Address-Stride Assisted Approximate Load Value Prediction in GPUs

Haonan Wang
College of William & Mary
hwang07@email.wm.edu

Mohamed Ibrahim
College of William & Mary
maibrahim@email.wm.edu

Sparsh Mittal
IIT Hyderabad
sparsh@iith.ac.in

Adwait Jog
College of William & Mary
ajog@wm.edu

ABSTRACT

Value prediction holds the promise of significantly improving the performance and energy efficiency. However, if the values are predicted incorrectly, significant performance overheads are observed due to execution rollbacks. To address these overheads, value approximation is introduced, which leverages the observation that the rollbacks are not necessary as long as the application-level loss in quality due to value misprediction is acceptable to the user. However, in the context of Graphics Processing Units (GPUs), our evaluations show that the existing approximate value predictors are not optimal in improving the prediction accuracy as they do not consider memory request order, a key characteristic in determining the accuracy of value prediction. As a result, the overall data movement reduction benefits are capped as it is necessary to limit the percentage of predicted values (i.e., prediction coverage) for an acceptable value of application-level error.

To this end, we propose a new Address-Stride Assisted Approximate Value Predictor (ASAP) that explicitly considers the memory addresses and their request order information so as to provide high value prediction accuracy. We take advantage of our new observation that the stride between memory request addresses and the stride between their corresponding data values are highly correlated in several applications. Therefore, ASAP predicts the values only for those requests that have regular strides in their addresses. We evaluate ASAP on a diverse set of GPGPU applications. The results show that ASAP can significantly improve the value prediction accuracy over the previously proposed mechanisms at the same coverage, or can achieve higher coverage (leading to higher performance/energy improvements) under a fixed error threshold.

CCS CONCEPTS

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

KEYWORDS

GPU, Value Prediction, Approximation, Scheduling

ACM Reference Format:
Haonan Wang, Mohamed Ibrahim, Sparsh Mittal, and Adwait Jog. 2019. Address-Stride Assisted Approximate Load Value Prediction in GPUs. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330362

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330362

1 INTRODUCTION

Graphics Processing Units (GPUs) are capable of providing very high peak throughput and memory bandwidth at a competitive power budget [9–11, 43, 45]. However, continuous scaling of GPU performance and energy efficiency is challenging primarily because of the high system energy consumption [12] caused by excessive data movement across different levels of the memory hierarchy, and limited memory bandwidth. To address these bottlenecks, previous works have proposed several techniques such as data compression [42], warp scheduling for better cache and memory performance [8, 9, 11, 32], and cache management [30, 31]. Another promising strategy for reducing data movement is value prediction, whereby the values are not necessarily required to be *fetched* from memory as they can be *predicted* at the core. In the context of CPUs, previous techniques [3, 4, 17, 26–28, 38, 39] used to both predict and fetch the data. The predicted values are later compared with the fetched values. If the prediction turns out to be correct, the data-dependent stall cycles are reduced significantly. However, in the case of a misprediction, the execution is rolled back leading to the flushing of the dependent instructions in the pipeline. Such performance and data movement overheads are the critical impediments towards leveraging the benefits of value prediction. To address the challenges of *precise* value prediction, recent research has explored *approximate* value usage [13, 20, 35, 36, 44, 45], which leverages the observation that for approximable applications the requirement of rollbacks can be omitted as long as the application-level loss in quality is within an acceptable range.

While rollback-free value approximation has received significant attention in the context of CPUs [13, 20, 35, 36, 41], only a few works have explored it in the context of GPUs [44, 45]. Application execution in GPUs relies on multi-threading, where associated threads are scheduled on GPU cores at the granularity of *warps*, where a warp usually consists of 32 threads. Each load instruction in a warp can generate one or more cache block request depending on how well the data is coalesced across threads within the warp. As hundreds of warps can concurrently execute and cache sizes in GPUs are much smaller than CPUs [22], data movement between caches and memory is a serious performance and energy efficiency bottleneck [8, 9, 11, 42]. If values of these requests can be correctly predicted at the core, the data movement and stall cycles can be significantly reduced thereby improving latency tolerance, performance, and energy efficiency. However, if the predictor predicts incorrectly, each mispredicted cache line leads to a certain level of quality loss in the application's final output. This quality loss is dependent on many factors such as the prediction coverage (defined as the ratio of predicted load requests to the total load requests), the magnitude of error in value prediction, and the error resilience of instructions that use erroneous values as their operands. Therefore, if values can be predicted more accurately, higher coverage can be applied for better performance and energy efficiency.

The goal of this paper is to improve the accuracy of value prediction in GPUs. One of the major challenges in achieving this goal is to identify the value stride pattern(s) in a highly multi-threaded environment where thousands of memory requests can be on-the-fly and their access order is highly dependent on GPU-specific features such as warp scheduling and coalescing. Previous works for CPUs used large per-thread prediction tables to achieve high accuracy [21, 37, 40]. However, it can become prohibitively expensive to apply those approaches directly to the highly multi-threaded environment in GPUs [45]. To address this problem, we take advantage of our key new observation that consideration of memory addresses and the relationship with their value strides is effective for providing high value prediction accuracy. Specifically, we find that for many realistic inputs used by GPGPU applications, particular address strides have linear correlations with their value strides. For example, Figure 1 shows that for the extracted pixels, an address stride of $1\times data\_size$ correlates to a value stride of -1. Meanwhile, an address stride of $1\times row\_size$ correlates to a value stride of 1.

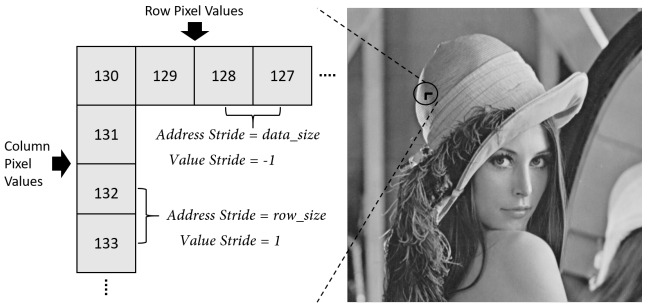


Figure 1: Pixel values of consecutive row and column positions.

Based on this new observation, we propose an Address-Stride Assisted Approximate Value Predictor (ASAP), which predicts the values only if it detects strides in their corresponding addresses. Each entry in the ASAP prediction table carefully keeps track of one type of address stride and their corresponding value stride. We find that as the number of address stride patterns in typical GPGPU applications is usually limited, the number of prediction table entries is significantly reduced, thereby making it area and power-efficient (Section 4). We also show that ASAP remains effective even under different address patterns, which can be influenced by warp scheduling and coalescing (Section 6).

To the best of our knowledge, this is the first work that shows that there is a high correlation between address stride and value strides in several GPGPU applications and this observation can be used to design an efficient GPU-specific value predictor. Our simulation results across a set of diverse GPGPU applications show that ASAP can significantly improve the prediction accuracy over the state-of-the art GPU value predictor while providing high performance improvement (up to 40%) and energy reduction (up to 30%). Specifically, the previously proposed RFVP-style value predictor [45] incurs 3.48% (up to 40.08%) and 8.10% (up to 63.59%) Application Error, at 10% and 20% coverage, respectively. In contrast, under a similar area budget, our ASAP predictor produces on average only 0.26% and 0.43% Application Error, respectively.

2 BACKGROUND

This section provides background on the GPU architecture followed by details of the existing value prediction techniques in GPUs.

2.1 Baseline Architecture and Metrics

Figure 2 shows the baseline GPU architecture consisting of cores (also known as Streaming Multi-processors (SMs)) and memory partitions. Each core has its private L1 cache and the L1 cache is connected to a slice of L2 cache via an on-chip interconnect. The L2 cache is further connected to the off-chip GPU memory. We assume each SM is also attached to a value predictor (VP). We simulate our baseline architecture using a cycle-level simulator – GPGPU-Sim [1] and faithfully model all key parameters (Table 1). The energy measurements are gathered using GPUWattch [15].

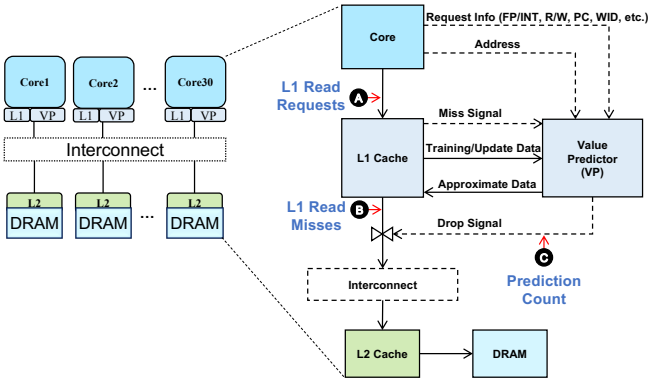


Figure 2: Baseline GPU Architecture with a value predictor.

Table 1: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [5] for the full list.

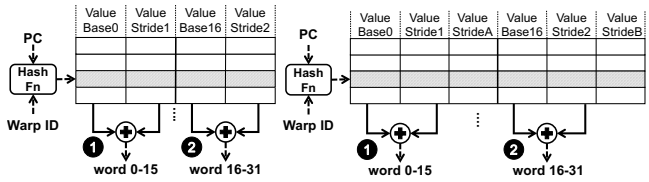
Core Features	1400MHz core clock, 30 SMs, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file Up to 1536 threads (48 warps, 32 threads/warp)
L1 Caches / Core	16KB 4-way L1 data cache 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (768KB in total) 128B cache block size
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [6], t_{CL} = 12, t_{RP} = 12, t_{RC} = 40, t_{RAS} = 28, t_{CCD} = 2, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12
Interconnect	1 crossbar/direction (30 SMs, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

Evaluation Metrics. We summarize the metrics evaluated in this paper with the help of Figure 2. *Coverage* is the ratio between *Prediction Count* C (i.e., cache lines that are predicted and not sent to the lower level) and *L1 Read Requests* A. Since the number of L1 Read Requests is constant for an application, the prediction accuracy across different predictors can be compared at the same coverage. *Miss Match Rate (MMR)* is the maximum achievable ratio between *Prediction Count* C and *L1 Read Misses* B. The prediction quality is measured in terms of *Application Error*, which is defined as the average relative error between the output of the approximate version and the baseline accurate version of an application.

2.2 Baseline Value Predictors

Figure 2 (right side) shows the general structure of the value predictor and its operation. The value prediction works concurrently with the cache access. We rely on user-supplied annotations to identify approximable instructions (more details are in Section 4.4). When a load request is issued from the core, its information (e.g., address, program counter (PC), Warp ID (WID), a bit to indicate floating-point vs. integer value (FP/INT), user-supplied annotation, memory space) is passed to the predictor. If the cache access results in a miss, a miss signal is generated to inform the value predictor. If the value predictor is able to predict the associated cache line, it will: a) issue a drop signal to inform the MSHR to not send the cache request to the lower level of the hierarchy, and b) fill the L1 cache with the predicted data. Whenever the L1 cache is filled with a request fetched from the lower level of memory, it will be sent to the value predictor for training and update (see Section 4.1).

Our baseline value predictor is based on rollback free value predictor (RFVP) for GPUs [45]. RFVP takes advantage of the prediction tables implemented in the hardware to track the patterns in the data values. Specifically, RFVP uses a hash of Warp ID and PC to map different requests to particular entries in the prediction table. The prediction is performed at a granularity of the memory access size (typically 4 bytes) of a thread. However, as the prediction of all the words in a cache line is desired, the observation of intra-warp value similarity is used to predict values within the cache line. Our baseline predictor has two sub-predictors [45]. The first sub-predictor is responsible for predicting the first word, which is then copied to the first half of words (words 1 to 15). Similarly, the second sub-predictor is used for the second half (words 17 to 31). The following discussion provides the necessary background on two different RFVP-style baseline predictors.



(a) One Stride Predictor (OSP) **(b) Two Stride Predictor (TSP)**

Figure 3: Design of the baseline value predictors.

RFVP-Style One Stride Value Predictor (OSP). Figure 3(a) shows the design of OSP, which uses a hash function [45] based on PC and Warp ID to map requests to entries in the prediction table. Each cache line is predicted with two one-stride sub-predictors. The predicted value of word 0 (which is later copied to words 1 through 15) is the sum of ValueBase0 and ValueStride1 (❶). Similarly, the predicted value of word 16 (which is later copied to words 17 through 31) is the sum of ValueBase16 and ValueStride2 (❷).

Before prediction, both sub-predictors need to be trained. For the training, at least two successive cache lines are needed from the main memory. ValueBase0 is updated by the word 0 of the second cache line and ValueStride1 is updated to the difference between the word 0 of the second cache line and the first cache line. The same process is repeated for ValueBase16 and ValueStride2 of the second sub-predictor with word16 (instead of word0) of the two successive cache lines. To control the accuracy, data is periodically fetched from the main memory to update the base and stride values.

RFVP-Style Two Stride Value Predictor (TSP). Figure 3(b) shows the design of TSP, which uses the same hash function as OSP. However, the cache line is predicted with the help of two two-stride sub-predictors. The prediction process of TSP is similar to OSP. The predicted value of word 0 (which is later copied to words 1 through 15) is the sum of ValueBase0 and ValueStride1 (❶), and the predicted value of word 16 (which is later copied to words 17 through 31) is the sum of ValueBase16 and ValueStride2 (❷). The training process of TSP is different from OSP only regarding how the stride is calculated. For the training, at least three cache lines are needed from the memory. With three successive cache lines, ValueBase0 is updated by the word 0 of the third cache line, and the ValueStrideA is updated to the difference between the word 0 of the third and the second cache line. ValueStride1 is updated to the value of ValueStrideA only if ValueStrideA also equals the difference between the word 0 of the second and the first cache line, otherwise, the sub-predictor is considered to be not trained. The second sub-predictor adopts the same process for the values of ValueBase16 and ValueStride2. The training process stops when both ValueStride1 and ValueStride2 are found. Again, the accuracy can be controlled by periodically fetching data from the memory.

3 MOTIVATION AND ANALYSIS

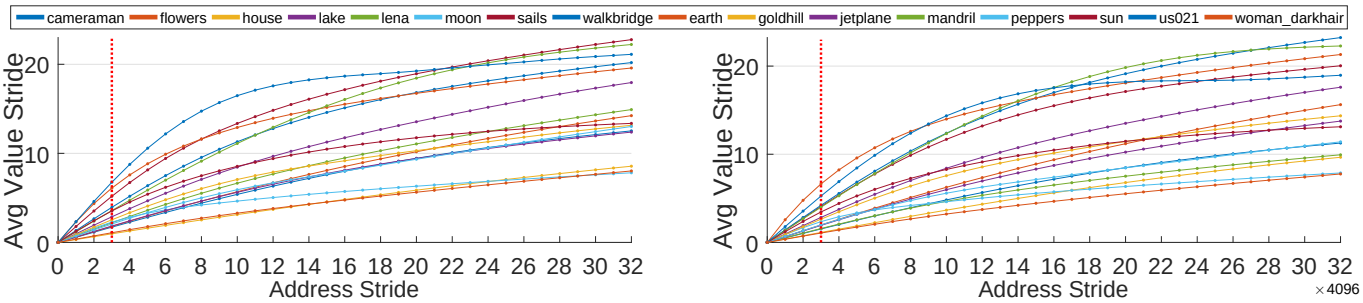
In this section, we first analyze the relationship between address and value strides in the inputs of GPGPU applications, followed by a discussion on how this relationship helps in improving the accuracy of the value prediction.

3.1 Analysis of Address and Value Strides

We find that a wide range of GPGPU workloads work on inputs that have regular values strides (also discussed in Section 1). In general, we observe that a large number of nearby pixels in the image are similar or have gradually changing grayscales leading to regular value strides. To validate this observation, we picked a series of images including the commonly used standard test images to analyze the correlation between their address strides and value strides. Figure 4 shows the average absolute value strides with increasing address strides for all pixels in each image. For example, for the address stride of 1, the corresponding average absolute value stride is the average absolute value difference between every two pixels with consecutive addresses. The unit of the address stride is the size of the data type used. Specifically, Figure 4(a) shows how the average value stride changes along the row of the image. Meanwhile, Figure 4(b) shows how the average value stride changes along the column of the image. As we can observe from the two figures, different images show different extent of linear correlations. Overall, the smaller the address stride, the more linear the correlation. Specifically, for the value strides of the next three nearby pixels (i.e., on the left of the red dashed line) both row-wise and column-wise, all images show nearly constant slopes between their address strides and value strides.

3.2 Motivation

We find that the observation of linear correlation between address stride and value stride can help to improve the accuracy of value prediction in GPUs. Consider an illustrative example shown in Figure 5. Assume that three cache line requests are generated from three different warps with addresses 0,1,2 and values 0,2,4, respectively (Ⓐ). Therefore, the address stride and value stride are linearly



(a) Addresses are incremented row-wise. **(b) Addresses are incremented column-wise.**

Figure 4: Illustrating the relationship between average value stride of data with different address strides for a variety of inputs.

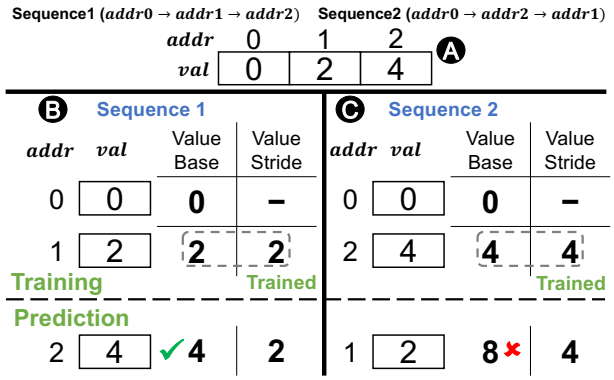


Figure 5: Illustrative example showing the importance of request order on value strides and the ease of value predictability.

correlated. However, these requests can be generated in different orders based on the warp scheduling policy in a GPU. Consider two possibles address sequences: Sequence I (0,1,2) (Ⓑ) and Sequence II (0,2,1) (Ⓒ), as shown in Figure 5.

In the first sequence, OSP is trained with the first two accesses. It can accurately predict the third value for Sequence I because the predicted value stride conforms with the actual stride. However, if the predictor is trained with Sequence II, a large relative error is detected because the calculated stride is incorrect. Hence, if a new value predictor is able to take advantage of the address and value stride correlation, then it is able to generate an approximation with better quality for images with similar attributes as shown in Figure 4. Meanwhile, the same data movement reduction and performance improvements are also achieved (Section 6).

To confirm this intuition, we analyzed a variety of real GPGPU applications[1]. Our profiling analysis examines the value strides by calculating the average stride difference between every two *consecutive* observed value strides. For example, if the values of three consecutive loads are V1, V2, V3, we examine the difference between (V2-V1) and (V3-V2). A smaller stride difference means that strides are more regular and hence it is easier to predict the values of future loads. As the value of stride difference is dependent on the load access order, we measure it on a simulated baseline GPU architecture (Section 5) under three scenarios. Note that we use the first 4B of cache lines accessed by load instructions to determine value strides. First, the stride difference is calculated from the loads belonging to the same PC and are generated as determined by the baseline GTO warp scheduler. Such a scenario mimics a PC-based value predictor that only considers value patterns of loads that have the same PCs (i.e., PC-Based). Second, the stride difference is calculated from loads that do not necessarily belong to the same PC but their addresses have regular strides (i.e., Address-Stride-Based). Third, as indicated in Figure 4 that nearby data tend to show stronger address and value stride correlation, we use the same design as in the second scenario but restrict the address stride to accept the closest data only for each application depending on their inputs (i.e., Address-Stride-Based-Restricted). The selection process of the restricted address stride is described in Section 5.2.

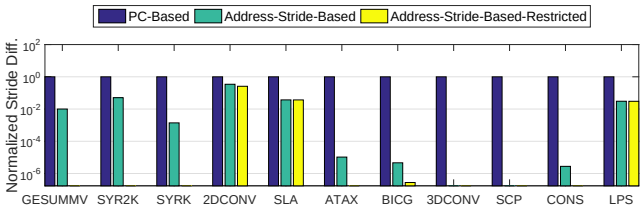


Figure 6: Normalized Stride Difference (in log scale) between *consecutively* observed value strides. Considering address-stride-based (2nd and 3rd bar) improves the value predictability over traditional PC-based approach (1st bar).

Figure 6 shows the normalized results for these scenarios: PC-based, Address-Stride-Based and Address-Stride-Based-Restricted, respectively. We observe that in the second scenario where the address strides are considered, the average stride difference is much lower. This indicates that consideration of memory addresses with regular strides can facilitate detecting regularities among value strides. In the third scenario, the average stride difference is even lower, as the average stride difference of many applications even reach 0. This confirms our observation in Figure 4. However, this scenario requires the user to specify an acceptable address stride. Considering that the Address-Stride-Based scenario already provides good improvements, we propose Address-Stride Assisted Value Predictor (ASAP) with the default mode and also evaluate the restricted mode for comparison purposes.

4 DESIGN AND OPERATION

In this section, we describe the design and operation of ASAP via answering the following these high-level questions: 1) How do we recognize the patterns in the memory address stream and leverage them for improving the accuracy of value prediction? 2) How do we handle irregular memory access orders? and 3) How do we ensure the design of the value predictor to be area-efficient?

[1]More details on the application characteristics/inputs and evaluation methodology are discussed in Section 5.

4.1 Design of ASAP

Overview. The Address-Stride Assisted Value Predictor (ASAP) provides two modes: default mode and restricted mode. Both of them have the same operations and working sequence except that the restricted mode only accepts user-defined address strides. For this reason, we do not differentiate them when introducing the design of ASAP. Figure 7 shows the overall design of ASAP that is built upon the baseline predictor as described earlier in Section 2.2. There are two major changes associated with ASAP. First, ASAP does *not* rely on the PC or Warp ID based tags or hash functions but uses the address of the memory requests to map them to the prediction table entries. Second, each prediction table entry is appended with additional fields containing the information of AddressBase (i.e., Cache Block Index) and AddressStrides (❶) to facilitate in predicting the strides in the memory addresses. The key idea behind these changes is to identify and then leverage address patterns in the memory requests in order to facilitate value prediction. If the next address is predicted correctly, *only then* its corresponding value can be predicted. Essentially, we treat each entry of the prediction table as a holder for a certain kind of address pattern in the access stream. As we observe that the types of different address stride patterns in GPGPU applications are limited, we find that eight entries are sufficient (sensitivity studies are discussed in Section 7).

ASAP has two versions: ASAP-OSP and ASAP-TSP, based on the type of sub-predictor it employs. For brevity, we only discuss the design of ASAP-OSP (Figure 7) as it captures all the design issues of ASAP-TSP. We use the same entry to store either floating-point or integer data and use 1 bit (FP/INT bit) to differentiate between them. The FP bit also indicates whether floating-point adders or integer adders should be used.

The Prediction Process. In order to track various stride patterns in the memory access stream, we use two types of AddressStride fields: AddressStrideShort and AddressStrideLong. For tracking the strides in the value stream we use two types of ValueStride fields: ValueStrideShort and ValueStrideLong. If the incoming address equals to (i.e., the address matches) the sum of AddressBase and AddressStrideShort or AddressStrideLong (❷), then its value can be predicted. We define such a situation as a *match* (❸). For example, if an entry has AddressBase 2, AddressStrideLong 2, and AddressStrideShort 1, then the entry is able to match the next request with address 3 or 4. Once a match is detected and if the address is correctly predicted using AddressStrideShort or AddressStrideLong, then the value of word0 is predicted with the sum of ValueBase0 and ValueStrideShort1 or ValueStrideLong1 (❹), which is later copied to words 1 through 15. The value of word16 is predicted with the sum of ValueBase16 and ValueStrideShort2 or ValueStrideLong2 (❺), which is later copied to words 17 through 31. After each prediction, AddressBase is updated to the matched address. ValueBase0 and ValueBase16 are updated to the predicted values of word0 and word16, respectively.

The Training Process. Before prediction, an entry must be trained. Each entry is responsible for tracking different address stride patterns in the access stream and is trained with at least two memory requests. For a sequence of requests, AddressBase will be set to the last address that accessed the entry. AddressStrideShort will be set to the difference between the two most recent addresses. A third request is required to train AddressStrideLong as it is the sum of the most recent two AddressStrideShort values. Therefore, the training is based on the last three requests mapped to the entry.

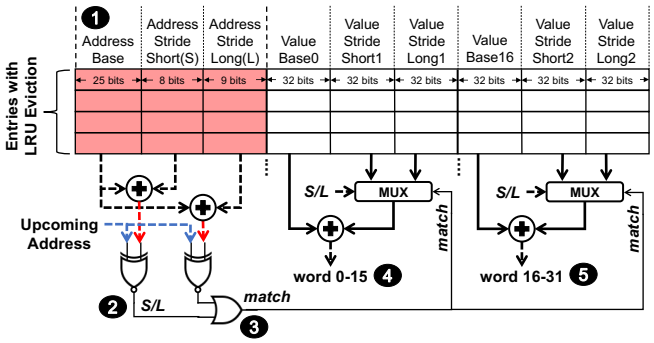


Figure 7: Design of the Address-Stride assisted value predictor.

For example, let us assume that addresses 1, 2, 4 will consecutively update an entry. After the 2nd address comes, AddressBase will be 2, AddressStrideShort will be 1 (2-1), and AddressStrideLong will remain unchanged. After the 3rd address comes, AddressBase will be 4, AddressStrideShort will be 2 (4-2), and AddressStrideLong will be 3 (1+2). During an entry's training phase, we also create and update a new entry with the 2nd and 3rd requests of that entry. This step is performed to warm-up new entries for faster matching. New entries are created based on least recently used policy.

The Matching Process. At the first match of an entry, it leaves its training phase (i.e., it is trained) and enters the prediction phase. The value of AddressStrideShort will be set to the chosen stride (Short or Long) and the value of AddressStrideLong will be set to twice the value of AddressStrideShort in order to capture missing intermediate addresses as we will discuss in Section 4.3. Similarly, the corresponding Value-Stride (ValueStrideShort or ValueStrideLong) will be assigned to ValueStrideShort, and the ValueStrideLong will be twice the value of ValueStrideShort. Subsequently, the values of AddressStrideShort and AddressStrideLong will remain fixed during the lifetime of the entry. Note that before an entry is trained, StrideLong is not necessarily equal to the twice of StrideShort, as mentioned in the training process.

The Updating Process. When an L1 Miss is not predicted, the fetched cache line is used to update the AddressBase, ValueBase, ValueStrideShort, and ValueStrideLong of the matched entries to increase the accuracy of future predictions, while the AddressStrideShort and AddressStrideLong remain unchanged. The ratio of prediction and update is controlled by the desired coverage that a user can specify. Hence, the predictor will predict only if the desired coverage has not been reached. Both the prediction and the update can only happen in a matched entry. To ensure the updates are evenly distributed, we predict and update in a fine-grained manner. For example, 50% coverage can be achieved by doing 5 consecutive predictions followed by 5 consecutive updates.

Note that there should be at least 2 consecutive updates for ASAP-OSP and 3 for ASAP-TSP in order to update the value strides in their corresponding sub-predictor (Section 2.2). When an update or a prediction request comes to the predictor, entries are checked one by one to see if there is a match in any of the entries. We find that this small extra latency does not affect the performance benefits obtained from dropping the request. The match for AddressStrideShort and AddressStrideLong inside each entry happens in parallel. If no match is found, we replace an old entry with a new entry based on LRU policy and start its training phase.

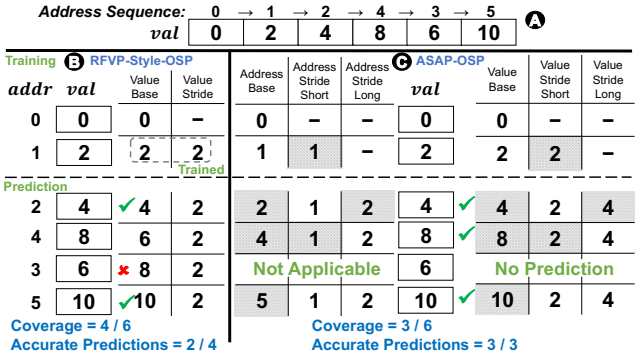


Figure 8: Operation of ASAP and its advantages over OSP. The matched addresses, predicted values, relevant strides are shaded.

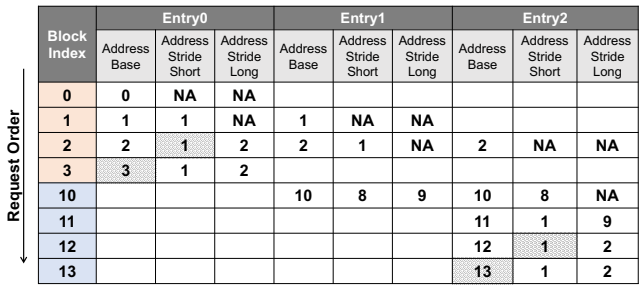


Figure 9: Working steps of ASAP in Scenario I: Regular Address Pattern. The address stream considered is: 0, 1, 2, 3, 10, 11, 12, 13. The matched addresses and relevant strides are shaded.

4.2 Operation of ASAP

Figure 8 illustrates the operation of ASAP-OSP and its advantages over RFVP-Style-OSP by considering a sequence of addresses (0,1,2,4,3,5). The corresponding data values are shown in boxes next to each other (A). In this example, we assume that the address sequence is generated from the same PC. When RFVP-Style-OSP is employed (B), the first two requests train the entry. After training, ValueBase is 2 and ValueStride is 2. The third request is correctly predicted because its value is the sum of ValueBase and ValueStride. However, the values of fourth and fifth memory requests are incorrectly predicted because the ValueStride of 2 does not correctly capture the value pattern. However, the value of the sixth request is correctly predicted as it matches with the sum because the ValueBase was still being updated even when the predictions were wrong. Overall, the coverage is 66% (4 out of 6 requests are predicted) and accuracy is 50% (2 out of 4 predictions are accurate).

In the case of ASAP-OSP (C), AddressBase, AddressStrideShort, and AddressStrideLong are responsible for detecting strides in the addresses. After the first two accesses, AddressBase and AddressStrideShort are trained in addition to ValueBase and ValueStrideShort. As the third address is the sum of AddressBase and AddressStrideShort, it implies that there is a match and its corresponding value can be predicted. We observe that ASAP-OSP can correctly predict its value as its sum is equal to ValueBase and ValueStrideShort. At this point, AddressStrideLong and ValueStrideLong are also set as equal to twice of AddressStrideShort and ValueStrideShort, respectively. The fourth address is also a correct match as the sum of AddressBase and AddressStrideLong matches with the predicted address. Therefore, its value can be predicted using the sum of ValueBase and ValueStrideLong, which is also correctly predicted. The fifth request is not predicted because its address does not match the pattern in the addresses (Addr 3 is neither equal to the sum of AddressBase and AddressStrideShort nor AddressBase and AddressStrideLong). Finally, the sixth request can be correctly predicted as its address pattern can be captured via AddressStrideShort. Overall, the coverage is 50% (3/6 requests are predicted) and accuracy is 100% (3/3 predictions are accurate). In summary, as opposed to the RFVP-Style-OSP, ASAP-OSP can take advantage of the readily available address information and improve the accuracy significantly by trading-off coverage.

4.3 Use Cases of ASAP

For the address sequences which are generated from the core, we find that there are two possible cases which can make them difficult to be captured. The first case is that in an address sequence with a particular stride some of the intermediate addresses are missing. The second case is that multiple address sequences are interleaved together leading to a complicated address sequence. Our ASAP design takes these two cases into consideration and we will also evaluate its effectiveness with real applications later in Section 7. To help understand how ASAP can capture different kinds of strides in the addresses, we present three scenarios.

Scenario I: Regular Address Pattern – Demonstrating the utility of multiple entries. Consider a scenario when consecutive address sequences: (0, 1, 2, 3) and (10, 11, 12, 13) are generated back to back. Figure 9 shows the values of AddressBase, AddressStrideShort, and AddressStrideLong for each entry. For brevity, we only show the first three entries that are relevant for this example. After the first two addresses are mapped to the first entry (Entry0), the remaining addresses of the sequence (2,3) are matched as they belong to the same stride pattern (AddressStrideShort of Entry0 is set to 1). Note that AddressStrideLong is set to twice of AddressStrideShort and the next Entry (Entry1) is also prepared in anticipation of other possible patterns in the addresses by setting the AddressBase to be 2 and AddressStrideShort to be 1.

When the second sequence of addresses arrive at the predictor, the first address (10) among them cannot be matched by Entry0 because neither the sum of AddressStrideShort or AddressStrideLong with AddressBase matches with the address. Therefore, it will be mapped to Entry1. AddressStrideShort is set to 8 (10-2) and AddressStrideLong becomes the sum of the previous two AddressStrideShort (8+1) for Entry1. After Entry1 is trained with 3 requests, it cannot match the next address 11, so again 11 is put into a new entry (Entry2). The Entry2 is trained with 2, 10, 11 and is able to match remaining addresses of the sequence (12,13).

Scenario II: Interleaved Address Pattern – Demonstrating the utility of AddressStrideLong. The interleaved address pattern may be caused by the interleaved execution of two warps, or by the poorly coalesced requests from certain load instructions. For example, one warp generates addresses (1, 2), another warp generates (4, 5), and so on. Figure 10 demonstrates such a sequence: (1, 2, 4, 5, 7, 8, 10, 11). For Entry0, it is trained with three addresses 1, 2, 4. Also, 2, 4 are copied to Entry1. For the next address 5, since Entry0 has reached its maximum training count of 3, it can only be put into Entry1 which still has 1 slot for training. At this point, both Entry0 and Entry1 have AddressStrideLong equal to 3. So when address 7 comes, it matches with AddressStrideLong in Entry0. The next address 8 also matches with AddressStrideLong in Entry1. Addresses 10, 11 can also be matched with Entry0 and Entry1, respectively.

Block Index	Entry0			Entry1		
	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long
1	1	NA	NA			
2	2	1	NA	2	NA	NA
4	4	2	3	4	2	NA
5				5	1	3
7	7	3	6			
8				8	3	6
10	10	3	6			
11				11	3	6

Request Order

Figure 10: Working steps of ASAP in Scenario II: Interleaving Address Pattern. The addresses considered are: 1, 2, 4, 5, 7, 8, 10, 11. The matched addresses and relevant strides are shaded.

Block Index	Entry0			Entry1		
	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long
0	0	NA	NA			
1	1	1	NA	1	NA	NA
2	2	1	2	2	1	NA
3	3	1	2			
5	5	1	2			

Block Index 4 is missing

Request Order

Figure 11: Working steps of ASAP in Scenario III: Missing Intermediate Address Pattern. The addresses considered are: 0, 1, 2, 3, 5. The matched addresses and relevant strides are shaded.

Scenario III: Missing Intermediate Address Pattern. We present an example of handling a missing intermediate address in a consecutive address sequence. The missing intermediate address pattern may be caused by the non-consecutive scheduling of warps or control divergence. Figure 11 demonstrates such case with a sequence: (0, 1, 2, 3, 5). After 0, 1, 2 are mapped to Entry0, it is trained with the value of AddressStrideShort to be 1 and AddressStrideLong to be 2. Hence, after address 3 is matched, it can match address 5 from the AddressBase 3 directly using AddressStrideLong. Without the AddressStrideLong, we would not have matched with address 5, thereby limiting the coverage.

4.4 Output Quality Control

The rollback-free value prediction eliminates pipeline rollbacks, which is prohibitively expensive in GPUs. However, it also introduces errors in GPU pipelines. These errors lead to different types of consequences if no restrictions are enforced to control them. First, the application may crash or lead to an unknown behavior if errors are generated for critical values. For example, an incorrect PC or address value will likely cause a fatal error. Second, the application's execution trace can become vastly different and produces unexpected results if errors are generated for values involved in conditional branching. For example, an incorrect counter in a *for* loop can produce unusual results in an application's output. Third, the application's output can lose a certain level of quality. For example, some mispredicted values in an input matrix may cause a certain level of distortion to an application's output. In this case, the level of quality loss depends on: a) the accuracy of the individual predictions, b) the number of values predicted (i.e., prediction coverage), and c) the future operations that will be applied to these predicted values.

ASAP guarantees that only limited output quality loss can happen by requiring the programmer's annotations of approximable load values and taking the input of prediction coverage values. The compiler is also slightly modified to accept these additional directives to facilitate value approximation. For example, as shown in Listing 1, 10% prediction coverage is specified by the fetch and predict ratio of 9 to 1. The programmer has also indicated to approximate the value of vector B in the following memory load operation. Therefore, as shown in Listing 2, the added directives will inform the predictor on two items: a) the amount of load requests to approximate (i.e., coverage), and b) which load instruction to approximate.

```
#pragma add_pred{fetch, 9, predict, 1}
...
#pragma approx{B}
C[i] = A[i] + B[i];
```

Listing 1: annotated CUDA code

```
.fetch 9
.predict 1
...
ld.global.u32.approx %r0, [%r1]
```

Listing 2: generated PTX code

ASAP uses prediction coverage to trade off output quality for performance. To satisfy a certain output quality threshold, ASAP can rely on the programmer to provide an appropriate prediction coverage so as to maximize the performance gains under this threshold. Previous works [2, 18, 25, 34] have indicated that the application error cannot be bounded automatically in the first kernel invocation as the error of approximation depends on the semantics of the application. However, a multi-invocation approach is still able to automatically find the optimal prediction coverage. Hence, ASAP can employ a searching method which is similar to the proposed approach of prior work [33] to find the highest prediction coverage for a given output quality requirement. As we will show in Section 6.1, at the same prediction coverage, ASAP can lead to less output quality loss than the state-of-the-art value predictor for GPUs. Reciprocally, ASAP is able to achieve higher performance improvements under the same output quality threshold.

4.5 Hardware Overhead

Figure 7 shows that ASAP-OSP uses 234 bits per entry. Additionally, it uses three fields (not shown) namely Status (5 bits), LRU (3 bits), and Floating-point (1 bit), making the overhead per entry 243 bits. Status bits are used to track the current status of the entry (e.g., training phase, predicting phase or update phase) in order to decide the entry's action for the next matched request. We have discussed the transitions between different statuses in Section 4.1. The LRU bits are used to track the LRU information. The Floating-point bit is used to differentiate the floating-point and integer data. Since ASAP uses eight entries per core, the total overhead is 243 bits ×8 = 1944 bits/Core. ASAP-TSP has four extra fields per entry. These fields are ValueStrideShortA, ValueStrideShortB, ValueStrideLongA, and ValueStrideLongB, thus, the total overhead is (243+32×4)×8 = 2968 bits/Core (0.36KB/Core). In addition to these bits, each core employs four integer adders, two floating-point adders, four 2×1 MUXes, two comparators, and one OR gate.

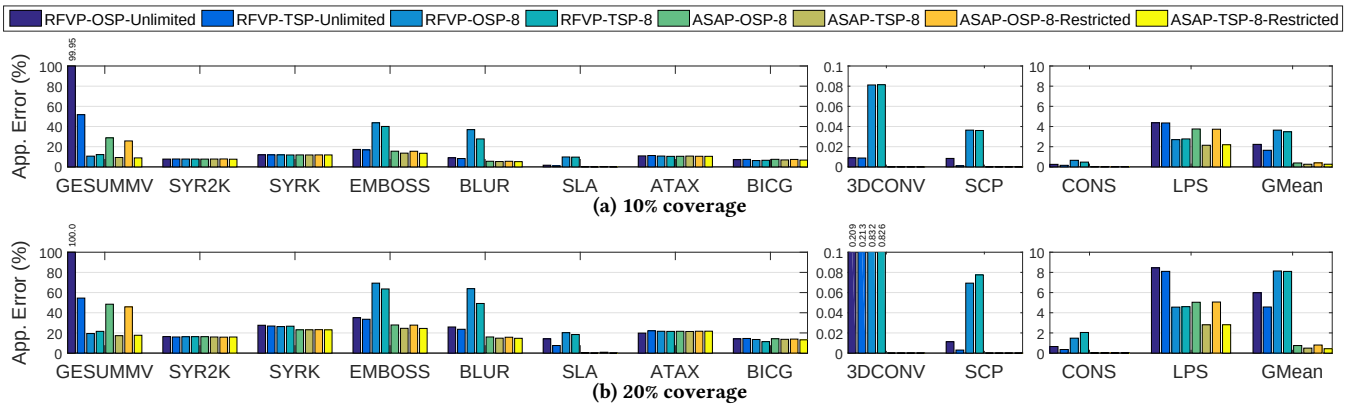


Figure 12: Application Error for different value predictors at (a) 10% (b) 20% coverage.

Table 2: List of evaluated GPGPU applications.

Abbr.	Input	Category	Coalescing	Int.
GESUMMV [29]	2048 × 2048 Matrix	BW-Bound	Good	No
SYR2K [29]	128 × 128 Matrix × 3	Latency-Bound	Good	No
SYRK [29]	256 × 256 Matrix × 2	Latency-Bound	Good	No
EMBOSS (2DCONV) [29]	4096 × 4096 Image	BW-Bound	Poor	Yes
BLUR (2DCONV) [29]	4096 × 4096 Image	BW-Bound	Poor	Yes
ATAX [29]	4096 × 4096 Matrix	Latency-Bound	Good	No
BICG [29]	3072 × 3072 Matrix	Latency-Bound	Good	No
3DCONV [29]	256 × 256 × 256 Matrix	BW-Bound	Poor	Yes
SLA [1]	Size 24000000 Array	Energy-Bound	Good	No
LPS [1]	256 × 256 × 256 Matrix	BW-Bound	poor	Yes
SCP [1]	Vector × 16384	BW-Bound	Good	No
CONS [1]	8192 × 8192 Matrix	Energy-Bound	Good	Yes

5 EVALUATION METHODOLOGY

5.1 Application Characteristics

We consider a variety of GPGPU applications from Polybench [29] and CUDA SDK [1] as shown in Table 2. We chose them as they show diversity in terms of memory intensity and coalescing behavior. Also, these applications use matrix or vector inputs with strided values provided by their corresponding benchmark suites and they can accept realistic images as their inputs. We use annotations to mark the approximable loads. We ensure that they do not contain pointers or lead to fatal errors, and thus can be approximated safely. The programmer can also tune the aggressiveness of value approximation by adjusting the prediction coverage [20, 35, 45] or using only restricted address strides (Section 5.2). If there are drastic variations in value strides for all given address strides, the programmer can choose to turn off the value predictor.

We classify applications into multiple categories. The **BW-Bound** applications have high DRAM bandwidth utilization (at least 40%) and relatively low IPC (at most 500). The **Latency-Bound** applications have low IPC (less than 100) and low bandwidth utilization (less than 10%). For both these classes, we expect value prediction would provide performance and data movement reduction benefits. The **Energy-Bound** applications have high IPC (more than 500) and significant off-chip traffic. For such applications, we expect value prediction would provide data movement reduction benefits but not necessarily performance benefits. Finally, we also considered coalescing conditions. The loads of `EMBOSS`, `BLUR`, `3DCONV`, and `LPS` are poorly-coalesced (i.e., two or more cache line requests are generated per load instruction per warp.). Other applications have good coalescing characteristics. Our applications use integer or floating point data. The last column of Table 2 shows whether the data type is integer (Int.) or floating point.

5.2 Choice of the Restricted Address Strides

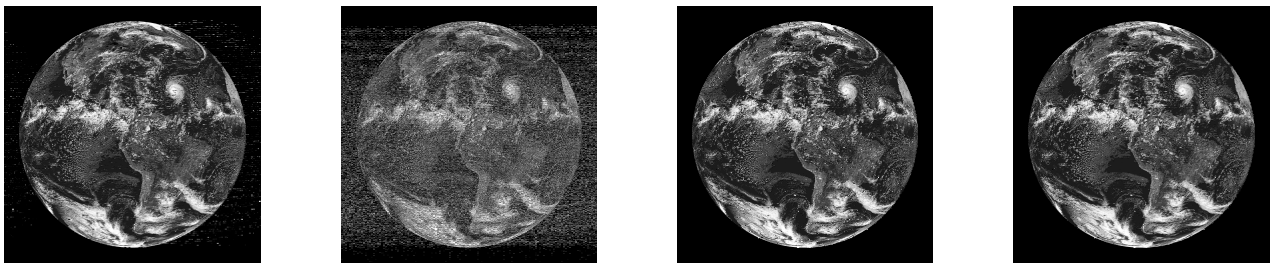
For the restricted mode of ASAP, we manually set the acceptable address strides. As we have discussed in Section 3, we would prefer address strides that correspond to closer data on the same row or column of the input. However, as cache lines in GPUs typically contain 128 consecutive bytes, the address stride need to be set to at least 128 in order to predict cache lines in the same row. Therefore, the address stride of $\pm row\_size$ of inputs is used for all applications, which corresponds to the closest cache lines in the same column. Also, other address strides are used if they show linear correlations with their corresponding value strides.

6 EXPERIMENTAL EVALUATION

We compare the proposed ASAP design with the prior RFVP-Style predictors adopting both OSP and TSP sub-predictors (Section 2.2). For a fair comparison, we use the same number of entries (i.e., 8) across all predictors. They are RFVP-OSP-8, RFVP-TSP-8, ASAP-OSP-8, ASAP-TSP-8, ASAP-OSP-8-Restricted and ASAP-TSP-8-Restricted. We also compare ASAP design with the oracle implementations of RFVP-Style-OSP and RFVP-Style-TSP, namely RFVP-OSP-Unlimited and RFVP-TSP-Unlimited, respectively. These oracle implementations assume unlimited hardware budget for the prediction table entries. Hence, the loads from each PC and Warp ID combination can use a separate entry such that predictions from different PCs and warps do not affect each other.

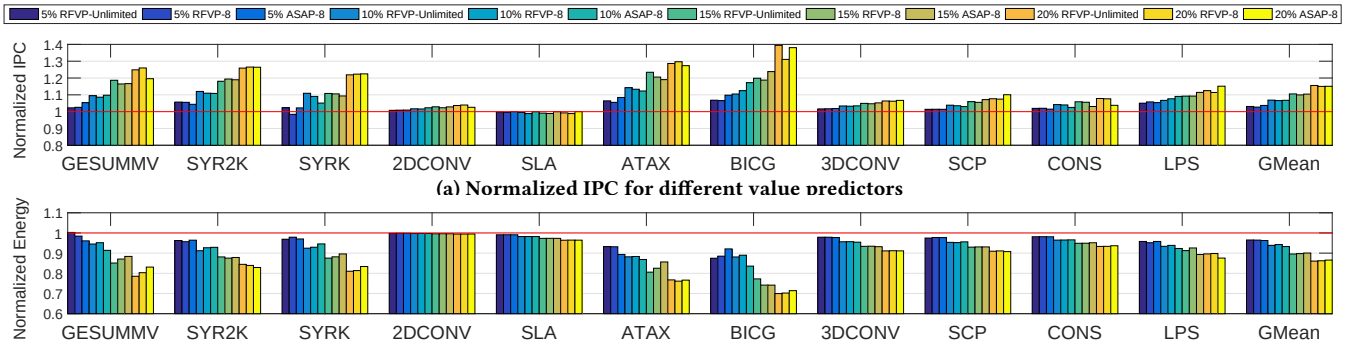
6.1 Effect on Output Quality

Figure 12 shows the comparison of Application Error of 10% and 20% coverage. We limit the predictors' coverage to be under 20% to restrict the error. However, if the user has knowledge of the application's good predictability (e.g., `SCP`), coverage can be set higher for more performance and data movement reduction benefits. We make the following observations. First, we find that at the same coverage, our proposed predictors have better accuracy than the previous predictors, even when the number of entries for ASAP is much less than RFVP. On average, RFVP-Unlimited predictors predict more accurately than RFVP-8 predictors, and ASAP-8 predictors are more accurate than RFVP-Unlimited predictors. Also, for each category of predictors, the TSP predictors are more accurate than the OSP predictors, showing the effectiveness of predicting with more regular strides. At 10% coverage, both ASAP-TSP-8 and ASAP-TSP-8-Restricted reduce Application Error by 92% over RFVP-TSP-8 and 84% over RFVP-TSP-Unlimited. At 20% coverage, ASAP-TSP-8



(a) RFVP-TSP-Unlimited (b) RFVP-TSP-8 (c) ASAP-TSP-8 (d) ASAP-TSP-8-Restricted

Figure 13: `EMBOSS(2DCONV)` **outputs at 10% coverage.**



(a) Normalized IPC for different value predictors

(b) Normalized total energy consumption for different value predictors

Figure 14: GPU performance and total energy consumption at different coverages.

reduces Application Error by 94% over RFVP-TSP-8 and 89% over RFVP-TSP-Unlimited. ASAP-TSP-8-Restricted reduces Application Error by 95% over RFVP-TSP-8 and 91% over RFVP-TSP-Unlimited.

Second, we find that for certain applications, (i.e., `EMBOSS`, `BLUR`, `SLA`, `3DCONV`, `SCP`, `CONS`, `LPS`), the increase in application error with increasing coverage is at a much slower rate in ASAP compared to that in other predictors. This implies that ASAP is able to better exploit the relationship between address and value strides to improve the accuracy even at higher coverages. For applications `SYR2K`, `SYRK`, `ATAX`, `BICG`, the accuracy benefits of ASAP and RVFP are comparable, implying no obvious address and value stride correlations exist in them. There are cases where RFVP-8 predictors have better accuracy than the RFVP-Unlimited predictors (i.e., `GESUMMV`, `LPS`). This indicates that not sharing the prediction table entries across warps degrades the prediction accuracy in some cases. However, ASAP-TSP-8 and ASAP-TSP-8-Restricted still have better accuracy in this case, because they capture more stable value strides according to the address pattern observed across different warps and PCs.

For the image output quality, we pick `EMBOSS(2DCONV)` at 10% coverage to study the difference between predictors. As shown in Figures 13(a) to (d), there is significantly less noise in ASAP-TSP-8 and ASAP-TSP-8-Restricted than in RFVP-TSP-8. Further, there is slightly less noise in ASAP-TSP-8 and ASAP-TSP-8-Restricted than in RFVP-TSP-Unlimited. This trend matches the Application Error result. For these outputs, ASAP-TSP-8 shows 13.6% Application Error and ASAP-TSP-8-Restricted shows 13.5%. We find that these errors do not cause significant quality losses.

We conclude that by leveraging the address and value stride correlation, (regardless of the PC and Warp ID information), ASAP can effectively improve the prediction accuracy over the RFVP-Style predictors with a similar or lower area budget. Without extra burden on the user, ASAP's default mode can provide accuracy close to that of the restricted mode. Meanwhile, the restricted mode can further increase the accuracy with user-specified address strides.

6.2 Effect on Performance and Energy

Figure 14 shows the performance and energy benefits of applying value prediction. For brevity, we show results of RFVP-TSP-Unlimited, RFVP-TSP-8, and ASAP-TSP-8 from each of the three predictor categories. We also confirm that other predictors show similar trends. Since the RFVP and ASAP predictors predict similar numbers of cache lines at the same coverage, they provide similar IPC and energy benefits. However, ASAP produces smaller errors. On average, ASAP-TSP-8 improves IPC by 7% at 10% coverage and improves IPC by 15% at 20% coverage. Specifically, for the Latency-Bound applications, we observe an average IPC improvement of 11% at 10% coverage and an average IPC improvement of 29% at 20% coverage. On the other hand, ASAP-TSP-8 reduces GPU energy consumption by 7% at 10% coverage and reduces GPU energy consumption by 14% at 20% coverage. We conclude that the prediction coverage is the dominant factor of performance and energy benefits in value approximation. Value approximation can effectively improve performance and reduce energy consumption. Also, these benefits grow larger when the prediction coverage increases.

On average, the best performing ASAP predictor produces only 0.26% (up to 13.51%) Application Error at 10% coverage and 0.43% (up to 24.47%) Application Error at 20% coverage. In contrast, with the same number of entries, the best performing RFVP predictor incurs 3.48% (up to 40.08%) Application Error at 10% coverage and 4.57% (up to 54.54%) Application Error at 20% coverage. Even when using different entries for each PC and Warp ID combination, RFVP incurs 1.65% (up to 51.74%) Application Error at 10% coverage and 8.10% (up to 63.59%) Application Error at 20% coverage, which is much higher than that of ASAP with 8 entries. This means that ASAP can employ higher coverages and consequently obtain more performance and energy benefits if a certain error threshold needs to be satisfied. We conclude that ASAP can achieve more performance and energy benefits under a specific error threshold.

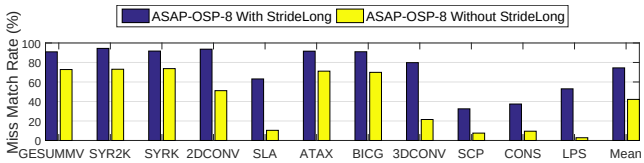


Figure 15: Effect of AddressStrideLong on Miss Match Rate.

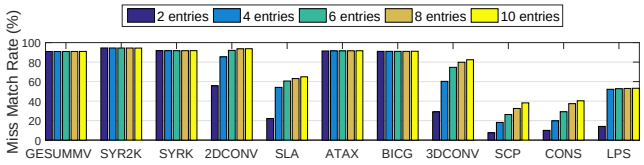


Figure 16: Miss Match Rate with different entry numbers.

7 SENSITIVITY STUDIES

Effect of AddressStrideLong. The Miss Match Rate (MMR) reflects how effective ASAP is able to capture the address patterns in GPUs. It is important for ASAP to achieve an acceptable MMR, as it determines the maximum coverage of ASAP. For example, the MMR needs to reach at least 20% for applications with 100% L1 Miss Rate, if the desired coverage is 20%. To prove the effectiveness of AddressStrideLong, we show the MMR of ASAP-OSP-8 with and without it. As shown in Figure 15, we find that for poorly-coalesced applications (i.e., `EMBOSS`, `BLUR`, `3DCONV`, `LPS`), ASAP-OSP-8 has much higher MMRs with StrideLong. We also find that for well-coalesced applications, the MMR can become low if StrideLong is not employed (i.e., `SLA`, `SCP`, `CONS`). This limits their maximum coverage, data movement reduction, and performance benefits. This effectively shows ASAP's ability to match for interleaving and missing intermediate address patterns with the help of AddressStrideLong (see Section 4). Other ASAP predictors also show the same trend. We conclude that ASAP's address matching ability under complex patterns can be significantly improved with AddressStrideLong.

Effect of Number of Entries. Figure 16 shows the MMR of ASAP-OSP-8 with different numbers of entries. We make two observations. First, applications `SLA`, `3DCONV`, `SCP`, `CONS` need more entries to achieve high MMR as there are more co-existing address patterns. Others can reach high MMR even with 2 entries, which indicates that they have fewer co-existing address patterns. Second, after size 8, all applications' MMRs do not improve significantly and therefore we use it as ASAP's default configuration. Other ASAP predictors also show the same trend. We conclude that ASAP achieves good MMR and accuracy without incurring large hardware overhead.

Effect of Warp Scheduling Policy. The choice of warp scheduler can affect the warp execution order, thereby affecting the address patterns [9]. Figure 17 shows the MMR of ASAP-OSP-8 working under: The baseline, Greedy-Then-Oldest (GTO) scheduler, and Round-Robin (RR) scheduler. We make two major observations. First, ASAP-OSP-8 has high MMR for both schedulers proving that it is adaptive for different warp schedulers. Second, ASAP-OSP-8 usually achieves higher MMR with the RR scheduler. This is because the address order is more regular under the RR scheduler [9].

8 RELATED WORK

Previously proposed RFVP for GPUs [45] relies significantly on the program counter (PC) to detect value patterns in the memory requests. Such PC-based mapping mechanism implicitly assumes that the memory requests originated from that particular PC are ordered

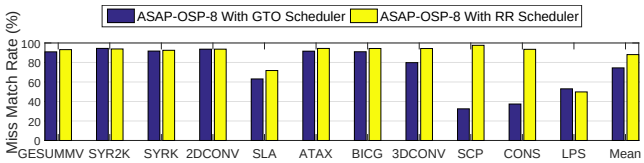


Figure 17: Miss Match Rate with GTO and RR Scheduler.

such that they would facilitate the prediction of values. However, as multiple warps can execute the same instruction (i.e., using the same PC) independently at different times in GPUs, the memory request order from a particular PC can be highly influenced by factors such as the choice of warp scheduling scheme. Therefore, as we show quantitatively in Section 6, the PC-based predictors cause a significant loss of accuracy in GPUs. This problem can be partially addressed by using a separate entry for each PC and Warp ID combination. However, such a mechanism can become prohibitively expensive as the number of concurrent warps and schedulers grows with each new generation of GPUs [7, 14, 19, 23, 24]. Moreover, using separate entries also disallow the detection of value patterns that might exist across requests from different warps (e.g., when nearby pixels of an image with regular value strides are handled by nearby warps). Wong et al. [44] proposed to exploit the intra-warp value similarity such that only one representative thread within a warp is required to perform the computation. Value approximation techniques [16] are proposed to reduce GPU energy consumption by carefully considering lower precision data/instructions. We believe ASAP is complementary to them as it eliminates the need for accessing the main memory for the predicted cache lines.

Several value prediction techniques [3, 4, 17, 26–28, 38, 39] in the context of CPUs are based on PC-based hash mechanisms, which have similar limitations as that of RVFP described earlier. Load-value approximation techniques [20, 35, 36] and context-based value predictors [21, 37, 40] designed for CPUs consider memory addresses and other metadata for effective approximations. However, such techniques require significant *per-thread* hardware resources, which can become prohibitively expensive in GPUs as it concurrently executes thousands of threads.

9 CONCLUSIONS

In this paper, we presented a low-overhead value predictor for GPUs that considers the correlation between address strides and value strides in order to improve the prediction accuracy. Compared to the state-of-the-art value predictor, RFVP, we find that our predictor can significantly improve value prediction accuracy even at a high value of prediction coverage (leading to significant performance and data movement benefits). We also show it is also able to function effectively even under complicated address patterns. We believe that this paper can open up interesting research avenues that consider other readily available information locally at the core (e.g., address stride information) to improve the accuracy of value prediction.

ACKNOWLEDGMENTS

We thank anonymous reviewers and Ulya Karpuzcu for their detailed feedback. This material is based upon work supported by the National Science Foundation grants (#1657336, #1717532, and #1750667). This work was performed using computing facilities at the College of William & Mary. Sparsh Mittal was supported by Science and Engineering Research Board (SERB) award (#ECR/2017/000622).

REFERENCES

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[2] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 33–52, 2013.

[3] R. J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM Journal of Research and Development*, vol. 37, no. 4, pp. 547–564, 1993.

[4] F. Gabbay, "Speculative Execution Based on Value Prediction," Technion - Israel Institute of Technology, Tech. Rep. 1080, 1996.

[5] GPGPU-Sim v3.2.1. (2014) GTX 480 Configuration. [Online]. Available: https://dev.ece.ubc.ca/projects/gpgpu-sim/browser/v3.x/configs/GTX480

[6] Hynix. (2009) Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. [Online]. Available: http://0x04.net/~mwk/ram/H5GQ1H24AFR%28Rev1.0%29.pdf

[7] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *GPGPU*, 2014.

[8] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[9] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[10] G. Kadam, D. Zhang, and A. Jog, "RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques," in *HPCA*, 2018.

[11] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[12] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," in *MICRO*, 2011.

[13] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An Online Quality Management System for Approximate Computing," in *ISCA*, 2015.

[14] J. Lee, M. Samadi, and S. A. Mahlke, "Orchestrating Multiple Data-Parallel Kernels on Multiple Devices," in *PACT*, 2015.

[15] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[16] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal, "SFU-Driven Transparent Approximation Acceleration on GPUs," in *ICS*, 2016.

[17] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," in *MICRO*, 1996.

[18] D. Mahajan, K. Ramkrishnan, R. Jariwala, A. Yazdanbakhsh, J. Park, B. Thwaites, A. Nagendrakumar, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Abstractions for Approximate Hardware Design and Reuse," in *MICRO*, 2015.

[19] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *ASPLOS*, 2014.

[20] J. S. Miguel, M. Badr, and E. N. Jerger, "Load Value Approximation," in *MICRO*, 2014.

[21] T. Nakra, R. Gupta, and M. L. Soffa, "Global Context-Based Value Prediction," in *HPCA*, 1999.

[22] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[23] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *ASPLOS*, 2013.

[24] J. J. K. Park, Y. Park, and S. A. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *ASPLOS*, 2015.

[25] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language Support for Safe and Modular Approximate Programming," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[26] A. Perais and A. Seznec, "Practical Data Value Speculation for Future High-End Processors," in *HPCA*, 2014.

[27] Perais, Arthur and Seznec, André, "EOLE: Paving the Way for an Effective Implementation of Value Prediction," in *ISCA*, 2014.

[28] Perais, Arthur and Seznec, André, "BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction," in *HPCA*, 2015.

[29] L.-N. Pouchet, "Polybench: the Polyhedral Benchmark Suite," in *URL: http://www.cs.ucla.edu/~pouchet/software/polybench/*, 2012.

[30] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.

[31] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.

[32] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[33] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-Tuning Approximation for Graphics Engines," in *MICRO*, 2013.

[34] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 164–174, 2011.

[35] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel, "The Bunker Cache for Spatio-Value Approximation," in *MICRO*, 2016.

[36] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger, "Doppelganger: A Cache for Approximate Computing," in *MICRO*, 2015.

[37] Y. Sazeides and J. E. Smith, "Implementations of Context Based Value Predictors," Technical Report ECE-97-8, University of Wisconsin-Madison, Tech. Rep., 1997.

[38] Sazeides, Yiannakis and Smith, James E, "The Predictability of Data Values," in *MICRO*, 1997.

[39] Sazeides, Yiannakis and Smith, James E, "Modeling Program Predictability," in *ISCA*, 1998.

[40] R. Thomas and M. Franklin, "Using Dataflow Based Context for Accurate Value Prediction," in *PACT*, 2001.

[41] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a Systematic Framework for Instruction-Level Approximate Computing and Its Application to Hardware Resiliency," in *MICRO*, 2016.

[42] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, O. Mutlu, C. Das, M. T. Kandemir, T. Mowry, and R. Ausavarungnirun, "Enabling Efficient Data Compression in GPUs," in *ISCA*, 2015.

[43] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multiprogramming in GPUs via Effective Bandwidth Management," in *HPCA*, 2018.

[44] D. Wong, N. S. Kim, and M. Annavaram, "Approximating Warps with Intra-Warp Operand Value Similarity," in *HPCA*, 2016.

[45] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-free Value Prediction with Safe-to-Approximate Loads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 62, 2016.