld} (indexed by PC_{ld}) and the train status (i.e., trained or not).

Each entry of this table consists of eight key fields: (1) a head PC_{ld} (called PC1 in this paper), (2) a consecutive PC_{ld} (called PC2 in this paper), (3) the stride between these two PCs (i.e., the Inter-thread stride), (4) the train status of the Inter-thread stride, (5) a vector that stores the ids of the warps that observed the Inter-thread stride pattern (i.e, warp_{ID} vector), (6) Intra-warp stride, (7) the train status of the Intra-warp stride, and (8) the Inter-warp stride. Note that the Inter-warp stride does not need a dedicated train field, since it is added to the table when it is detected at least in three warps (i.e., trained). For more details on the table configuration and field sizes, see Figure 15, and Table 3, respectively. Note that, due to limited on-chip space, we need to prioritize the most effective and frequent chains of strides in the Tail table (using warp_{ID} vector field) and evict others, i.e., the more set bits in warp_{ID} vector, the more warps candidates for prefetching. However, relying solely on the warp_{ID} vector is insufficient for determining which entries are eligible for eviction, because newer PCs, which are more crucial than the older ones, may not have trained on enough warps yet, and they could be given higher priority for eviction. To propose an effective eviction policy, we augment our previous approach with the Least Recently Used (LRU) policy [38]. In our improved eviction policy, we first identify the most eligible entries for eviction using the LRU policy. Then, from the selected group of entries, we evict the one with the fewest '1's in its warp_{ID} vector. For more information on the effectiveness of our eviction policy, refer to Section 5.

Inter-thread strides. Figure 12 shows the detection step based on the Inter-thread strides in the Tail table. The Tail table is referenced upon an entry update in the Head table; Whenever an entry is updated in the Head table, the Head table sends the warp ID, the previous PC_{ld} before the update, the current PC_{ld}, and the stride between their addresses to the Tail table ❶. Snake creates a new entry in the Tail table for the received information (i.e., the previous PC_{ld} (PC1), the current PC_{ld} (PC2)s, and the stride between their related addresses), in three main conditions: (1) there is no matching entry for the received PC1 value ❷, (2) PC1 is matched, but there is no matching entry for the received PC2 value ❸, and (3) PC1 and PC2 are matched, but the received Inter-thread stride is not matched in the Tail table ❸. Finally, Snake sets the corresponding bit (to the warp ID) in the warp_{ID} vector either it finds a match or creates a new entry in the Tail table ❹.

Intra/Inter-warp strides. To prefetch more data, Snake stores Intra-warp and Inter-warp strides as well. Intra-warp strides are

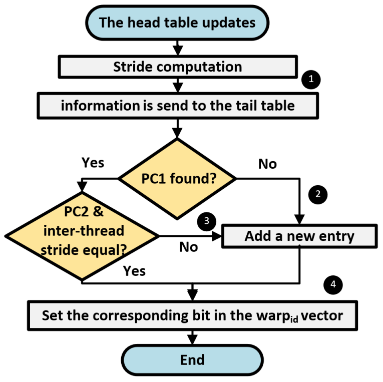

The head table updates
Stride computation
1
information is send to the tail table
PC1 found?
Yes
No
2
PC2 & inter-thread stride equal?
3
No
Add a new entry
Yes
4
Set the corresponding bit in the warp_id vector
End

Figure 12: Detection step flowchart.

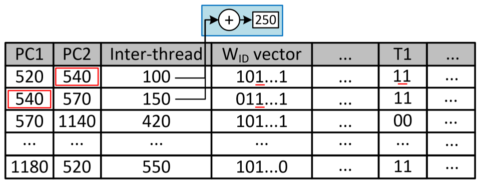
calculated when a PC_{ld} is re-executed in a warp (e.g., loops). A warp re-executes a PC_{ld} in two cases. First, the same PC_{ld} are re-executed consecutively. In this case, Snake calculates the stride between two same consecutive PCs using the previous and current addresses requested by the received PCs. Second, the same PC_{ld} is re-executed after other PCs' execution. Hence, the requested address of the previous executed PC_{ld} is replaced by the last executed PC_{ld}, in the Head table, and the intra-warp stride value can not be computed directly using the received information from the Head table. In this case, Snake calculates the stride between two same non-consecutive PCs accumulating the Inter-thread strides where the warp ID executing the replaced PC_{ld} is set (warp_{ID} vector).

To provide an example clarifying the second case, consider a loop of consecutively executed instructions - PC_1, PC_2, and PC_3. Since PC_1 repeats non-consecutively, Snake calculates the Intra-warp stride by accumulating the stride between PC_1 and PC_2 with that between PC_2 and PC_3, and then subtracting the result from the current address of the new execution of PC_1. This enables Snake to calculate the base address of the previous execution of PC_1 and the Intra-warp stride for PC_1. Snake adds the newly spotted stride in the corresponding field of the Tail table and its train status (T_2) is initiated to '00'. The Inter-warp stride is detected when a fixed stride occurs between at least three warps executing the same PC_{ld}.

3.2 Prefetching

Snake can issue prefetch requests based on three key stored strides: Intra-warp, Inter-warp, and Inter-thread strides. We proceed to issue prefetch requests in the case of not locating the prefetch request in the L1 data cache. To improve the accuracy of prefetching, it is essential to ensure that the detected strides are consistent and repeatable. Therefore, the detected strides need to undergo further observation and confirmation to qualify for prefetching, which is essentially a training process.

In the single instruction, multiple threads (SIMT) execution model, a single instruction is executed on numerous threads within warps. Thus, if a consistent stride is detected across multiple threads, it is highly probable that all threads observe the same stride [29]. In such cases, Snake confirms the stride's consistency across multiple warps before promoting it to issue prefetch requests for future warps. Additionally, in the presence of loops, threads execute a set of instructions repeatedly. Upon detecting a stride on threads for the first time and subsequently observing its repetition, Snake considers the stride to be trained.



PC1	PC2	Inter-thread	W_{ID} vector	...	T1	...
520	540	100	101...1	...	11	...
540	570	150	011...1	...	11	...
570	1140	420	101...1	...	00	...
...	...	...	...	...	...	...
1180	520	550	101...0	...	11	...

Figure 13: Capability of Snake in prefetching a longer chain for each warp.

Once a stride is trained, it becomes eligible for prefetching. This means that when a PC_{ld} is encountered for the first time, Snake issues prefetching requests for all future warps, as soon as the train status in the Tail table is updated to "promoted" (i.e., '10'). If the same PC_{ld} is executed again, Snake checks the relevant entry in the Tail table and issues prefetching requests, considering their Inter-warp or Intra-warp training status. For warps whose corresponding bits are set in the warp_{ID} vector, Snake issues prefetching requests based on the Inter-thread stride training status as well.

Snake is capable of delving deeper and looking for longer chains of strides, as shown in Figure 13. To accomplish this, Snake revisits the Tail table to locate an entry that possesses the same PC1 value as the current entry's PC2 value, while the corresponding bit in warp_{ID} vector is set and the status is trained. Once these conditions are met, a new stride can be calculated and used for further prefetching requests. The depth of Inter-thread prefetching, i.e., the distance from the current PC_{ld}, is controlled by a throttling mechanism (explained in Section 3.3). During the prefetching step, each warp's PC2 and Inter-thread stride are compared with their corresponding values in the Tail table. If they do not match, that particular warp is removed from the warp_{ID} vector. If more than two warps are removed from the warp_{ID} vector of an entry, the inter-thread train status is updated to "not-trained" (i.e., '00'), and it returns to the detection step.

Prefetching can increase the pressure on the cache and data traffic, As discussed in Section 2, reservation fail and contention are side effects of the high TLP in GPUs, and prefetching could potentially exacerbate these issues especially when Snake prefetches based on long chains of strides. To mitigate these issues, we implement a decoupled strategy to separate the storage locations of the prefetched data and the L1 data cache. This approach ensures that prefetched data does not replace useful data that is already in the L1 cache. This decoupling is essential, especially for modern memory-bound GPGPU applications that often process large working data sets. Modern GPU architectures unify the L1 data cache with shared memory to improve storage capacity and bandwidth. However, previous studies reveal that shared memory is not always used efficiently [11, 30].

Once the size of the shared memory is determined in the unified cache, the remaining space is allocated between the prefetch space and the standard L1 data cache space, as illustrated in Figure 14. Each part is indexed by separately configuring each half to reflect

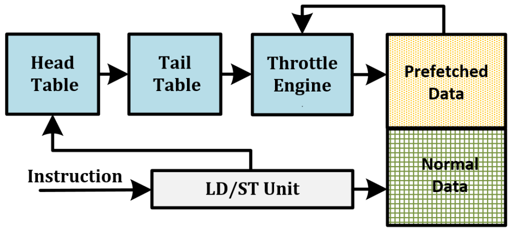


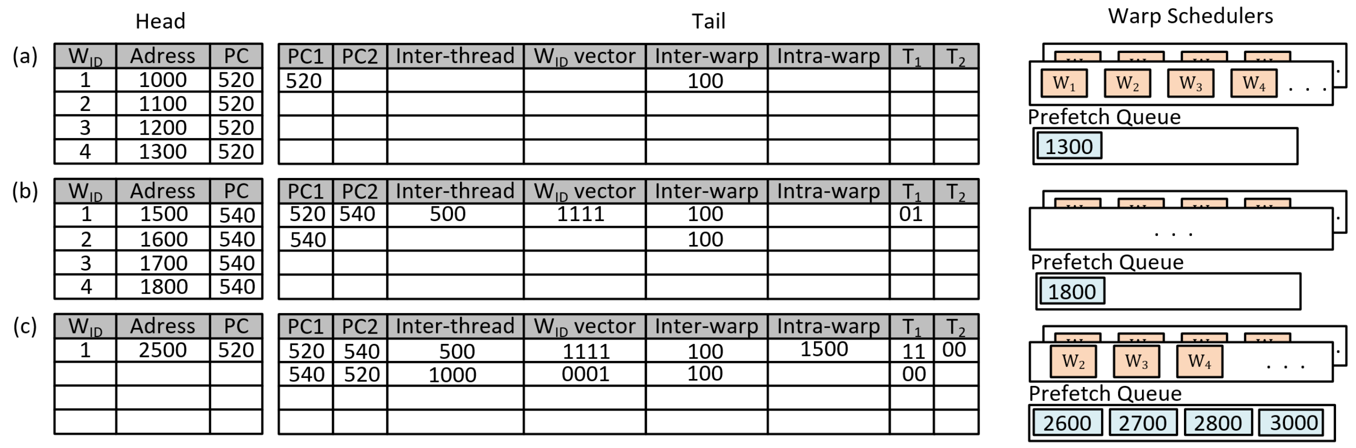
Figure 14: Snake's major components.

the actual configuration of the L1 data cache (for instance, k-way set associative). Snake differentiates between prefetch and L1 data cache entries using a flag. Initially, Snake situates the prefetched data on the upper side and the L1 data cache on the lower side of the unified memory. Both sides are permitted to expand freely until the unified memory is fully occupied. Snake initially restricts the L1 cache to utilize only up to 50% of the available decoupled space until the prefetcher is trained. The training process takes an average of 3 cycles (up to 10 cycles). Upon completion of training, the prefetcher begins to store the prefetched data until it becomes throttled due to a lack of free space or limited bandwidth. In instances of prefetch throttling, we confine the L1 data cache to operate within its designated space for up to 50 cycles (i.e., typically, the data is consumed within 50 cycles.). However, it is always permitted to transfer data that hits the prefetch space to the L1 data space. This transfer does not involve a physical movement but instead occurs through the alteration of the corresponding flag. In instances where there is no free space left in the unified cache, we free up 25% [2] of the unified cache space following the LRU policy. If more than 80% of the prefetched data has been transferred to the L1 data cache, we evict older L1 data cache entries to create free space due to the accurate behavior of prefetching. Otherwise, we evict older prefetched data entries to free up space.

3.3 Throttling

To mitigate the negative effects of aggressive prefetching in Snake, we develop a throttling mechanism based on the available memory space and bandwidth as two main triggering metrics. Our throttling mechanism is triggered under any of these two conditions: (1) When the unified memory has no free space, our throttling mechanism halts prefetching for a predefined number of cycles (50 cycles, See Section 5.4 for more information) [3], to ensure that the prefetched data has enough time to be utilized. (2) When the bandwidth approaches saturation (i.e., when the measured bandwidth reaches around 70% of the theoretical peak bandwidth), our throttling mechanism halts prefetching until the bandwidth saturation is resolved (i.e., when the measured bandwidth reaches 50% of the theoretical peak bandwidth). Our results from applications that are highly affected by memory conflicts between L1 data and prefetched data show that our throttling mechanism substantially reduces the early eviction rate, by up to 20%.

[2]We choose 25% as the hit rate of L1 data cache is up to 75%.
[3]Our results show that enough warps have been executed to consume a considerable amount of the prefetched data after 50 cycles



(a)

W_{ID}	Adress	PC
1	1000	520
2	1100	520
3	1200	520
4	1300	520

PC1	PC2	Inter-thread	W_{ID} vector	Inter-warp	Intra-warp	T_1	T_2
520				100			

(b)

W_{ID}	Adress	PC
1	1500	540
2	1600	540
3	1700	540
4	1800	540

PC1	PC2	Inter-thread	W_{ID} vector	Inter-warp	Intra-warp	T_1	T_2
520	540	500	1111	100		01	
540				100			

(c)

W_{ID}	Adress	PC
1	2500	520

PC1	PC2	Inter-thread	W_{ID} vector	Inter-warp	Intra-warp	T_1	T_2
520	540	500	1111	100	1500	11	00
540	520	1000	0001	100		00	

Figure 15: An illustrative example showing the training and prefetching mechanism of Snake.

3.4 A Prefetch Generation Illustration

The example in Figure 15 showcases the Snake prefetching algorithm. For the sake of simplicity, we consider four warps and a non-greedy scheduling policy. The Head table is filled first, as displayed in Figure 15 (a). As a warp issues a memory request, its warp ID, base address, and PC_{ld} are stored in the Head table. Note that even though each warp has 32 threads and hence can potentially compute 32 distinct base addresses (one for each thread), our empirical evaluations demonstrate that the stride between the threads in a warp is consistently equal. Consequently, we examine the stride between threads and only retain the addresses of the first thread if the stride is equal. Otherwise, we exclude the warp from prefetching. The Tail table updates the inter-warp stride when a fixed stride is detected among three distinct warps executing the same instruction. Upon the execution of a new instruction by a warp, relative data is promptly dispatched to the Tail and Head tables. Figure 15 (b) illustrates the filing process of these tables. In cases where three warps demonstrate similar inter-thread stride behavior, the T_1 status is upgraded to promotion status (i.e., 10), and prefetch requests issue utilizing the inter-thread stride. It is noteworthy that Snake accords priority to the inter-thread stride over the inter-warp stride due to its higher accuracy. Whenever a warp executes an instruction that corresponds to a PC_{ld} found in the Tail table Snake initiates prefetch requests utilizing the trained strides as illustrated in figure 15 (c). Furthermore, Snake utilizes a mechanism for monitoring the stride between two instances of the same PC_{ld} in order to determine the intra-warp stride. Upon establishing consistency of intra-warp stride in three distinct warps, Snake proceeds to update the train status associated with that particular PC_{ld}(i.e., T2). Note that examples supporting variable strides are possible, whereby different entries in the table may store the same PC1 and PC2 with various strides for different groups of warps. However, for the sake of simplicity, we demonstrate a single stride between two consecutive PC_{ld}s (i.e., PC1 and PC2).

3.5 Compiler Optimization Support

Software and compiler optimization techniques aim to boost code execution efficiency, reuse data, reduce data transfers (e.g., between CPU and GPUs, or compute and memory units), and refine instruction scheduling to fully engage available computational resources by considering the memory access patterns of the applications [16].

Snake performs autonomously but synergistically in conjunction with software/compiler optimization techniques like tiling. Tiling promotes data locality by partitioning input data into smaller cache-friendly tiles and performing computations on each tile individually. It is an important optimization for workloads like matrix multiplication. Matrix multiplication is a fundamental task in scientific and learning applications, including Deep Neural Networks (DNNs). Tiling improves cache utilization, reduces average memory latency, and avoids main memory bandwidth bottlenecks [28]. To achieve these benefits, compilers/programmers determine the optimal tile dimensions to fit into the cache while considering the available bandwidth and interconnection traffic.

When Snake engages with tiling optimization, it detects the strides between tiles by calculating the distances between the elements of tiles and proactively prefetches a sequence of addresses from the following tile. Note that Snake consistently prioritizes the data requested from the L1 data cache, and hence, the L1 data cache can always utilize the unified cache as much as needed. Although Snake constrains the utilization of the L1 data cache to only half of the unified space during the training, non-interrupted storage of tiles is ensured during the training of Snake, even when dealing with tiles that exceed 50% of the available unified cache space. It is because the act of fetching these large tiles from lower memory levels demands more time than the designated 10-cycle period, which represents the maximum required time for training Snake.

For tiles that exceed 50% of the unified cache space, Snake prefetches a segment of the subsequent tile into the available space. Consequently, this prefetching action can lead to Snake becoming throttled, as the unified cache experiences a shortage of free space. Note that even if the execution of the current tile surpasses the time required for prefetching the forthcoming tile's segment and the subsequent throttling period (50 cycles), Snake does not evict the prefetched data. It is because when there is no free space, Snake stops prefetching and keeps the prefetched data until the L1 data cache experiences a miss (see Section 5.6 for quantitative analysis).

4 METHODOLOGY

Simulation. In this study, we evaluate the performance and power consumption of the Snake, utilizing Accel-Sim [21] v1.2.0. Specifically, we analyze the power consumption of Snake using Accel-Wattch [19] v1.0. Detailed configuration parameters are listed in Table 1 modeling the NVIDIA VOLTA V100 architecture.

Baseline benchmark suites. We evaluate the effectiveness of Snake compared to other prefetching mechanisms on Rodinia [31], Parboil [44], and ISPASS [5] benchmark suites. Each application is simulated until the end of its execution or when the simulated instruction count reaches 1 billion. Table 2 shows a list of our benchmark suites.

Comparison Metrics. We compare Snake to a variety of state-of-the-art prefetching mechanisms in terms of overall IPC, energy consumption, coverage (the number of correctly predicted addresses to the total number of demand addresses), and accuracy (the number of correctly predicted addresses that are timely enough to be used by demand requests to the total number of demand addresses).

Comparison Points. To measure the effectiveness of Snake (i.e., our proposed chained-based prefetching method alongside decoupling and throttling mechanisms) we compare it to the baseline GPU and 9 other prefetching mechanisms. (1) INTRA is an Intra-warp prefetcher simply issues prefetch requests into the existence of loop iterations. (2) INTER is an inter-warp prefetcher that enables threads to prefetch for future threads of future warps. (3) MTA is a hardware implementation of Many-Thread-Aware prefetcher [29]. (4) CTA is an implementation of the CTA-Aware prefetcher [25]. (5) Tree is the most recent spatial prefetcher focusing on CPU-GPU [15]. We adopt this work in the GPU context which considers 64KB chunks of the global memory and prefetches them to the L1 data cache. (6) s-Snake is a simple model of Snake that only exploits the chains of strides without exploiting Intra-warp and Inter-warp strides. (7) Snake-DT is Snake without exploiting decoupling and throttling mechanisms. (8) Snake-T is Snake that exploits the decoupling mechanism without exploiting throttling mechanisms. (9) Snake+CTA, combines Snake with CTA-Aware to show that they are orthogonal.

Table 1: Baseline GPU configuration.

Parameter	Value
Number of SM	80
Core clock	1530 MHz
Scheduler	Greedy-then-Oldest
Number of scheduler per SM	4
Number of threads per SM	2048
Register file size per SM	65536
Unified cache	128KB, 256-way, 128B, 4-bank, 28-cycle
MSHR	512 entries, 8 merge capability
L1 instruction cache	128KB, 16-way, 128B
Constant cache	64KB, 8-way, 64B-line
L2 cache	96KB per sub-partition, 24-way, 128B-line, 64-bank, 212-cycle
DRAM parameters (ns)	t_{CCD}=1, t_{RRD}=3, t_{RCD}=12, t_{RAS}=28, t_{RP}=12, t_{RC}=40, t_{CL}=12, t_{WL}=2, t_{CDLR}=3, t_{WR}=10, t_{CCDL}=2, t_{RTPL}=3

Table 2: Benchmark suites

Benchmark	Abbr.
Coulombic Potential [5]	CP
3D Laplace Solver [5]	LPS
LIBOR Monte Carlo [5]	LIB
MUMmerGPU [5]	MUM
Back Propagation [31]	Backprop
HotSpot [31]	Hotspot
Speckle Reducing Anisotropic Diffusion [31]	Srad
LU Decomposition [31]	lud
Needleman-Wunsch [31]	nw
Histogram [44]	histo
mri-q [44]	MRQ

5 EVALUATION

5.1 Coverage/Accuracy Analysis

Figure 16 shows the coverage of Snake compared to the well-known GPU prefetchers. Based on these results, We make seven main observations. First, Snake demonstrates superior coverage compared to other methods. It is able to accurately predict 80% of memory requests by effectively utilizing chains of strides in combination with intra-warp and inter-warp strides. Second, Snake outperforms the state-of-the-art prefetcher (i.e., MTA in terms of coverage) by providing a 15% higher coverage of future memory accesses. Third, s-Snake achieves a noteworthy prediction accuracy of around 70% in forecasting future memory accesses. Fourth, using a throttling mechanism may slightly reduce coverage (by about 2%), but the improved accuracy (by about 20%, Figure 17) outweighs this drawback. Fifth, the coverage of INTRA and CTA-Aware is comparatively lower, primarily attributed to the absence of loops and the missed prefetch opportunities during the inter-CTA stride detection period. Sixth, Snake+CTA shows lower coverage than Snake because the inter-CTA coverage is low initially. Seventh, nw application exhibits low coverage compared to other applications, despite having regular memory access patterns. This is due to the low number of repetitions of these patterns in nw.

Figure 17 shows the accuracy of Snake compared to the well-known GPU prefetchers. The results of our study reveal that Snake exhibits high accuracy, surpassing 90% across various applications. This remarkable performance empowers Snake to effectively anticipate and cover 75% of future memory accesses in a timely manner, on average. The reason behind the high accuracy of Snake is that the inter-thread stride in Snake prefetches consecutive PC$_{ld}$ requests, reducing the chance of late prefetching. Additionally, The decoupling and throttling mechanisms in Snake helps reduce the early eviction rate for more accurate prefetching. It is noteworthy that without decoupling, Snake experiences a 50% loss in accuracy due to early eviction by normal data from the L1 data cache. However, inter-warp stride in Snake could result in inaccurate prefetches.

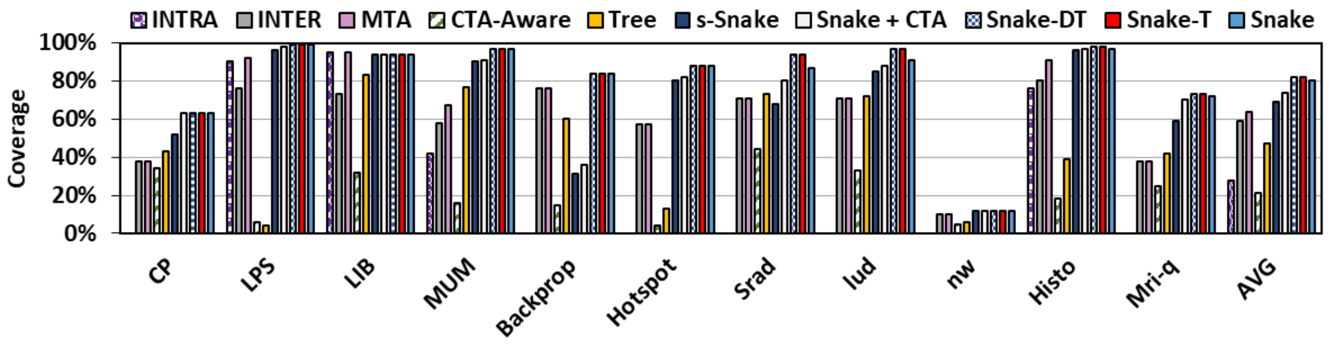


Figure 16: Prefetch coverage (the number of correctly predicted addresses to the total number of demand addresses).

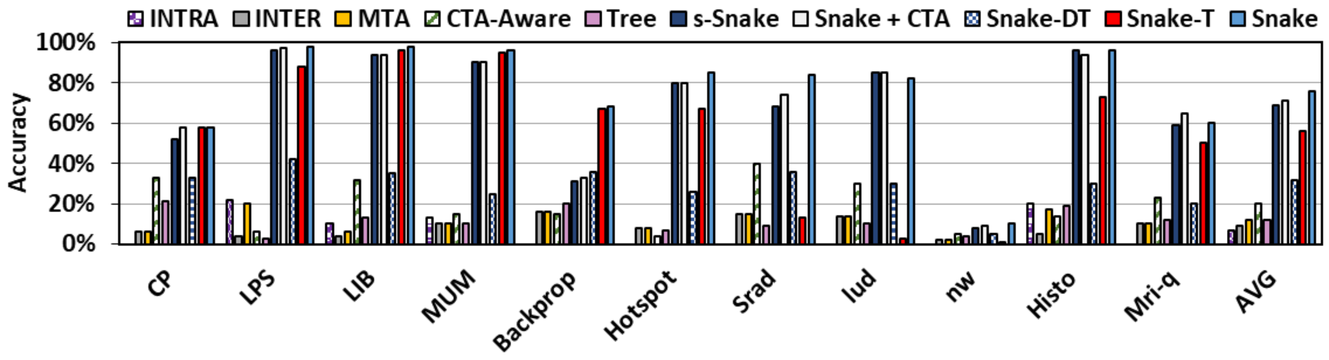


Figure 17: Prefetch accuracy (the ratio of timely correctly predicted addresses to the total number of demand addresses).

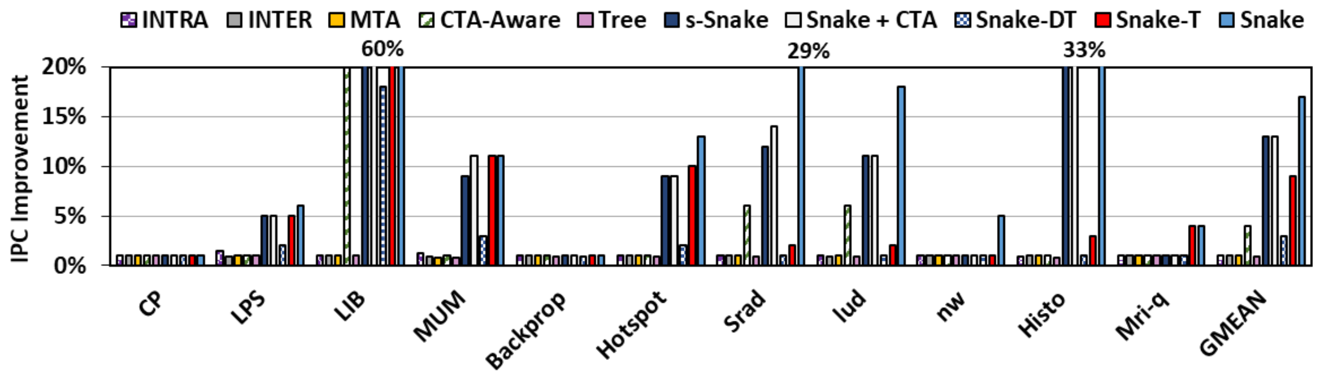


Figure 18: Performance improvement of prefetching mechanisms.

5.2 Performance Analysis

Figure 18 demonstrates the performance improvement of Snake compared to the well-known GPU prefetchers, normalized to the baseline. We make four key observations. First, Snake improves performance by 17% (up to 60%), on average. Second, LIB experiences the highest performance improvement, because Snake increases the L1 data cache hit rate by 10×, resulting in lower memory latency. Third, Snake enhances the performance of Histo and Srad applications by 33% and 29%, respectively. Despite these applications exhibiting a high hit rate in the baseline, they experience bursty misses, leading to resource congestion. However, thanks to Snake's precise prefetching, it effectively reduces memory latency, mitigates bursty misses, and significantly decreases congestion-related stalls. Fourth, Snake outperforms Snake-DT and Snake-T by 13% and 7%, respectively, showing the effectiveness of decoupled and throttling mechanisms. Decoupling increases the effectiveness of CTA-Aware, MTA, and Tree by 5%,3%, and 2%, respectively. These results show Snake outperforms the decoupled versions of competitors.

The performance improvement achieved by Snake is due to the reduction of two main factors: (1) miss rate, and (2) bursty misses and reservation fails. This improvement is attributed to the high coverage, accuracy, and low cache contention in Snake. These benefits are due to prefetching a chain of strides and utilizing a decoupled mechanism.

5.3 Energy Analysis

In Figure 19, we present the energy consumption results of Snake, normalized to the baseline. The results reveal that Snake consumes, on average, 17% less energy. This reduction in energy consumption can be attributed to two main factors: (1) proactive data prefetching, which reduces the effective memory access latency by fetching data that is likely to be accessed in the near future, and (2) reduction in idle time of warps and repetitive accesses due to reservation fail.

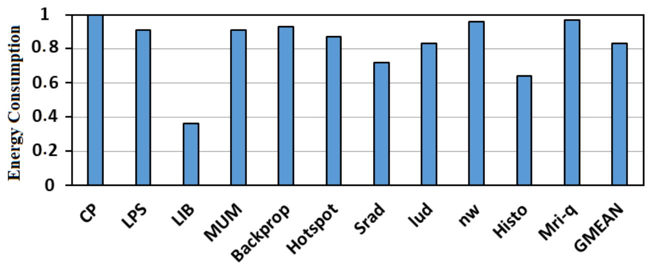


Figure 19: Energy consumption of Snake normalized to the baseline.

5.4 Sensitivity Analysis

We conduct a sensitivity analysis of prediction parameters, precisely Entry size, eviction policy, and throttling time interval, to evaluate their impact on the Snake's performance.

Tail table entry size. Figure 20 shows the impact of varying the number of Tail table entry size utilizing the main eviction policy (considers both LRU policy and the number of '1's in its warp$_{\mathrm{ID}}$ vector). The coverage of Snake decreases by only 8% when the entry size is limited to 10. To further evaluate the space overhead of Snake, we analyze the space utilization with various entry sizes as shown in Figure 21.

Eviction policy. Figure 22 shows the coverage of Snake that considers the number of '1's in its warp$_{\mathrm{ID}}$ vector (i.e., the entry with the fewest '1's is evicted) without using the LRU mechanism in its eviction policy. Comparing the results of Figure 20 and Figure 22, we conclude that Snake gains higher coverage using both LRU policy and the number of '1's in its warp$_{\mathrm{ID}}$ vector rather than considering only the number of '1's in its warp$_{\mathrm{ID}}$ vector.

Throttling time intervals. Figure 23 presents the trade-off between accuracy and coverage of Snake for different throttling time intervals. The results show that a longer duration of powering off Snake can increase the accuracy by reducing the early eviction rate. However, an excessively long duration of powering off Snake can result in a lower prefetching opportunity and coverage. Based on the findings in Figure 23, turning off Snake for 50 cycles can achieve an acceptable level of accuracy (75%) with only 2% coverage loss.

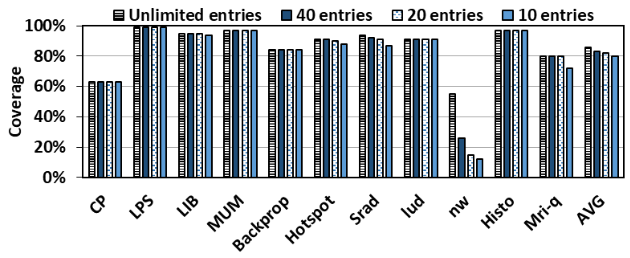


Figure 20: Effect of Tail table entry size on the coverage.

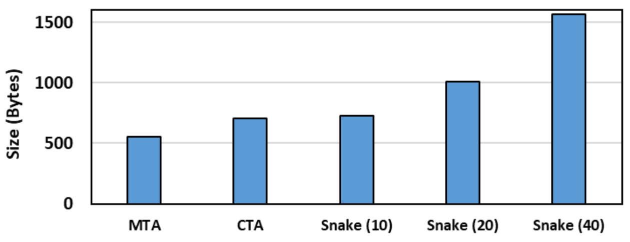


Figure 21: Hardware cost comparison.

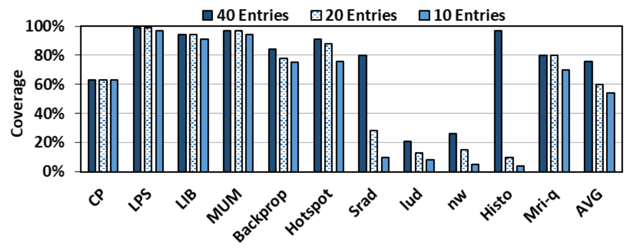


Figure 22: Effect of eviction policy on the coverage with different Tail table entry size.

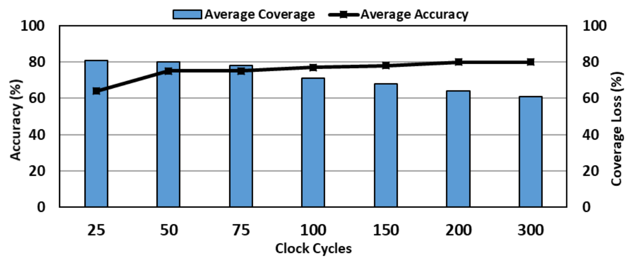


Figure 23: Trade-off between accuracy and coverage of Snake for different throttling time intervals.

5.5 Cost Analysis

Snake utilizes two tables per SM (Head and Tail tables). The Head table consists of five columns (two warp IDs, two base addresses, and one PC$_{ld}$), with a total of N rows ($N = \#warps/2$). Note that two columns (one warp ID and one base address) are added to support greedy-like schedulers. Conversely, for non-greedy schedulers, we use only three columns, as depicted in Figure 15. The Tail table has 10 entries. Our results show that having 10 entries in the Tail table is enough for maintaining a balanced coverage-to-memory space ratio (see section 5.4). We report the area and power overheads in Table 3 and Section 5, respectively.

Area. To estimate the area overhead, we model the proposed tables in CACTI V7.0 [48] for 22nm technology and scale the extracted area to 12nm (i.e., the fabrication technology in NVIDIA Volta V100 [36]). Considering the tables in all SMs, Snake imposes less than 1% area overhead over the NVIDIA Volta V100 [36]) die size (815 mm^2).

Power. To calculate the power consumption of each component, we synthesize the Hardware Description Language (HDL) model of the tables in Snake for NanGate 28nm open cell library using Synopsys Design Compiler, scaled to 12nm. We then import the calculated numbers into AccelWattch [19] and measure the power overhead of our proposal. We observe that Snake consumes 6.4 pJ per access and 6 mW of static power and imposes a negligible power overhead (less than 1%) due to the additional hardware. Note that Snake reduces GPU energy consumption when running memory-bound GPGPU applications by providing low-latency access.

Latency. Snake organizes the detection and prefetching steps into a pipeline, overlapping them with the issuance of memory requests from the warps. We implemented the prefetcher using Verilog HDL, synthesized it in the Synopsys Design Compiler [1], and performed placement and routing in Cadence SOC Encounter [2]. Table 3 reports detailed hardware characteristics. The latency of Snake is two cycles, as the search process involves a comparator circuit that searches 10 available PC1s in the Tail table at once, along with two AND gates to verify the warp ID and training status.

Table 3: Snake's tables parameters.

Table	Configuration	Total
Head	14 bytes per entry, 32 entries	448 bytes
Tail	32 bytes per entry, 10 entries	320 bytes

5.6 Tiling Sensitivity Analysis

To evaluate Snake in the presence of tiled/cache-fitting applications, we implement a tiled convolution, modeled by matrix multiplication while varying the tile size from 0% (i.e., no tiling) to 100% of the unified memory space. Figure 24 reports performance and energy consumption for (1) tiled implementation (*Tiled*) and (2) fusion of tiling alongside Snake (*Snake+Tiled*), normalized to the baseline (without tiling and prefetching). We make two key observations.

First, *Tiled* improves the performance and energy consumption respectively by 5%/8%/10%/7% and 5%/7%/8%/6%, on average, for 25%/50%/75%/100% tile size, over the baseline. *Snake+Tiled* improves the performance and energy consumption by 13%/15%/17%/7% and 12%/13%/14%/6%, on average, for 25%/50%/75%/100% tile size, compared to the baseline. The maximum improvement is achieved by *Tiled* and *Snake+Tiled* implementations for the tile occupying 75% of the unified cache space. It is because larger tile sizes (1) need higher TLP which may exceed the maximum supported TLP, (2) increase the intermediate data storage and data movements, and (3) increase contention in the interconnection network and queues across different levels of the memory hierarchy. Second, *Snake+Tiled* improves the performance and energy consumption compared to *Tiled* by 2.6×/1.9×/1.7× and 2.4×/1.8×/1.7×, on average, for 25%/50%/75% tile sizes, respectively. For tiles occupying all the unified cache space, *Snake+Tiled* is always on throttling mode, and hence, achieves the same improvements in terms of both performance and energy consumption over *Tiled.*

In summary, executing a tile-based implementation of convolution, the greatest improvements are observed both with and without Snake when the tile size occupies 75% of the unified cache space. Snake's improvements are attributed to its efficient detection and prefetching of subsequent tiles, leading to reduced memory access latency and fewer stalls.

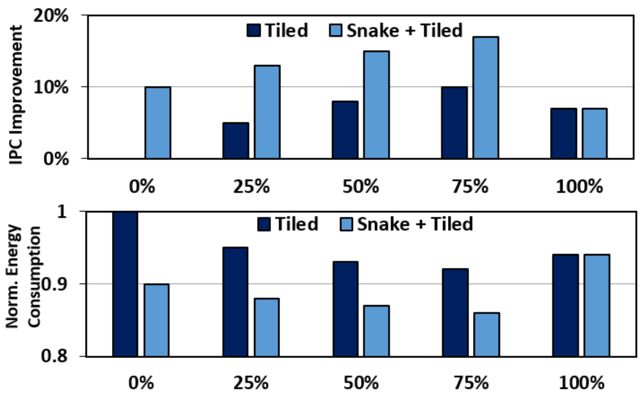


Figure 24: Effect of tiling with/without prefetching on IPC and energy consumption.

5.7 Sensitivity Analysis on Decoupling

To explore the impact of decoupling on the behavior of L1 data cache, we devise a variant of Snake that stores prefetched data in a buffer distinct from the unified memory, called Isolated-Snake. Figure 25 showcases the L1 data cache hit rate for the baseline GPU, Snake, and Isolated-Snake. The hit rate of the L1 data cache is 45%/79%/84% in baseline/Snake/Isolated-Snake, on average, indicating Snake increases the hit rate significantly, which is within 5% margin of Isolated-Snake's. It is because Snake prefetches data with high accuracy and the L1 data cache uses the prefetched data, hence, eliminating a portion of future miss requests. Thanks to Snake's timeliness and its decoupling mechanism effectiveness, Snake achieves a close hit rate to Isolated-Snake.

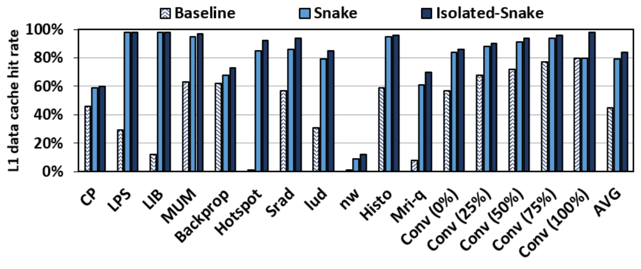


Figure 25: Hit rate of L1 data cache.

6 RELATED WORK

Snake is the first GPU prefetcher that utilizes chains of strides. We briefly discuss well-known CPU and GPU prefetching mechanisms in the following.

6.1 CPU Prefetching Mechanisms

Prefetching was initially proposed for CPUs to address the memory wall problem and improve performance. Here we briefly explain the state-of-the-art prefetchers proposed for CPUs. Domino [6] is a modern prefetching approach based on temporal locality. Domino uses a novel indexing technique that saves a corresponding pointer for each address in the history, pointing to the next recent address instead of the current one. Bingo [7] is a spatial data prefetcher that uses two sets of information related to different pages to find correlations between various memory accesses. Bingo starts searching from the long-term event (i.e., PC_{ld} + Address). If there is no corresponding entry in the table for this event, it moves to the short-term event (i.e., PC_{ld} + Offset).

Learning-based methods have a high ability in detecting repeated patterns, which allows them to identify regular or irregular memory access patterns. These models range from light-weight algorithms (e.g., decision tree [32]) to deep neural networks [8]. However, using light-weight networks reduces the prediction accuracy, while deep networks often have memory overhead, inference delays, and high energy consumption, which make them difficult to implement on a chip and may outweigh their potential benefits. Hardware prefetchers designed for CPUs cannot be directly applied to GPUs due to their inherent characteristics.

6.2 GPU Prefetching Mechanisms

Lee et al. [29] observed that future access patterns in GPUs could be predicted using Intra-warp and Inter-warp strides and proposed a hardware prefetcher specifically designed for GPUs, called MTA. While Snake improves the chance of prefetching by utilizing Intra-warp and Inter-warp strides, it also introduces inter-thread stride to overcome the limitation of Intra-warp stride coverage and Inter-warp stride accuracy. Utilizing a chain of strides, as opposed to a fixed stride, in Intra-warp processing enables the identification of additional prefetch opportunities between threads (inter-thread stride). This approach facilitates the issuance of prefetch requests for future PC_{ld}, ensuring timely and accurate prefetching, which differs from the approach involving Inter-warp stride. Koo et al. [25] proposed a prefetcher and scheduler, called CTA-aware that considers the trade-off between timeliness and accuracy of Inter-warp stride. They found that warps within a CTA have the same stride but across CTAs the stride changes. However, warps within a CTA are scheduled close in time and can not hide memory access latency. Snake could be used orthogonally to the CTA-aware prefetcher to reduce the negative effect of the Inter-warp stride on the prefetching accuracy. However, computing the base address of a CTAs in the CTA-aware prefetcher is time-consuming and results in lower coverage. Ganguly et al. [15] proposed a spatial-based locality prefetcher using a binary tree structure. However, aggressive prefetching in spatial-based locality prefetchers hurts the performance in GPUs due to limited memory resources. Although Snake can be considered an aggressive prefetcher, it accurately calculates the stride between memory accesses to prevent prefetching useless data.

7 CONCLUSION

We proposed Snake, a stride-based prefetching that leverages chains of strides. Snake increases the accuracy by (1) decoupling prefetch data from L1 data cache, and (2) exploiting a throttling mechanism to turn off Snake whenever it is necessary. Our results showed that Snake accurately prefetch 75% of future memory accesses, improves the average IPC and average energy consumption by 17% at the expense of 1% L1 data cache area overhead.

REFERENCES

[1] 2000. Design Compiler, Synopsys inc.

[2] 2015. Cadence SoC Encounter. https://www.cadence.com/.

[3] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. 2015. Exploiting inter-warp heterogeneity to improve gpgpu performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 25–38.

[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 136–150.

[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 163–174.

[6] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[7] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 399–411.

[8] Chandranil Chakraborttii and Heiner Litz. 2022. Deep Learning based Prefetching for Flash. In *Nonvolatile Memory Workshop (NVMW)*.

[9] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 128–139.

[10] Sina Darabi, Negin Mahani, Hazhir Baxishi, Ehsan Yousefzadeh-Asl-Miandoab, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2022. NURA: A framework for supporting non-uniform resource accesses in GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–27.

[11] Sina Darabi, Ehsan Yousefzadeh-Asl-Miandoab, Negar Akbarzadeh, Hajar Falahati, Pejman Lotfi-Kamran, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2022. OSM: Off-chip shared memory for GPUs. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3415–3429.

[12] Hajar Falahati, Mania Abdi, Amirali Baniasadi, and Shaahin Hessabi. 2013. ISP: Using idle SMs in hardware-based prefetching. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)*. IEEE, 3–8.

[13] Hajar Falahati, Shaahin Hessabi, Mania Abdi, and Amirali Baniasadi. 2015. Power-efficient prefetching on GPGPUs. *The Journal of Supercomputing* 71 (2015), 2808–2829.

[14] Hajar Falahati, Masoud Peyro, Hossein Amini, Mehran Taghian, Mohammad Sadrosadati, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2021. Data-Aware compression of neural networks. *IEEE Computer Architecture Letters* 20, 2 (2021), 94–97.

[15] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.

[16] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.

[17] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. Orchestrated scheduling and prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 332–343.

[18] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.

[19] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 738–753.

[20] Mohammad Mahdi Keshtegar, Hajar Falahati, and Shaahin Hessabi. 2015. Cluster-based approach for improving graphics processing unit performance by inter streaming multiprocessors locality. *IET Computers & Digital Techniques* 9, 5 (2015), 275–282.

[21] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[22] Mohsen Kiani and Amir Rajabzadeh. 2018. Efficient cache performance modeling in GPUs using reuse distance analysis. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–24.

[23] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[24] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 163–175.

[25] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. 2018. Cta-aware prefetching and scheduling for GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 137–148.

[26] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 307–319.

[27] Nagesh B Lakshminarayana and Hyesoon Kim. 2014. Spare register aware prefetching for graph algorithms on GPUs. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*. IEEE, 614–625.

[28] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review* 25, Special Issue (1991), 63–74.

[29] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-thread aware prefetching mechanisms for GPGPU applications. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 213–224.

[30] Bingchao Li, Jizeng Wei, Jizhou Sun, Murali Annavaram, and Nam Sung Kim. 2019. An efficient GPU cache architecture for applications with irregular memory access patterns. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 3 (2019), 1–24.

[31] Zheng-Xiang Li, SVb Bogdanova, AS Collins, Anthony Davidson, Bert De Waele, RE Ernst, ICW Fitzsimons, RA Fuck, DP Gladkochub, J Jacobs, et al. 2008. Assembly, configuration, and break-up history of Rodinia: a synthesis. *Precambrian research* 160, 1-2 (2008), 179–210.

[32] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–10.

[33] Pejman Lotfi-Kamrana and Hamid Sarbazi-Azadb. 2022. Introduction to data prefetching. *Data Prefetching Techniques in Computer Systems* (2022), 1.

[34] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. 2020. Efficient Nearest-Neighbor Data Sharing in GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2020), 1–26.

[35] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. 2018. Neda: Supporting direct inter-core neighbor data exchange in GPUs. *IEEE Computer Architecture Letters* 17, 2 (2018), 225–229.

[36] Tesla NVIDIA. [n. d.]. V100 Volta Architecture. *URL http://www. nvidia. com/object/volta-architecture-whitepaper. html* ([n. d.]).

[37] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. 2016. APRES: Improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH computer architecture news* 44, 3 (2016), 191–203.

[38] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[39] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–131.

[40] Reena Panda, Yasuko Eckert, Nuwan Jayasena, Onur Kayiran, Michael Boyer, and Lizy Kurian John. 2016. Prefetching techniques for near-memory throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–14.

[41] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. 2019. ITAP: Idle-time-aware power management for GPU execution units. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–26.

[42] Mohammad Sadrosadati, Amirhossein Mirhosseini, Ali Hajiabadi, Seyed Borna Ehsani, Hajar Falahati, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2021. Highly concurrent latency-tolerant register files for GPUs. *ACM Transactions on Computer Systems (TOCS)* 37, 1-4 (2021), 1–36.

[43] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*. 141–152.

[44] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[45] Steven P Vander Wiel and David J Lilja. 1997. When caches aren't enough: Data prefetching techniques. *Computer* 30, 7 (1997), 23–30.

[46] Bin Wang, Yue Zhu, and Weikuan Yu. 2016. OAWS: memory occlusion aware warp scheduling. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 45–55.

[47] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and fair multi-programming in GPUs via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 247–258.

[48] Steven JE Wilton and Norman P Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of solid-state circuits* 31, 5 (1996), 677–688.

[49] Ping Xiang, Yi Yang, and Huiyang Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 284–295.

Received 28 April 2023; revised 9 Jul 2023; accepted 3 Sep 2023