# Rethinking Prefetching in GPGPUs: Exploiting Unique Opportunities

Ahmad Lashgar
*Electrical and Computer Engineering*
*University of Victoria*
*Victoria, BC, Canada*
*Email: lashgar@uvic.ca*

Amirali Baniasadi
*Electrical and Computer Engineering*
*University of Victoria*
*Victoria, BC, Canada*
*Email: amirali@ece.uvic.ca*

***Abstract*—In this paper we investigate static memory access predictability in GPGPU workloads, at the thread block granularity. We first show that a significant share of accessed memory addresses can be predicted using thread block identifiers. We build on this observation and introduce a hardware-software prefetching scheme to reduce average memory access time. Our proposed scheme issues the memory requests of thread block before it starts execution. The scheme relies on static analyzer to parse the kernel and find predictable memory accesses. Runtime API calls pass this information to the hardware. Hardware dynamically prefetches the data of each thread block based on this information. In our scheme, prefetch accuracy is controlled by software (static analyzer and API calls) and hardware controls the prefetching timeliness. We introduce few machine models to explore the design space and performance potential behind the scheme. Our evaluation shows that the scheme can achieve a performance improvement of 59% over the baseline without prefetching.**

## I. Introduction

Hardware or software prefetching mechanisms have been investigated as techniques to improve performance and energy-efficiency in computing systems by reducing average memory access latency. An effective prefetching mechanism must address three challenges: prefetching accuracy, timeliness, and extra memory bandwidth usage. First, prefetching mechanism must predict the memory addresses correctly. Wrong predictions may harm performance by increasing the number of memory requests, occupying on-chip buffers more often, increasing memory latency, and wasting memory bandwidth. Second, the correctly predicted data should be fetched timely; not too early, not too late. Otherwise, the prefetched data may not be fed to thread; if prefetched early, data might be evicted before the real request or if prefetched late, prefetching can impose redundant non- advantageous memory fetches. Finally, prefetching usually results in extra traffic, along the application traffic, to fetch data from lower memory levels. Effectiveness of a prefetching mechanism relies on smart bandwidth usage management.

In this paper, we study GPGPUs and identify unique opportunities to address the aforementioned three challenges. In GPGPUs, hundreds of the threads are grouped into thread blocks. Since threads of the same thread block share many resources, programmers are encouraged to increase the locality within the thread block. As we show in this paper, in most cases, each thread block works on a tile of data, which is statically predictable for a given thread block. Exploiting this behavior, we address prefetching challenges.

We propose a prefetching scheme working at the thread block granularity. We introduce a combination of software and hardware techniques. We use software solutions to address the accuracy challenge of prefetching and hardware solutions to address timeliness and memory bandwidth challenges.

On the software side, we develop a static analyzer to parse GPU kernel and determine the ranges of the data that each thread block accesses. Extra procedure calls are introduced to pass these information in an abstract form to GPU. The procedure calls set a prefetch table in the hardware, right at the kernel launch time.

On the hardware side, a prefetch table per kernel is maintained. Upon initializing a new thread block for execution, the thread block dispatcher injects a burst of memory requests to fetch the thread block's data, based on the information maintained by the prefetch table. To assure the prefetching timeliness, all threads of the thread block are stalled until prefetching is finished. Stalling has also the advantage of stoping threads from issuing redundant memory accesses, avoiding the generation of excessive memory bandwidth traffic that prefetchers are prone to.

We develop various machine models to investigate the performance potential behind this scheme. We show the proposed scheme can achieve up to 59% performance improvement over the baseline without prefetching.

The remainder of this paper is organized as follows. In Section II, we investigate the static predictability of memory accesses in GPGPU workloads. In Section III, we introduce our hardware-software approach for prefetching. In Section IV, we overview experimental methodology. In Section V, we investigate the performance potential of the proposed prefetching system. In Section VI, we review previous work. Finally, in Section VII, we offer concluding remarks.

## II. Static Predictability of Memory Accesses

In this Section, we investigate the static predictability of memory accesses in GPGPUs. In particular, we investigate the static predictability of global memory accesses in CUDA benchmarks. We developed a static analyzer parsing one CUDA for C kernel at a time. For each kernel, the static analyzer performs three phases. First, the analyzer forms a list of variable names which are listed in the kernel's arguments as pointer variables. These pointers essentially point to a location in the global memory. Within the kernel, each pointer is treated as array. Second, for each of these arrays, the static analyzer extracts the array indexes which are referred in the kernel body. Finally, the static analyzer examines static predictability of the value of each index. Below we explain our definition of static predictability and report the findings of static analysis under different CUDA kernels.

### A. Static Predictability

We classify array indexes into *statically predictable*, *quasi-static predictable*, and *unpredictable*, based on the static predictability of the index value. We declare the predictability of index values based on the operators (add, multiply, shift, etc.) and terms (variables or constants) constructing the index expression. We identify predictability in two steps:

**Processing operators**. We consider index as unpredictable if it is composed of any operator other than addition, subtraction, and multiplication. As we clarify later, this reduces hardware complexity for address calculations on predictable addresses.

**Processing terms**. If the index is not found unpredictable in previous step, then the analyzer examines the terms of the expression as it might be predictable (either statically or quasi-static). The index is statically predictable, if the terms are values or constant variables (e.g. *a[0]* or *a[blockDim.x]* in CUDA). The index is quasi-static predictable, if the memory index term depends on at least one built-in CUDA thread identifier (*threadIdx* or *blockIdx*). Since the thread and thread block identifiers are known at the time of dispatching the thread block, we refer to this type as quasi-static predictable. In unpredictable, the memory index term depends on a runtime variable. A runtime variable can be a memory location (e.g. *a[b[0]]*) or a control-dependent variable (e.g. *a[condition?1:0]*).

### B. Results

Here we report static predictability findings in 14 memory-bounded benchmarks, as measured by our static analyzer (for the methodology refer to Section IV). Figure 1 reports the number of unique global memory arrays found in the kernel body of various benchmarks. Indeed, these arrays are the pointer variables in the kernel's arguments. The figure also reports the number of array indexes found in the kernel (only memory reads). In the case of multiple kernels in the benchmark (which is the case for BKP, BPT, NN, RDC, and SRD), we report the summation of arrays and indexes which are found in each kernel. As shown, the number of arrays ranges from two (in NNC benchmark) to 15 (in BPT) while the number of array indexes ranges from two (in HSP) to 71 (in BPT).

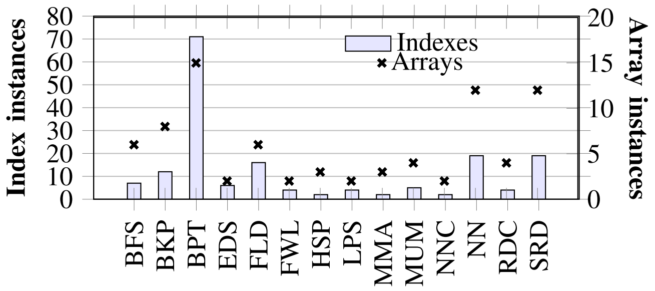

Figure 1. The number of arrays and indexes identified by the static analyzer.

Figure 2 complements Figure 1 and reports the breakdown of array indexes into statically predictable, quasi-static predictable, and unpredictable. As reported, none of the array indexes are found statically predictable in the evaluated kernels. But a significant portion of array indexes are found quasi-static predictable. In NNC and RDC, unpredictable array indexes depend on an induction variable. While we consider them unpredictable in the breakdown, this type may also be considered as quasi-static predictable, if the boundaries of the loop are predictable. In LPS and MUM, unpredictable array indexes depend on control statements. In BPT and FLD, unpredictable array indexes depend on a load from another array. In BFS, one of the unpredictable array indexes depends on an induction variable and the other unpredictable index depends on a load from another array. In NN, 17 unpredictable array indexes depend on an induction variable. In this case, the range of this variable can be evaluated statically.

Generally, very high percentage of array indexes are found to be quasi-static predictable. This number ranges from 38% in FLD to 100% in many benchmarks. Quasi-static predictable indexes only depend on CUDA *threadIdx* and *blockIdx* variables which are evaluated at the time of dispatching the thread block. Hence, during runtime, a range of the data that each thread block accesses can be predicted. In the rest of the paper, we build on this observation and propose a combined software and hardware solution to prefetch data for each thread block.

## III. Proposed Prefetching Scheme

### A. Software Side

The software side of our proposed prefetching solution includes an API for passing static analysis information from CUDA applications to GPU hardware, at the kernel launch time. This information is passed for the array indexes which are marked as predictable by the static analysis. The

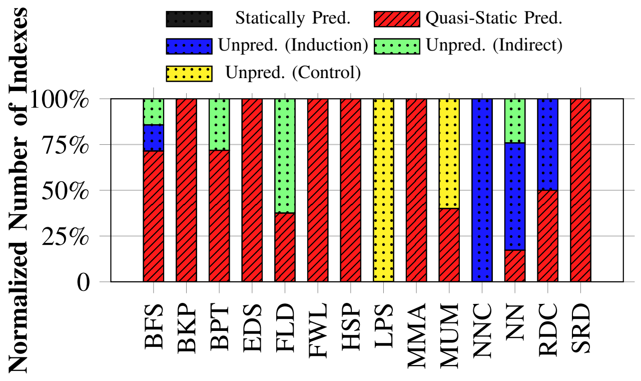

Figure 2. Breakdown of array indexes into statically predictable, quasi-static predictable, and unpredictable. Unpredictable indexes are either depending on induction variable (Induction), another memory load (Indirect), or a control statement (Control).

information guides the GPU to prefetch data chunks required by each thread block, after dispatching the thread block and before issuing any instruction from individual threads of the thread block. The API passes the information to the GPU in an abstract form. Below we first discuss the essence of information which is needed for prefetching and introduce the API to pass that information. Secondly, we present an example to clarify the API usage.

*1) API:* If the index is marked as predictable, static analysis evaluates the array index as an expression of i) thread identifiers and ii) constant terms. Using this expression, the value of an array index can be obtained for each thread of thread block. Knowing the threads that belong to each thread block, the compiler statically specifies the domain of values that index may have within each thread block. The domain of values of index is essentially the range between minimum and maximum values that index may have. Since the minimum and maximum varies for each thread block, the compiler passes the essential information to hardware and allows the hardware to calculate the minimum and maximum at runtime. Below we elaborate on the essential information that has to be passed to hardware.

In order to determine the data chunk of each thread block, the static analyzer looks for the *minimum* and *maximum* value of the array index for each thread block. The compiler evaluate the index expression to specify its minimum and maximum values based on the fact that in CUDA there are four three-dimensional thread identifier variables: *gridDim*, *blockDim*, *blockIdx*, and *threadIdx*. The index expression can be reduced to only one variable (*blockIdx*) provided that: i) *gridDim* and *blockDim* are constant terms within a particular kernel and can be evaluated as a constant value, ii) Lowest and highest values of *threadIdx* are constant, and iii) *blockIdx* is determined at runtime upon dispatching the thread block

Applying the above assumptions to any predictable array index (statically predictable or quasi-static predictable) simplifies the index to an expression of constant values and only one runtime variable (blockIdx). In general, predictable array index expression can be presented in the polynomial form of Equation 1.

$$v = a_0 + \sum_{i=1}^{n} (a_i \times bIdx^i) \tag{1}$$

where $a_i$ for $0 \leq a_i \leq n$ are life-time constant terms, $bIdx$ is *blockIdx*, and $n$ is the degree of polynomial. If an array index is constant (or statically predictable) then $a_i$ for $1 \leq a_i \leq n$ is zero. Array index is affine constraint of degree one if $a_i$ for $2 \leq a_i \leq n$ is zero. Similarly, array indexes with affine constraint of higher degrees are possible. To simplify API and hardware support, here we limit prefetching to arrays having indexes with affine constrains of degrees one or lower. Notice that the degree of array index can be identified by the static analyzer by simply searching for the number of *blockIdx* variables that are multiplied. In the evaluated benchmarks, we found all quasi-static predictable in the first degree or linear polynomial.

For every predictable array index, the static analyzer finds $a_0$ and $a_1$ by parsing the dependency graph. It returns the pair of $a_0$ and $a_1$ to specify minimum and maximum values of index, based on varying values of *threadIdx*. Then, the proposed API passes two ($a_0$, $a_1$) pairs, one for the minimum and the other for the maximum, to guide the GPU in specifying the lowest and highest values of the thread block's data. The prefetcher, on the hardware side, uses this information and dynamically calculates the minimum and maximum values of indexes, provided that the value of the last unevaluated variable, *blockIdx*, is known at the time of dispatching the thread block. Minimum/maximum point to the beginning/end of range of addresses that threads of thread blocks may request. After calculating the range of address, the hardware sends a burst of read requests to memory controller. This prefetches the entire range to the GPU core that the thread block is dispatched to. The core starts issuing instructions from this thread block as soon as the entire range is prefetched.

Here is the proposed API prototype:
*cudaSetCTATracker(baseptr, typesize,*
*minbidxp, minbidyp, minbidzp, minoffset,*
*maxbidxp, maxbidyp, maxbidzp, maxoffset);*

This procedure sets registers on the GPU before kernel launch and simplifies calculating the address of continuous data chunk of each thread block required by the prefetcher. The prefetcher hardware then calculates the range of data using the following linear equations:

$$\begin{aligned} min = baseptr + typesize \times (&blockIdx.x \times minbidxp \\ &+ blockIdx.y \times minbidyp \\ &+ blockIdx.z \times minbidzp + minoffset) \end{aligned} \tag{2}$$

and a very similar equation is used for calculating *max*.

### *B. Hardware Side*

Below we discuss hardware modifications of the proposed prefetching scheme.

*1) Prefetch table:* The proposed API calls allow compiler/programmer to pass the prefetching information to hardware. Hardware uses this information to dynamically prefetch the data regions associated with each thread block. To maintain this information in the hardware, we propose enhancing thread block dispatching unit with a table; refers to as prefetcher table. Each row of the prefetcher table stores the information passed by a single API call. Each row registers base pointer, data type size, minimum products, and maximum products.

*2) Prefetch controller unit:* The proposed prefetch controller unit, or simply prefetcher, is shared among all thread blocks and resides in the thread block dispatching unit. Prefetcher reads every valid row of the prefetch table to prefetch the data region associated with the ready-for-dispatch thread blocks. Upon dispatching every thread block, the prefetcher reads each row of the prefetcher table, calculates the boundary of data region of the row using Eq. 3 and 4, and issues a burst of requests per row to prefetch the data region. Requests are issued from thread block dispatcher unit to memory controllers. In each request, prefetcher attaches the network address of the target GPU core, the core on which the thread block is dispatched, as the destination. This allows the network to route the reply packets from DRAM directly to the target core. On the core, the prefetched data is stored in a prefetch buffer.

*3) Prefetch buffer:* The proposed prefetching scheme, also extends the GPU cores with a logical prefetch buffer. The buffer stores the burst of incoming prefetch data for every outstanding thread block of the core. Physically, this buffer can be a dedicated cache or available caches of GPU core (e.g. data cache or software-managed cache). The buffer is also responsible to count the number of received packets for each newly scheduled yet-stalled thread block. The buffer signals the core's warp scheduler to activate the thread block once all the packets from prefetching unit are received at the core's end.

*4) Prefetch timeliness:* After scheduling a thread block, GPU core can start execution of the thread block in parallel to the prefetcher's prefetching operations. However, there may be a delay between prefetching the entire thread block's data region into the prefetch buffer and demanding the data by a thread within the thread block. Since handling such cases increases the hardware complexity, we simply stall the entire thread block until the prefetching is completed. Later, in Section V, we show the performance potential behind interleaving the execution of thread blocks and completing of prefetching requests.

## IV. Experimental Methodology

**Modeling the prefetching scheme**. We develop a static analyzer that highlights predictable array indexes. We reform this information to the proposed API calls and inject the code manually to the benchmarks' source code. We use GPGPU-sim 3.2.2 [1] for modeling both hardware and software sides of the proposed prefething scheme. We used GPGPU-sim default configuration files to model a hardware similar to NVIDIA GTX 480 as the baseline GPU of this study [3].

**Benchmarks**. We used 22 benchmarks included from Rodinia, GPGPU-sim, and two third-party applications (edge-detection by sobel filter (EDS) and matrix-matrix add (MMA)). We limited the study to the benchmarks that are not merely compute-bounded. We only consider benchmarks which show tangible performance improvement under the ideal zero-latency memory machine. We model such machine by assuming an ideal L1 caches which has a hit rate of 100%.

## V. Machine Models

The proposed prefetching scheme is prone to two performance limiting overheads. The first overhead is the excessive memory bandwidth usage (caused by prefetching). This may increase average memory access latency since there are more memory requests contending for resources. The second overhead is stalling thread blocks till prefetching completes. This may harm performance if there are not enough concurrent thread blocks to hide the latency. Below we introduce few machine models to investigate the effect of these overheads on the performance of the proposed prefetching scheme. We also use machine models to investigate the prefetching accuracy and DRAM impacts. For the evaluation that follows, we assume 64 entries per prefetch table and an ideal unlimited fully-associative prefetch buffer. Here are the machine models:

- **I-Machine:** is an ideal implementation resolving both overheads. First, no memory request is issued by the prefetcher, so the prefetcher does not impact the memory bandwidth. Second, the prefetcher instantly fetches the data of each thread block into the destination prefetch buffer making prefetching zero-latency. Hence, thread blocks are not stalled for prefetching. This machine shows the performance potential behind the proposed scheme if all overheads are mitigated.
- **S-Machine:** is a semi-ideal implementation modeling the memory bandwidth demand of the prefetcher but assumes a zero-latency prefetching. The prefetcher issues a burst of memory requests to prefetch the data of every thread block. But, on the core side, the data is ideally fed to threads in zero latency, meaning thread blocks are not stalled and the machine instantly fetches the data of each thread block. This machine shows the performance potential behind the proposed scheme

once the overhead of stalling the thread blocks are mitigated but under real bandwidth restrictions.

- **R-Machine:** is the non-ideal implementation modeling both overheads; the prefetcher memory bandwidth demand and stalling thread blocks until completion of prefetching requests. The only difference between this machine and the realistic implementation is the ideal prefetcher buffer that this machine has.

## A. *Performance*

Figure 3 compares the performance of machine models under various workloads. Numbers are normalized to the baseline machine without prefetching.

Under removal of all prefteching overheads I-Machine reports the full performance potential behind the scheme. The numbers range between 1% to 101% improvement over the baseline. This number is significant in BFS, EDS, MMA, and SRD. In these cases, the prefetch buffer covers a large portion of dynamic memory accesses (up to 86%). Also performance improvement can be seen under HSP, NN, and NNC. Under HSP, prefetching scheme fetches two two-dimensional input tiles for each thread block. Also there is an overlap among the input of different thread blocks. This overlap contributes in improving L2 cache hit rate. This explains why S- and R-Machine (modeling the memory bandwidth demand of prefetcher and issue memory requests) outperform I-Machine. Under NN, all machines sustain the improvement of 8% which is close to the performance of the zero-latency memory machine (9% improvement, as reported by the label below the bar group). Under NNC, I- and S-Machine improve the performance, however, R-Machine negates this. Since NNC runs under very low occupancy (thread block size is 16 and 8 thread blocks are executed concurrently), stalling the threads for the completion of prefetching degrades performance.

The performance potential is low under BKP, BPT, and RDC. We explain this for each of the benchmarks separately. Performance in **BKP** is not heavily bounded by memory performance, as we found that even an ideal zero-latency memory improves performance in BKP only by 11% (reported by the label below the bars in the figure). This leaves a little room for improving the performance with prefetching. Under **BPT**, firstly, predictable memory accesses account for a small portion of the total dynamic memory accesses. We found that only 15% of the memory requests are covered by the prefetcher. BPT has lots of unpredictable memory accesses due to indirect dependencies[1]. Secondly, the data type of prefetched data is an *struct* of two 512-element arrays of *int*s (nearly 4K per thread block). Prefetching scheme conservatively fetches entire bytes of these two arrays. BPT kernel, however, is divergent and does not access whole the array. In point of fact, one of the arrays is used thoroughly

[1]Indirect dependency is explained in Section II-B.

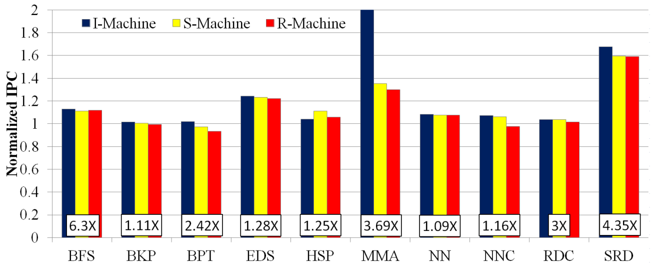

Figure 3. Comparing performance of ideal prefetching machine models. Numbers are normalized to baseline without prefetching. At the bottom of bars, the number indicates the speedup from the zero-latency memory machine.

by all threads of the thread block and the accesses to the other array is control-dependent and the array is partially accessed. Compared to the no-prefetching baseline, the extra memory accesses that the prefetcher issues explains why the scheme performs poor under BPT. In **RDC**, there are two kernels and 50% of array indexes are predictable in total (as reported earlier in Figure 2). Indexes from one kernel are all predictable while indexes from the other kernel are all unpredictable, due to control dependencies. Despite this seemingly promising static analysis, the predictable kernel contributes less than 20% to the total execution time of the RDC. This leaves little room for improving performance.

S- and R-Machine add realistic overheads to I-Machine and, as expected, generally S- and R-Machine have lower performance (except in HSP where realistic memory bandwidth of prefetcher improves L2 cache hit rate). S-Machine models the excessive memory bandwidth usage of the prefetcher. As reported, the memory bandwidth demand of the prefetcher may degrade performance by up to 55% (in MMA benchmark). On average, the memory bandwidth usage of the prefetcher impacts performance of I-Machine by less than 6%.

Built on top of S-Machine, R-Machine also models the overhead of stalling thread blocks until prefetching completes. This overhead may harm performance of the applications that have low occupancy, e.g. BPT (66% occupancy) and HSP (50% occupancy). However, the application tolerates this overhead if the occupancy is high. For example, in SRD, the occupancy is 100% and R-Machine is able to improve performance by 59%. BFS, EDS, and MMA also have occupancies of 100% which allows tolerating the latency of stalling thread blocks and sustaining the performance improvement of S-Machine.

## B. *Detailed Analysis*

Prefetcher generates a burst of memory requests for a contiguous data region. This traffic pattern can improve the DRAM row locality by mitigating row changes. However, this will not necessarily turn into faster DRAM, since the prefetching scheme may meanwhile increase the total

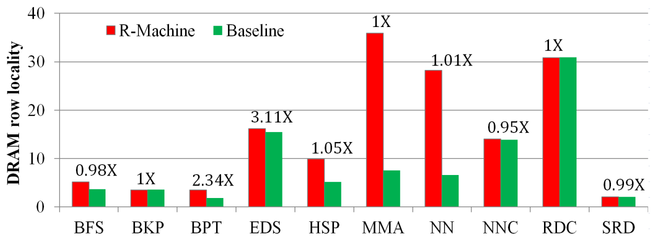

Figure 4. Comparing average DRAM row locality of R-Machine prefetching to the baseline. Numbers above the bars indicate the ratio of the total DRAM accesses under R-Machine to the baseline.

number of memory requests. This is the case when prefetching accuracy is low or prefetcher fetches the entire data range while the thread block sparsely accesses the range. Generally, to have a faster DRAM, the prefetching scheme must preserve i) high row locality at the DRAM and ii) high hit rate at the prefetch buffer. Below we investigate this aspect of the scheme through machine models.

**DRAM row locality**. To maximize exploitable locality, we use the following two techniques in the hardware. First, the prefetcher issues the request of a single data region (or single entry of prefetch table or *cudaSetCTATracker()* call) back-to-back, avoiding early interleaving. Second, the memory controller prioritizes prefetcher requests over the requests coming from GPU cores. The combination of these techniques lowers DRAM row changes. Figure 4 compares the average DRAM row locality of R-Machine to the baseline. *Average DRAM row locality* is defined as the ratio of total row accesses to total row changes. As shown, R-Machine generally improves DRAM row locality. Beside the improvements in row locality, the number of DRAM accesses under R-Machine are higher than the baseline in some cases, e.g. BPT and EDS. In Figure 4, this number is shown above R-Machine bars for each benchmark, ranging from 1% (in RDC benchmark) to 3.11X (in EDS). In BPT, this higher memory reads are explained by the conservative prefetching which fetches entire range of data, while the kernel sparsely accesses the data. In EDS, data type size is one byte. Accesses from thread of the thread block form memory requests for 16 consequent bytes. The baseline is able to issue variable packet sizes, e.g. 32-byte, 64-byte, or 128-byte. Prefetcher, however, is restricted to fetch 128-byte requests. This explains why prefetcher issues nearly 4X higher memory requests than the baseline.

**Performance of the prefetch buffer**. To understand the performance of the prefetch buffer, we report the coverage. Coverage is defined as the percentage of the memory requests that are captured by the prefetch buffer (including both read or write requests). Figure 5 reports the coverage and also the maximum buffer size needed by each benchmarks. Coverage ranges from nearly zero in RDC to 90% in NNC. Average buffer size per thread block is 2 KB (204-byte in BKP and 10KB in NN).

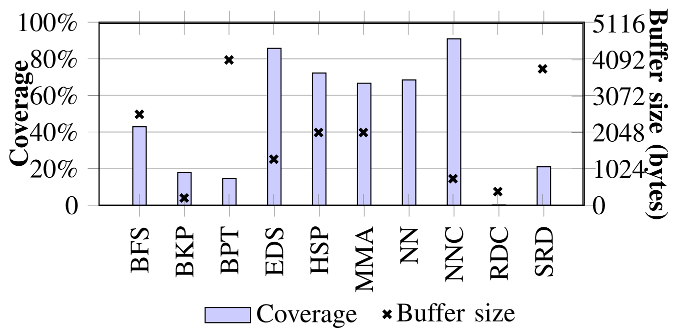

Figure 5. Maximum prefetch buffer size for a thread block and the percentage of memory accesses that is captured by the prefetch buffer. Buffer size for NN is 10KB.

## VI. Related Work

Ryoo et al [4] investigated software prefetching in matrix multiplication test case. They found prefetching advantageous as long as the register pressure does not degrade occupancy. Lee et al. [2] evaluate several hardware prefetching mechanisms. Generally, they found that memory patterns are highly predictable. They also found out that the real challenge is excessive memory bandwidth usage and timeliness of the prefetcher. They introduced a threshold-based heuristic to address these challenges. Sethia et al [5] used prefetching as a technique to improve energy-efficiency.

## VII. Conclusion

We investigated static predictability of memory accesses in GPGPUs, at the thread block granularity. Finding a high percentage of accesses predictable, we proposed a hardware-software prefetching scheme to improve performance. In this early presentation of our work, we focused on presenting the potential advantages behind the scheme. It is our on going research to further investigate this scheme, specially in improving static analysis, optimizing API calls, and designing an efficient prefetch buffer.

## References

[1] A. Bakhoda et al. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS 2009*.

[2] J. Lee et al. Many-thread aware prefetching mechanisms for gpgpu applications. In *MICRO '43*.

[3] NVIDIA Corp. GeForce GTX 480. *Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications*.

[4] S. Ryoo et al. Program optimization space pruning for a multithreaded gpu. In *CGO '08*, pages 195–204.

[5] A. Sethia et al. Apogee: Adaptive prefetching on gpus for energy efficiency. In *PACT '13*.