Multidimensional Grid Aware Address Prediction for GPGPU

Shivani Tripathy*, Debiprasanna Sahoo† and Manoranjan Satpathy‡
Indian Institute of Technology Bhubaneswar
Email: *st15@iitbbs.ac.in, †ds12@iitbbs.ac.in, ‡manoranjan@iitbbs.ac.in

***Abstract*—GPGPUs are predominantly being used as accelerators for general purpose data parallel applications. Most GPU applications are likely to exhibit regular memory access patterns. It has been observed that warps within a thread block show striding behaviour in their memory accesses corresponding to the same load instruction. However, determination of this inter warp stride at thread block boundaries is not trivial. We observed that thread blocks along different dimensions have different stride values. Leveraging this observation, we characterize the relationship between memory address references of warps from different thread blocks. Based on this relationship, we propose a multidimensional grid aware address predictor that takes the advantage of SM level concurrency to correctly predict the memory address references for future thread blocks well in advance. Our technique provides a cooperative approach where information once learned is shared with all the SMs. When compared with the CTA-aware technique, our predictor enhances average prediction coverage by 36% while showing almost similar prediction accuracy.**

.

I. Introduction

GPUs are now increasingly being used to accelerate general purpose data parallel and throughput intensive applications. The advent of CUDA [1] and OpenCL [2] have made programming of general purpose applications in GPUs easier.

Data parallel applications that can be executed on GPUs have two components: host-specific code (CPU code) and kernel code (GPU code). When a kernel is invoked, a grid of thread blocks is launched to execute on the streaming multiprocessors (SMs) of GPU. The dimension of this grid ranges from 1 to 3. Thread blocks in the grid consists of multiple warps; warp is a group of 32 threads. Figure-1 depicts a kernel having a two dimensional grid and thread blocks. Each thread block can be uniquely identified within a grid. Similarly, each thread can be uniquely identified within a thread block. These identifiers are used to determine the index of the array that a thread is going to access for a memory load [1], [3]. It has been observed that memory address references of warps within a thread block show a constant inter-warp stride [4], [5], [6], [7]. The memory address references for warps across the boundary of two consecutive thread blocks should also maintain this stride behaviour. However, these consecutive thread blocks are assigned to GPU SMs in a round robin manner to achieve maximum parallelism. Hence, it is not trivial to determine the stride behavior across thread block boundary within an SM. Furthermore, application data is distributed among the thread blocks arranged in multiple

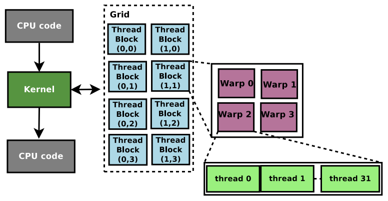


Fig. 1: CUDA programming model

dimensions. Inter thread block stride may appear irregular because of the way indices of the accessed data array are computed.

Analyzing the relationship between load address references can help in optimal architectural modifications for performance improvement. Some of the possible optimizations can be adaptive scheduling of thread blocks and warps, prefetching, change in cache replacement policy, efficient register file utilization etc. Jeon et al. [4] have addressed the problem of predicting the memory address references of warps from different thread blocks. In this work, the authors have proposed a warp scheduling policy that schedules a warp from each thread block early to determine the base address for each thread block. Even though this approach provides good prediction accuracy, it requires at least one warp of the thread block to be scheduled before prediction for other warps of that thread block can be done. This limits the applicability of prediction for further optimizations. Additionally, prediction can be incorrect for iterated loads when different warps of the thread block are at different stages of progress. Furthermore, as all the threads execute the same source code, once a pattern is detected for a particular load instruction, it can be used by all the threads running on different SMs. Hence, a cooperative scheme to allow sharing of learned pattern among SMs should be explored instead of relearning the pattern within the SM.

After analyzing the memory address references from different warps for several GPU benchmarks, we observed that memory accesses show striding behaviour across warps. Furthermore, the stride value among warps of different thread blocks is different along different grid dimensions. This creates the illusion that thread blocks have irregular strides among them. However, if we consider the change in thread block id along different dimensions and strides along these dimensions, we can correctly predict memory address references of all the thread blocks in the kernel grid. Leveraging these observations, we propose a hardware predictor which takes advantage of

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00064

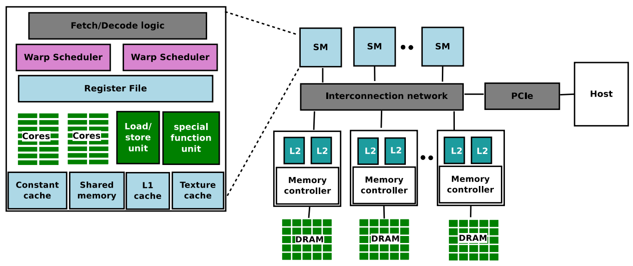


Fig. 2: GPU architecture

concurrency across SMs to detect the stride values at runtime. Once we have detected stride values for a load instruction, it is visible to all the SMs. We believe this is the first work that establishes relationship between memory accesses across thread blocks. Once the stride values are detected, we can determine the address references of all the warps of future thread blocks without waiting for executing any warp of the thread block. The main contributions of this paper are as follows:

- We characterize the relationship between memory address references of the thread blocks.
- We propose a hardware predictor which exploits SM level concurrency to detect the stride values at runtime. We implement this predictor in GPGPU-Sim.
- We discuss possible architectural optimizations based on early prediction of memory address references.

II. Background and Related Work

GPU Architecture: A GPGPU consists of multiple SMs connected to memory partitions through an interconnection network (Figure-2). When a kernel is launched, thread blocks are assigned to the SMs in a round robin manner until maximum number of thread blocks are assigned per SM. The number of concurrent thread blocks per SM is limited by resource requirements (such as registers, shared memory etc.) or by maximum number of thread blocks that can be assigned to an SM [8]. Within an SM, warps of the thread block are assigned to warp scheduler, which then issues instruction from the selected warp to SIMD lanes. Each SM has a private L1 data cache. L2 cache is partitioned and shared by all the SMs. Each memory partition has multiple L2 cache banks and a memory controller that connects to a DRAM channel.

Related Work: Koo et al. [9] have classified load instructions based on their locality behaviour and proposed a cache management policy. Oh et al. [5] have categorized load instruction as loads having strong locality across warps and loads having striding behaviour across warps. The authors have proposed a warp scheduling and a prefetching technique to improve cache utilization. Oh et al. [6] have proposed a warp progress aware prefetching technique which exploits striding behaviour across warps. It has been observed that intra-warp and inter-warp prefetching can be used to improve GPU performance because of constant stride across warps and within warps [7], [10]. Jeon et al. [4] have observed that strides are not visible across thread block boundaries due to round robin allocation of SMs to thread blocks and distribution of application data across thread blocks. However, warps within a thread block shows a constant stride. The authors also propose a prefetcher aware warp scheduling policy which enables early computation of the base address for thread blocks. Once the base address and inter warp stride is detected, prefetching is done for other warps in the same thread block. Jog et al. [11] have observed that consecutive thread blocks or CTAs have high DRAM page locality, and have proposed a CTA-aware warp scheduling technique to maximize bank level parallelism. They have also shown that consecutive warps access nearby memory locations, based on this, they have proposed a warp scheduling and spatial locality detection based prefetching to improve performance [12]. However, none of the existing works have discussed striding behaviour among warps of different thread blocks.

III. Multidimensional Grid Aware Address Prediction

We analyze the memory access pattern for different benchmarks from ISPASS [8], Rodinia [13], CUDA SDK [14] and SHOC [15] benchmark suites. The grid dimension and number of concurrent CTAs per SMs for these benchmarks are given in Table-I. As observed by previous works [4], [5], [7], warps

TABLE I: Benchmarks

Name	Grid Dimension	Concurrent thread blocks/core
convolutionSeparable (CNV)[14]	(24,768,1)	8
CP[8]	(8,32,1)	8
Dwt2D[13]	(3,12,1)/ (2,12,1)/(1,6,1)/ (6,15,1)	4 /6 /8
Lud[13]	(15,15,1) to (1,1,1)	6 / 8
Srad[13]	(128,128,1)	6
Stencil2D[15]	(64,2,1)	6
Histogram[14]	(4370,1,1)/(64,1,1)	6 /8
Scan[14]	(6656,1,1)	6
MD [15]	(48,1,1)	5
ScalarProd[14]	(128,1,1)	6
BFS [13]	(128,1,1)	3

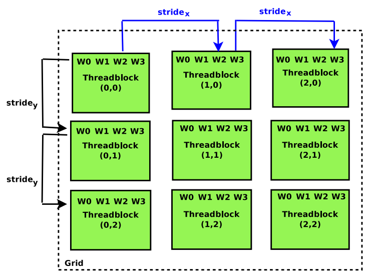


Fig. 3: Stride between warps of thread blocks along different dimensions

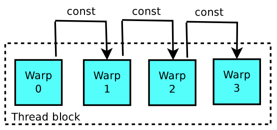


Fig. 4: Stride between warps within a thread block

within a thread block show striding behaviour in memory address reference. We also observed that benchmarks having multidimensional grids also exhibit stride behaviour across thread block boundaries. However, this stride value is different across different dimensions (Figure-3). For example, when a load instruction is considered, the benchmark CP exhibits stride values of 128 and 8192 along x and y dimensions, respectively. This inter-thread block stride is the difference between memory address references of warps, with the same local warp id (id within the thread block), of two consecutive thread blocks. In Figure-3, $stride_x$ is the difference between the memory address references of 2^{nd} warp from thread block (0,0) and thread block (1,0). Figure-5 shows the kernel code for CP that illustrates how the index of the data array to be accessed by a thread is determined. In this kernel, variables gridDim.x, blockDim.x and blockDim.y are constants; it represents the grid dimension along x dimension and the block dimension along x and y dimensions, respectively. The variable UNROLLX and BLOCKSIZEX (used in array index calculation in CP kernel) are also defined as constants. Figure-5 shows that the index of the array to be accessed by a thread is computed by using a constant multiple of xindex and yindex. However, these constant values are different for xindex and yindex.

```
__global__ void cenergy(int numatoms, float gridspacing,
float * energygrid)
{
unsigned int xindex = blockIdx.x * blockDim.x * UNROLLX
                      + threadIdx.x;
unsigned int yindex = blockIdx.y * blockDim.y
                     + threadIdx.y;
unsigned int outaddr = gridDim.x * blockDim.x * UNROLLX
                      *yindex+ xindex;
.....................................................
.....................................................
energygrid[outaddr] += energyvalx1;
energygrid[outaddr+1*BLOCKSIZEX] += energyvalx2;
}
```

Fig. 5: Kernel code of CP benchmark

The address references can be predicted by using the knowledge of the above stride values along x and y dimensions. This can be represented by equations (1) and (2).

A. Address referred by same local warp of target thread block

For a two dimensional kernel grid, when the address reference of the i^{th} warp within the reference thread block ($threadblock_{ref}$) is known, the predicted address reference of the i^{th} warp from target thread block is given in equation (1a). For example, this equation can give predicted memory address of 2^{nd} warp from thread block (1,1) using memory address reference of 2^{nd} warp from thread block (0,0). In equation (1a), $threadblock_t.x$ and $threadblock_t.y$ refer to the thread block id of target thread block along x and y dimensions; $stride_x$ and $stride_y$ refer to the stride between the memory address references by warps (with same local warp id) of two consecutive thread blocks along x dimension and y dimension, respectively; baseaddress denotes the address referred by the warp of the reference thread block.

$$\begin{aligned} address_t = baseaddress \\ + (threadblock_t.x - threadblock_{ref}.x) * stride_x \\ + (threadblock_t.y - threadblock_{ref}.y) * stride_y \end{aligned} \tag{1a}$$

Similarly, we can calculate the memory address references for a kernel having a 3D grid. This is given in equation (1b). In this equation, $stride_z$ is the stride between the memory address references by warps (with same local warp id) of two consecutive thread blocks along z dimension.

$$\begin{aligned} address_t = baseaddress \\ + (threadblock_t.x - threadblock_{ref}.x) * stride_x \\ + (threadblock_t.y - threadblock_{ref}.y) * stride_y \\ + (threadblock_t.z - threadblock_{ref}.z) * stride_z \end{aligned} \tag{1b}$$

B. Address referred by other warps of target thread block

Equation (2) specifies how predicted memory address references for j^{th} warp of the target thread block can be computed – when the memory address reference of i^{th} warp from reference thread block is known – by using predicted address of i^{th} warp from the target thread block. In equation (2), $baseaddress_{new}$ (computed as $address_t$ using equation (1)) is the memory address that will be referred by the warp of the target thread block, which has the same local warp id (i^{th} warp) as the warp of the reference thread block. InterWarpStride refers to the stride between memory address references from warps of the same thread block as shown in Figure-4. The address offset due to the inter warp stride is added to the base address of the target thread block to determine addresses corresponding to every warp of the target thread block.

$$\begin{aligned} address_t = baseaddress_{new} \\ + (warp_{target} - warp_{base}) * InterWarpStride \end{aligned} \tag{2}$$

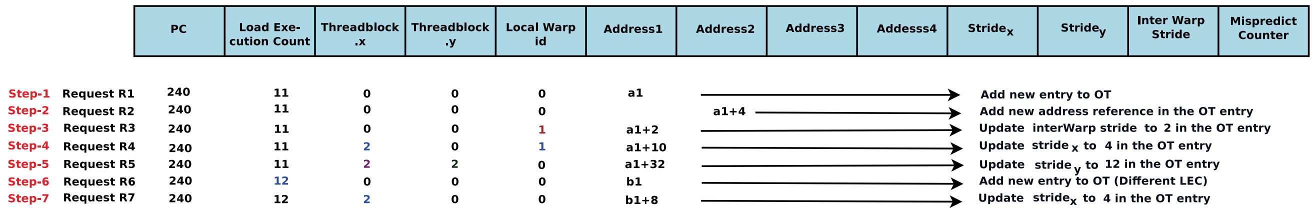
C. Runtime Stride Detection

Detecting strides across thread block boundaries within an SM is challenging because consecutive thread blocks are not assigned to the same SM. A thread block scheduler typically schedules thread blocks to different SMs in a round robin manner to improve parallelism and utilization [8], [11], [16]. At any instance of execution, each SM runs warps from different thread blocks. The memory address references of these warps running on different SMs should be observed for early determination of the strides along different grid dimensions. After the stride values are detected, it should be shared with all the SMs because this information can be used for memory address reference prediction of warps from different thread blocks.

We have proposed a cooperative predictor structure which takes advantage of concurrency among SMs to detect strides across thread block boundaries for both single dimensional and multidimensional grids. The predictor snoops on memory requests forwarded to memory partitions from SMs. The learned stride values are visible to all the SMs.

The cooperative predictor maintains a global table called the *observation table* (OT) which stores a set of entries. For each OT entry, the following information are stored:

- PC (Program Counter)
- LEC (Load Execution Count)
- $Threadblock_d$: Thread block id along dimension d
- local warp id: warp id within the thread block



		PC	Load Execution Count	Threadblock .x	Threadblock .y	Local Warp id	Address1	Address2	Address3	Adesss4	$Stride_x$	$Stride_y$	Inter Warp Stride	Mispredict Counter	
Step-1	Request R1	240	11	0	0	0	a1								Add new entry to OT
Step-2	Request R2	240	11	0	0	0		a1+4							Add new address reference in the OT entry
Step-3	Request R3	240	11	0	0	1	a1+2								Update interWarp stride to 2 in the OT entry
Step-4	Request R4	240	11	2	0	1	a1+10								Update $stride_x$ to 4 in the OT entry
Step-5	Request R5	240	11	2	2	0	a1+32								Update $stride_y$ to 12 in the OT entry
Step-6	Request R6	240	12	0	0	0	b1								Add new entry to OT (Different LEC)
Step-7	Request R7	240	12	2	0	0	b1+8								Update $stride_x$ to 4 in the OT entry

Fig. 6: Example showing how OT entries are updated after observing new requests; Action after each step is shown in right hand side

- memory address reference
- $stride_d$: Stride along dimension d
- Inter warp stride
- Mispredict Counter

Oh et al. [6] have used LEC to determine the progress of a warp. However, we observed that it can also help to distinguish between different instances of the same iterated load. Figure-7 shows a snippet of kernel code from the *scalar product* benchmark. Here, load in a different loop iteration is assigned a different LEC. Different warps of different thread blocks can be at different stages of progress depending on the warp scheduling policy and program characteristics. For example, one warp may be starting the loop iteration, whereas another warp might have executed multiple loop iterations. Hence, distinguishing between different instances of iterated load is important to get correct base address and stride value for address prediction. A warp can generate up to 32 memory requests for a load instruction. However, storing all the memory address references for a load PC will incur high storage overhead. Moreover, Jeon et al. [4] have observed that storing information for loads having more than four memory requests is not beneficial. Hence, we believe that storing information for load instructions having maximum four memory requests is enough for good prediction. The predictor also maintains a mispredict counter for each entry to limit the number of mispredictions, similar to [4]. When the number of mispredictions exceeds the misprediction threshold, the entry is not considered for further predictions.

```
for (int iAccum = threadIdx.x; iAccum < ACCUM_N; iAccum
+= blockDim.x)
{
  float sum = 0;
  for (int pos = vectorBase + iAccum; pos < vectorEnd; pos
  += ACCUM_N)
      sum += d_A[pos] * d_B[pos];
  accumResult[iAccum] = sum;
}
```

Fig. 7: Snippet of scalar product kernel code

Figure-8 shows how the predictor updates the entries in OT for an application having a two dimensional kernel grid. Figure-9 shows how an OT entry is updated after each step. When a new request arrives the predictor checks OT for a matching entry (Entry with same PC and LEC). If no match is found, predictor adds a new entry (Step-1 and Step-6 in both Figures-6 and 9). If a matching entry is found and the new request is from the same thread block and the

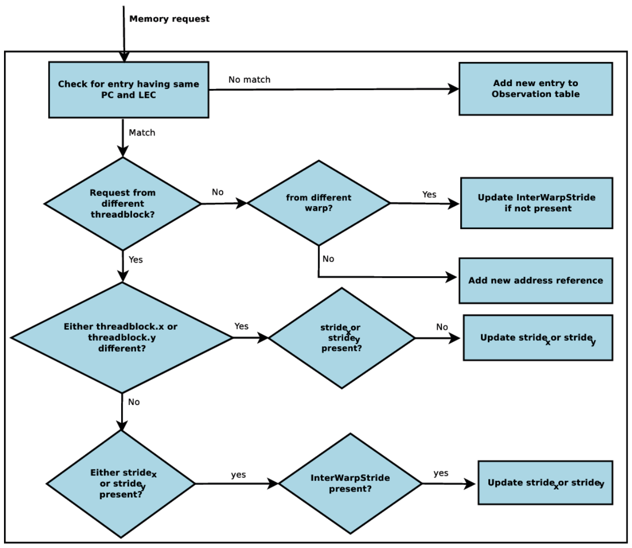
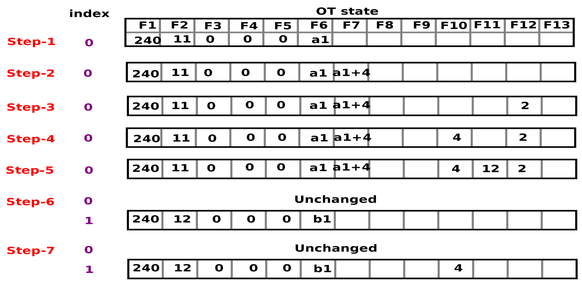

Memory request
Check for entry having same PC and LEC
No match
Add new entry to Observation table
Match
Request from different threadblock?
No
from different warp?
Yes
Update InterWarpStride if not present
Yes
No
Add new address reference
Either threadblock.x or threadblock.y different?
Yes
$stride_x$ or $stride_y$ present?
No
Update $stride_x$ or $stride_y$
No
Either $stride_x$ or $stride_y$ present?
yes
InterWarpStride present?
yes
Update $stride_x$ or $stride_y$

Fig. 8: Working of Predictor (For undefined choices OT entry is not updated)

same warp, then a new address is added (Step-2 in Figures-6 and 9). Least recently used (LRU) replacement policy is applied, when there is no empty entry in OT. If the request is from the same thread block but different warp, inter warp stride is determined and the table entry is updated (Step-3 in Figures-6 and 9). However, if the new request is from the warp of a different thread block, with only $threadblock_x$ or $threadblock_y$ different, then the predictor updates $stride_x$ or $stride_y$ in the OT entry using equations (1a) and (2). If the local warp id of the new request is same as that of the OT entry, then stride is the ratio of the address offset to that of the difference between thread block ids of the new request and the OT entry (Step-7 in Figures-6 and 9). Otherwise, first address offset due to inter warp stride is subtracted from the total offset between the stored and the new addresses. Then, the ratio of address offset to that of the difference in thread block ids is calculated to update the stride value (Step-4 in Figures-6 and 9). If the thread block dimension is different in both dimensions and either $stride_x$ or $stride_y$ value is present along with inter warp stride, the missing stride value is updated using equation (1a) and (2) (Step-5 in Figures-6 and 9). When all the stride values for a entry are present, the predictor is ready for prediction. This predictor can similarly be used to determine stride values for applications having 3D grids. The benchmarks we have analyzed have 1D or 2D grids.

This predictor can work with any warp scheduling policy.



	index	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Step-1	0	240	11	0	0	0	a1							
Step-2	0	240	11	0	0	0	a1	a1+4						
Step-3	0	240	11	0	0	0	a1	a1+4					2	
Step-4	0	240	11	0	0	0	a1	a1+4			4		2	
Step-5	0	240	11	0	0	0	a1	a1+4			4	12	2	
Step-6	0	Unchanged												
	1	240	12	0	0	0	b1							
Step-7	0	Unchanged												
	1	240	12	0	0	0	b1				4			

Fig. 9: State of OT entry after each step in Figure-6. F1...F12 represents Field1..Field12 of OT entry

Moreover, our design amortizes the storage overhead as it is shared by all the thread blocks across the SMs.

IV. Experiments

A. Methodology: We have implemented our proposed prediction logic in GPGPU-Sim v3.2.2 [8]. We have implemented the predictor of CTA-aware scheduling and prefetching scheme [4] for comparison. We have also compared with a variant of our cooperative prediction scheme where LEC is not taken into consideration. The simulation parameter values are specified in Table-II. For quantifying the performance of prediction, we have used two metrics: (a) prediction coverage (PCOV) and (b) prediction accuracy (PACC). PCOV and PACC are defined in equations (3) and (4), respectively .

$$PCOV = \frac{Total\ no.\ of\ predictions}{Total\ no.\ of\ memory\ load\ requests} \tag{3}$$

$$PACC = \frac{Total\ no.\ of\ correct\ predictions}{Total\ no.\ of\ predictions} \tag{4}$$

Prediction is done for the loads for which stride values along different grid dimensions and inter warp stride are already detected.

TABLE II: Experimental Setup

SM	15 SMs, 1400 MHz, maximum 8 thread blocks and 48 warps
Warp Scheduler	2 per SM, GTO/CTA-aware
L1 data cache	8-way, 32KB, 128B line, 32 MSHRs
L2 cache	8-way, 768KB, 128B line, 32 MSHRs per subpartition
GPU DRAM	924 MHz, 6 memory channels, FR-FCFS

It should be noted that with our approach, once an entry is ready for prediction, we can predict future memory access references for all the warps from all the thread blocks in the grid. Our approach requires only the thread block ids of the target thread block to compute the address references, which can be easily obtained from the variable gridDim in CUDA programs. The details of our predictor parameter is given in Table-III.

TABLE III: Predictor Configuration

No. of entries	1024 entries
Storage	entry size: 46B Total storage 46 KB distributed across L2
Mispredict counter threshold	64

B. Prediction Coverage and Prediction Accuracy: Figure-10 shows the prediction coverages for different prediction techniques. For benchmarks having iterated loads (Dwt2D, Stencil2D, BFS, Histogram, MD and ScalarProd), cooperative prediction with LEC provides better prediction coverage as compared to cooperative prediction without LEC. Inclusion of LEC to uniquely identify an instance of iterated load reduces mispredictions due to incorrect base address and stride values; this improves prediction accuracy. Prediction coverage is affected by a) prediction accuracy, b) time spent in learning the stride values, base addresses and c) the amount of regular loads. Cooperative technique shares learned stride values for a load across SMs instead of relearning the stride locally within each SM. Moreover, with our technique, base address of the thread blocks are computed using the stride values instead of waiting for one of the warps of the thread block to be scheduled. This reduces the learning time of the predictor. Hence, cooperative technique provides better prediction coverage than CTA-aware technique. Inclusion of LEC is also a factor that improves prediction coverage of our predictor. For BFS benchmark, CTA-aware technique has higher prediction coverage because CTA-aware predictor inside each SM continues prediction for a load instruction until its local mispredict counter is less than misprediction threshold. This means, even if misprediction threshold of one SM is crossed, predictors in other SMs can continue to predict for that load instruction. For such benchmarks our global mispredict counter threshold can be increased to allow higher prediction coverage with medium accuracy. Figure-11 shows the prediction accuracy of different prediction techniques. Our technique has similar accuracy as that of CTA-aware technique when LEC is considered.

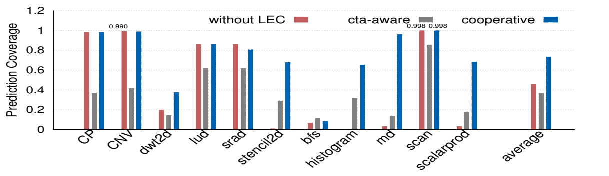


Fig. 10: Prediction Coverage; without LEC represents the results for coopearative technique without LEC

Higher prediction coverage without loosing prediction accuracy increases the effectiveness of a predictor. This is because predictor is able to predict for more number of load requests.

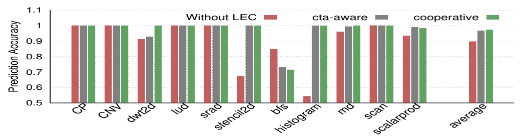


Fig. 11: Prediction accuracy; without LEC represents the results for coopearative technique without LEC

C. Sensitivity Study: Figure-12 shows the impact of the number of observation table (OT) entries on prediction coverage of our cooperative predictor. Prediction accuracy is not much affected except for BFS benchmark. Reducing the number of entries increased the prediction coverage but worsened the prediction accuracy for BFS. This is because mispredict counter is reset when the entry is evicted from predictor's OT and the learned information for that load PC is lost. When the number of entries in the OT is reduced to 64 entries, 128 entries and 256 entries, prediction accuracy is reduced by 19.7% , 9.7% and 2.12%, respectively. However when the number of table entries is reduced to 16 entries prediction coverage drops and prediction accuracy increases by 19.97%. This is due to thrashing of predictor's OT. For benchmarks like

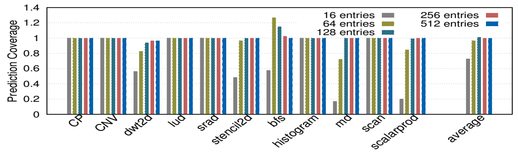


Fig. 12: Prediction coverage normalized to prediction coverage with 1024 entries

Dwt2d, Stencil2d, MD and Scalarprod prediction coverage is reduced due to insufficient number of entries to hold all the information. From Figure-12, it can been seen that for most of the benchmarks we are getting similar coverage with lesser number of entries. It also reduces the storage overhead of the predictor. It appears that 256 entries are enough for most of the benchmarks.

V. Application of Access Pattern Analysis

Cooperative prediction scheme can possibly be applied in various contexts. We discuss a few of those which constitute our future work.

Prefetching: In literature, inter warp stride behaviour has been used for data prefetching. Performance improvement due to prefetching depends on the following factors: accuracy of prediction, timeliness of prediction and prefetch, delay in serving demand requests due to prefetch requests and cache pollution. Cooperative predictor can provide timely prediction of addresses to be referred by all the warps of all the thread blocks. Selection of target warp for which data is to be prefetched and how early a prefetch request is generated play an important role in controlling the timeliness of data prefetching. For example, warps of the oldest thread block are more likely to be scheduled with GTO warp scheduling policy. Selecting the warp of the oldest thread block for prefetching may result in late prefetch. Similarly, prefetching for the youngest warp may lead to early prefetch and eviction of prefetched data before use. However, prefetching for warps of the next oldest thread block may result in timely prefetch. As our predictor can provide timely address predictions, the impact of other factors on prefetching could be explored with greater flexibility.

Thread block and Warp Scheduling: Prediction of memory address references of warps from all the thread blocks can help in designing better warp scheduling and thread block scheduling policy. Warps having memory requests to the same row or recently accessed rows can be scheduled near each other to improve the row buffer hit rate and the access latency of GPU DRAM banks [17]. Different memory partitions can have different amount of memory requests waiting to be serviced. Warps having memory requests for a memory partition, having lighter memory traffic, can be scheduled by the warp scheduler to improve memory access latency. Thread block scheduling to an SM can be done by observing memory access behaviour of thread blocks assigned to different SMs to improve bank level parallelism, row buffer hit rate, balanced memory traffic among memory partitions, etc.

Cache Working Set: Cooperative prediction can potentially be used to manage working set of applications to improve intra-warp and inter-warp data reuse.

VI. Conclusion

In this work, we have characterized the relationship between memory address references of warps across thread block boundaries. We have proposed a multidimensional grid aware address predictor which exploits SM level concurrency to detect strides across different dimensions of the grid at runtime. Our technique provides a cooperative approach, where once a pattern is learned it is shared among all the SMs. For applications having iterated load instructions, our technique provides better prediction accuracy by distinguishing different instances of iterated load. We have also discussed about future research directions based on early prediction of memory access references.

References

[1] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.
[2] J. E. Stone *et al.*, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.
[3] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
[4] H. Jeon *et al.*, "CTA-aware Prefetching for GPGPU," *Computer Engineering Technical Report Number CENG-2014-08*, 2014.
[5] Y. Oh *et al.*, "APRES: improving cache efficiency by exploiting load characteristics on GPUs," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 191–203, 2016.
[6] ——, "WASP: Selective Data Prefetching with Monitoring Runtime Warp Progress on GPUs," *TC*, no. 1, pp. 1–1, 2018.
[7] J. Lee *et al.*, "Many-thread aware prefetching mechanisms for GPGPU applications," in *MICRO*. IEEE, 2010, pp. 213–224.
[8] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*. IEEE, 2009, pp. 163–174.
[9] G. Koo *et al.*, "Access pattern-aware cache management for improving data utilization in gpu," in *ISCA*. ACM, 2017, pp. 307–319.
[10] A. Sethia *et al.*, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *PACT*. IEEE Press, 2013, pp. 73–82.
[11] A. Jog *et al.*, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 395–406.
[12] ——, "Orchestrated scheduling and prefetching for GPGPUs," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 332–343.
[13] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. IEEE, 2009, pp. 44–54.
[14] "NVIDIA CUDA SDK 4.2." [Online]. Available: https://developer.nvidia.com/cuda-toolkit-42-archive
[15] A. Danalis *et al.*, "The scalable heterogeneous computing (SHOC) benchmark suite," in *GPGPU-3*. ACM, 2010, pp. 63–74.
[16] M. Lee *et al.*, "Improving GPGPU resource utilization through alternative thread block scheduling," in *HPCA*. IEEE, 2014, pp. 260–271.
[17] H. Hassan *et al.*, "ChargeCache: Reducing DRAM latency by exploiting row access locality," in *HPCA*. IEEE, 2016, pp. 581–593.