Microprocessors and Microsystems 63 (2018) 169–181

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

TELEPORT: Hardware/software alternative to CUDA shared memory programming☆

Ahmad Lashgar[a,*], Ehsan Atoofian[b], Amirali Baniasadi[c]

[a] Electrical and Computer Engineering Department, University of Victoria, Canada
[b] Lakehead University, Canada
[c] Electrical and Computer Engineering Department, University of Victoria, 3800 Finnerty Rd, Victoria BC V8P 5C2, Canada

ARTICLE INFO

Article history:
Received 18 March 2018
Revised 11 September 2018
Accepted 13 September 2018
Available online 14 September 2018

Keywords:
Computing methodologies parallel programming languages
Software and its engineering runtime environments
Hardware hardware accelerators
Computer systems organization single instruction
Multiple data
Accelerator
GPGPU
CUDA
Software-managed cache
Prefetching

ABSTRACT

Using software-managed cache in CUDA programming provides significant potential to improve memory efficiency. Employing this feature requires the programmer to identify data tiles associated with thread blocks and bring them to the cache explicitly. Despite the advantages, the development effort required to exploit this feature can be significant. The goal of this paper is to reduce this effort while maintaining the associated benefits. To this end, we first investigate static precalculability in memory accesses for GPGPU workloads, at the thread block granularity. We show that a significant share of addresses can be precalculated knowing thread block identifiers. We build on this observation and introduce TELEPORT. TELEPORT is a novel hardware/software scheme for delivering performance competitive to software-managed cache programming, but at no extra development effort. On the software side, TELEPORT's static analyzer parses the kernel and finds precalculable memory accesses. We introduce Runtime API calls to pass this information to hardware. On the hardware side, this information is used to fetch the data required for each thread block into shared memory before the thread block starts execution. With this hardware support, TELEPORT outperforms hand-written CUDA code as a result of the associated DRAM row locality improvement. Investigating a wide set of benchmarks, we show that TELEPORT improves performance of hand-written implementations, on average, by 32% while reducing development effort by 2.5X. Our estimations show that the hardware overhead associated with TELEPORT is below 1%.

1. Introduction

Conventional GPUs have had a small cache per core to buffer input/output of the graphical pipeline. This buffer is critical to the performance of the processor as it facilitates avoiding significant amount of global synchronization and DRAM accesses. Later, in the GPU computing era, GPGPU programming models introduced a new memory hierarchy, called shared memory in CUDA (or local memory in OpenCL), to allow programs to take advantage of this buffer.[1] The new memory hierarchy is a software-managed cache (the same buffer in graphical pipeline) and can be shared among collaborating threads (known as thread blocks). This cache can be exploited in various ways to improve kernel's memory efficiency [24,38,42]. By using the software-managed cache, compared to hardware-managed cache, the programmer can assure the data will not be evicted by other cache requests.[2] Also parallel threads can fetch the data tile collaboratively to improve memory-level parallelism. Typically, software-managed cache accesses have 8X higher bandwidth [41] and 20X lower delay [42] than DRAM accesses and fetching the data from the cache is 32X more energy-efficient than DRAM [5].

Programming the software-managed cache, however, involves tremendous development effort (defined as the amount of effort required to develop the software, estimated by the number of lines of code.). Firstly, the programmer should identify the data to be fetched into the cache. Candidate data are the arrays representing high temporal/special locality. Secondly, code should be modified to add an extra array explicitly representing the software-managed cache. To this end, two set of indexes should be maintained; global and shared memory spaces.

☆ This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

* Corresponding author.
E-mail address: lashgar@uvic.ca (A. Lashgar).

[1] This paper uses CUDA terminology.

[2] Cache is allocated at the dispatch time of thread block and deallocated at the end of its execution.

https://doi.org/10.1016/j.micpro.2018.09.004
0141-9331/

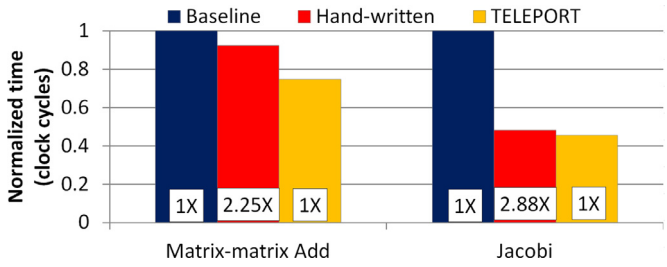

Fig. 1. Comparing three different implementations of Matrix-matrix Add and Jacobi iteration. Bars report kernel time and numbers below the bar indicate the development effort, normalized to Baseline (Effort is estimated by the number of lines of code.).

In this paper, we introduce TELEPORT, a hardware/software mechanism, to partially offload the shared memory development effort from the programmer to the compiler, while not sacrificing performance. Under TELEPORT, the compiler analyzes CUDA kernels to statically identify the data tiles assigned to each thread block. Later, during runtime, hardware loads the designated tiles into the software-managed cache in advance for each thread block. When both TELEPORT and hand-written CUDA versions implement similar algorithms, TELEPORT can outperform CUDA versions via a unique hardware optimization, improving DRAM row locality.

On the software side, we develop a static analyzer to parse the kernel, identify the candidate arrays, and determine data ranges that each thread block accesses. Extra procedure calls are introduced to pass this information in an abstract form to GPU. The procedure calls configure *preload table* in the hardware, before kernel launch calls. These steps can be fully integrated into the kernel compilation phase.

On the hardware side, a logical preload table per kernel is maintained. Upon dispatching a new thread block to GPU core, the thread block dispatcher issues a burst of memory requests to fetch the thread block's data, using the information in preload table. All threads of the thread block are put on hold till tiles are loaded completely. Putting the thread block on hold also stops threads from issuing redundant memory accesses, avoiding the generation of excessive memory bandwidth traffic (We also study alternatives, leveraging timeliness and bandwidth demand to maximize performance.)

To take a glance at the performance and development effort advantages of TELEPORT, we present a subset of findings in Fig. 1 where we compare three different implementations of two benchmarks: matrix-matrix addition (A + B = C) and Jacobi iterative method (See Section 4 for methodology.) The first implementation (Baseline) does not use the software-managed cache. The second implementation (Hand-written) employs the software-managed cache. The third implementation (TELEPORT) analyzes the source code of the Baseline implementation and takes advantage of the opportunities available for using software-managed cache (notice that this implementation relies on hardware support.) Below we explain each benchmark.

Matrix-matrix addition. Under Baseline, every thread calculates one element of the output matrix. While performance is very poor, the development effort is fairly low. Under Hand-written, threads of every thread block collaboratively fetch tiles of A and B to the software-managed cache and calculate the sum. This implementation exploits data locality among threads of the thread block and removes redundant memory fetches within thread block. While performance is very high, Hand-written implementation demands higher development effort compared to Baseline (2.25X greater). Under TELEPORT, development effort is similar to Baseline. During the compile time, the static analyzer parses the Baseline's kernel to specify the ranges of A and B that are assigned to each thread block. Finding opportunities for caching A and B, the compiler injects API calls before the kernel launch to configure the preload table for the kernel. With hardware support, the input tiles are fed to the thread block during runtime through the software-managed cache. As reported, TELEPORT outperforms Hand-written by 23%. As we explain later, part of this improvement comes from lowering the number of dynamic instructions.

Jacobi iterative method. In this benchmark, every thread calculates one element in the output by applying the smoothing function over nine elements (8 neighbors plus the element itself). This results in a strong data spatial locality among the thread inputs data as threads use adjacent elements to calculate the output. Under Baseline, threads fetch the elements from global memory separately. This implementation relies merely on memory access coalescing capabilities of hardware [27]. Hand-written fetches a tile of data, covering the input of all collaborating threads, into the shared memory. This lowers the global memory load instructions by nearly 9X (for a tile of 16*16 threads, Baseline performs 16*16*9 loads and Hand-written performs $(16+1)*(16+1)*1$ loads.). But not all of this gain translates to speedup, since Hand-written need to access the shared memory for (16*16*9)*2 times[3]). TELEPORT analyzes Baseline kernel and identifies the input tile associated with collaborating threads. This, combined with hardware support, lowers the development effort of Hand-written by 2.88X and improves its performance by 6%.

In this paper, we investigate a wide set of benchmarks and show TELEPORT improves performance of Baseline and Hand-written implementations, on average, by 56% and 32%, respectively. We also show TELEPORT lowers development effort by 1.46X to 3.4X, compared to Hand-written. TELEPORT uses the unused space in the software-managed cache of the GPU core as a buffer for storing tiles. The hardware overhead associated with TELEPORT includes the preload table and TELEPORT's controller unit (which are shared among GPU cores) plus an array of tags per GPU core for indexing the software-managed cache. Our estimations show that the hardware overhead is below 1%.

In summary we make the following contributions:

- We investigate static precalculability of memory accesses in CUDA kernels, at the thread block granularity. To this end, we develop a static analyzer to parse one CUDA kernel at a time. This analyzer examines every array index in the kernel to determine if it is precalculable. A precalculable index is an index whose range of values can be decided prior to kernel launch, by knowing the thread block identifier. Otherwise, the index is non-precalculable. We investigate 16 benchmarks and show that the majority of indexes are in fact precalculable.
- We introduce a simple abstract form to encapsulate the static analyzer information. We introduce API calls to covey this information to hardware. The information represents the range of data assigned to each thread block as a parameter of thread block identifier. During runtime, hardware evaluates the identifier and precisely determines the range of data assigned to each thread block.
- We introduce a low-overhead hardware mechanism to store the encapsulated information, calculate the range of data assigned to each thread block, and load the data for thread blocks.
- We evaluate our hardware/software scheme, TELEPORT, using 12 benchmarks that have large number of precalculable indexes. We show TELEPORT's performance and development effort advantages are remarkable. We also show TELEPORT improves DRAM row locality. This row locality improvement is achieved while keeping the number of memory accesses as low as the baseline.

[3] 2 factor accounts for one store and one load.

```
__global__ void
  kernel(int* arg1, int* arg2, int sz)
{
  int bid = blockIdx.x;
  int tid = threadIdx.x+bid*blockDim.x;
  arg1[tid] = 2*arg2[tid] + arg2[bid];
}
```

(a) A sample CUDA kernel.

```
Phase #1: extract pointers from the kernel
1) arg1[]                    2) arg2[]

Phase #2: extract indexes which are referred in the kernel
1) arg1[] ->                 2) arg2[] ->
    tid                          tid
                                 bid

Phase #3: determine the static precalculability of each index
1) arg1[] ->
    tid = threadIdx.x + blockIdx.x * blockDim.x   [PRED]
2) arg2[] ->
    tid = threadIdx.x + blockIdx.x * blockDim.x   [PRED]
    bid = blockIdx.x                              [PRED]
```

(b) Static analyzer operations.

Fig. 2. An example to clarify the static analyzer operations.

The remainder of this paper is organized as follows. In Section 2, we investigate static precalculability of memory accesses in GPGPU workloads. In Section 3, we introduce TELEPORT. In Section 4, we overview experimental methodology. In Section 5, we investigate the performance and development effort advantages of TELEPORT. In Section 6, we estimate hardware overhead of TELEPORT. In Section 7, we review previous work. Finally, Section 8 offers concluding remarks.

2. Static precalculability of memory accesses

In this Section, we investigate static precalculability of memory accesses in GPGPUs. In particular, we investigate the static precalculability of memory accesses in CUDA kernels. We limit the analysis to global memory space as we found it to be the most contributing space to performance. We developed a static analyzer parsing one CUDA 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 to in the kernel body. Finally, the static analyzer examines static precalculability of the value of each index. Fig. 2 clarifies the three phases of our static analyzer. Below we explain our definition of static precalculability and report the findings of static analysis under different CUDA kernels.[4]

2.1. Static precalculability

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

Processing operators. We consider an index as non-precalculable 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 precalculable addresses.

Processing terms. If the index is not found non-precalculable in the previous step, then the analyzer examines the terms of the expression to check precalculability (either statically or quasi-static). The index is statically precalculable, if the terms have constant values (e.g. *a[0]* or *a[blockDim.x]* in CUDA). The index is quasi-static precalculable, 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 precalculable. The index is non-precalculable, if 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]*).

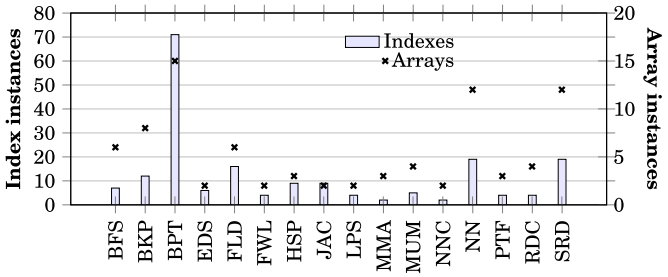

Fig. 3. The number of arrays and indexes identified by the static analyzer.

2.2. Results

Here we report static precalculability findings in 16 benchmarks, as measured by our static analyzer (for methodology refer to Section 4). Fig. 3 reports the number of unique global memory arrays found in the kernels. These arrays are declared as pointer variables in the kernel's arguments. The figure also reports the number of array indexes found in the kernel.[5] 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).

Fig. 4 complements Fig. 3 and reports the breakdown of array indexes into statically precalculable, quasi-static precalculable, and non-precalculable. As reported, none of the array indexes are found statically precalculable in the evaluated kernels and a significant portion of array indexes are found quasi-static precalculable. There are four subcategories of non-precalculable indexes: (i) induction variable dependence, (ii) indirect addressing, (iii) control dependent, and (iv) sophisticated operation. In NNC, NN, and RDC, non-precalculable array indexes depend on a loop induction variable. While we list them as non-precalculable in the breakdown, they may also be considered as quasi-static precalculable, if the boundaries of the loop are precalculable (which is the case

[4] Hereafter, by array we refer to every array identified in first phase of static analysis. Also array index is an array index extracted in the second phase.

[5] For indexes, we only report memory reads.

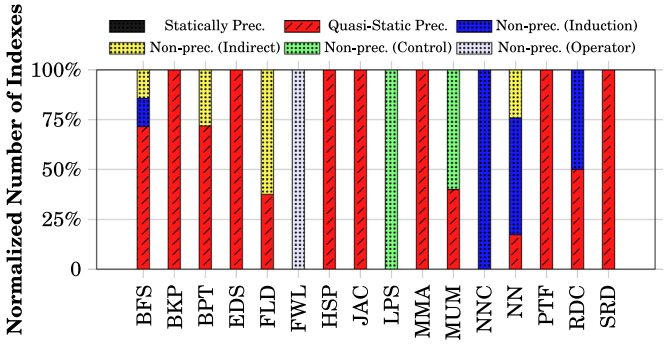

Fig. 4. Breakdown of array indexes into statically precalculable, quasi-static precalculable, and non-precalculable. Non-precalculable indexes either depend on induction variable (Induction), another memory load (Indirect), a control statement (Control), or use a sophisticated operator (Operator).

for these benchmarks[6]). For instance, in NN, 17 non-precalculable array indexes depend on an induction variable, whose range can be evaluated statically. In BPT and FLD, non-precalculable array indexes depend on a load from another array. In LPS and MUM, non-precalculable array indexes depend on control statements. In FWL, non-precalculable array indexes use the AND operator (&). In BFS, one of the non-precalculable array indexes depends on an induction variable and the other non-precalculable index depends on a load from another array.

Generally, a large portion of array indexes are found to be quasi-static precalculable (up to 100% in many benchmarks). Quasi-static precalculable 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 precalculated. In the rest of the paper, we build on this observation and propose TELEPORT hardware/software scheme to take advantage of the opportunity.

3. TELEPORT

3.1. Software side

The software side of TELEPORT includes (i) static analyzer (which we described in Section 2) and (ii) 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 precalculable by static analysis. The information guides the GPU to fetch 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 precalculating the demand addresses and introduce the API to pass that information. Secondly, we present an example to clarify the API usage.

3.1.1. API

If the index is marked as precalculable, static analysis evaluates the array index as an expression of (i) thread identifiers and (ii) constant terms. During the peiod of parsing this expression, we only evaluate thread IDs to determine the index value. Knowing the thread IDs 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 an index is the range between minimum and maximum values that index may have. Thread blocks have different IDs and, accordingly, the minimum and maximum varies for each thread block. Since thread block IDs are evaluated during runtime, the compiler passes the essential information to hardware and allows 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 evaluates 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 precalculable array index (statically precalculable or quasi-static precalculable) simplifies the index to an expression of constant values and only one runtime variable (*blockIdx*). In general, precalculable array index expression can be presented in the polynomial form of Eq. (1).

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

where a_i for $0 \le a_i \le n$ are life-time constant terms, *bIdx* is *blockIdx*, and n is the degree of polynomial. If an array index is constant (or statically precalculable) then a_i for $1 \le a_i \le n$ is zero. An array index is affine constraint of degree one if a_i for $2 \le a_i \le n$ is zero. Similarly, array indexes with affine constraint of higher degrees are possible. To simplify API and hardware support, here we limit TELEPORT to arrays having indexes with affine constrains of degrees one or lower (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 precalculable in the first degree or linear polynomial.)

For every precalculable 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. On the hardware side, this information is used to dynamically calculate 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 the 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 the memory controller. This fetches the entire range into a buffer on 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 fetched.

The proposed API prototype is shown in Listing 1. 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 TELEPORT. TELEPORT controller unit then calculates the range of data using the following linear equations:

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

[6] Here minimum and maximum values of the induction variables can be obtained from the loop statements statically. Then we can relax the induction dependency constraint and evaluate the indexes as quasi-static precalculable as the range of induction variables are evaluated statically.

Table 1
Output of static analysis determining the precalculable array indexes, affine index expressions of degree one, and the minimum and maximum value of index.

Array name	Line#	Extracted index expression	Index Status	min/max	
				a1	a0
mask	8				
nodes	11	(blockIdx.x* blockDim.x)+threadIdx.x	precalculable	blockDim.x	0/blockDim.x
nodes	12				
nodes	12				
cost	16				
edges	14	i	Induction	–	–
visited	15	edges[i]	Indirect	–	–

```
__host__ cudaError_t
cudaSetCTATracker(void *baseptr, size_t typesize,
  unsigned int minbidxp, unsigned int minbidyp,
  unsigned int minbidzp, unsigned int minoffset,
  unsigned int maxbidxp, unsigned int maxbidyp,
  unsigned int maxbidzp, unsigned int maxoffset);
```

Listing 1. Proposed API for passing per- thread block preload information.

```
1  __global__ void
2   Kernel(Node* nodes,
3     int* edges, bool* mask,
4     bool* visited, int* cost,
5     bool* over, int no_of_nodes)
6  {
7   int tid = blockIdx.x*blockDim.x + threadIdx.x;
8   if( tid<no_of_nodes && mask[tid]){
9    mask[tid]=false;
10   visited[tid]=true;
11   for(int i=nodes[tid].starting;
12    i<(nodes[tid].no_of_edges+nodes[tid].starting);
13    i++){
14    int id = edges[i];
15    if(!visited[id]){
16     cost[id]=cost[tid]+1;
17     mask[id]=true;
18     *over=true;
19    }
20   }
21  }
22 }
```

Listing 2. CUDA kernel in BFS.

$$max = baseptr + typesize \times (blockIdx.x \times maxbidxp + blockIdx.y \times maxbidyp + blockIdx.z \times maxbidzp + maxoffset) \quad (3)$$

3.1.2. Example

In this section, we explain the mechanism of passing information from the static analyzer to the GPU via runtime API. We overview a test case; BFS benchmark.

Listing 2 presents the kernel of BFS in CUDA, as available in Rodinia [7]. The static analyzer lists the following procedure arguments (or global memory pointers) as candidates for loading early: *nodes, edges, mask, visited, cost*, and *over*. It parses the kernel body and extracts all array reads from these arrays. Table 1 shows the list of identified read accesses. The first and second columns list array names and the corresponding lines of the code in Listing 2, respectively. The third column reports the index expression of the array access. Based on the terms and operators composing the expression, the fourth column reports whether the index is precalculable or not. 5 out of 7 indexes are marked as precalculable. Two array accesses are found non-precalculable since static analysis is unable to simplify the index expression to only thread identifiers and constant values, e.g. line #14. For precalculable array indexes, static analysis extracts the minimum and maximum possible values of the index. This returns the minimum and maximum, if the index expression is an affine expression of degree one.

```
1  cudaFlushCTATracker();
2  cudaSetCTATracker((void*)mask, sizeof(bool),
3    threads.x, 0, 0, 0,          // MIN
4    threads.x, 0, 0, threads.x); // MAX
5  cudaSetCTATracker((void*)visited, sizeof(bool),
6    threads.x, 0, 0, 0,          // MIN
7    threads.x, 0, 0, threads.x); // MAX
8  cudaSetCTATracker((void*)nodes, sizeof(Node),
9    threads.x, 0, 0, 0,          // MIN
10   threads.x, 0, 0, threads.x); // MAX
11 Kernel<<< grid, threads, 0 >>>(nodes,
12   edges, mask, visited,
13   cost, over, no_of_nodes);
```

Listing 3. API calls for passing static analysis information to GPU.

There are three unique precalculable array indexes in this example which are evaluated to single common index expression. According to the terminology of Eq. (1), *blockIdx.x* is $bIdx^i$, *blockDim.x* is a_1, and *threadIdx.x* is a_0 in this expression. Replacing minimum and maximum values of *threadIdx.x* in the expression returns the minimum and maximum values of the index expression. Minimum and maximum values of *threadIdx.x* are zero and $blockDim.x - 1$, respectively. This information is passed to GPU through *cudaSetCTATracker* API calls, one call per unique index.

Listing 3 shows the *cudaSetCTATracker* calls passing static analysis information to the GPU just before the kernel launch. *cudaFlushCTATracker* informs the GPU to flush and clear prior information of the CUDA stream on preload table, indicating a new kernel is about to be launched. Subsequent *cudaSetCTATracker* invocations pass loading information for one continuous data chunk at a time. In Listing 3, there are three *cudaSetCTATracker* calls to specify the data region for *mask*, *visited*, and *nodes*. For each call, the first argument specifies the base address of the array. The second argument passes the size of each array element in bytes. Next eight arguments pass *blockIdx* products and offsets so the GPU can calculate the minimum and maximum addresses for each thread block at runtime. As stated earlier, the GPU calculates the minimum and maximum addresses using Eqs. (2) and (3).

3.2. Hardware side

Below we explain hardware modifications of TELEPORT.

3.2.1. Preload table

The proposed API calls allow the compiler (or optionally the programmer) to pass the preload information to hardware. Hardware uses this information to dynamically load 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 preload table. Each row of preload table stores the information passed by a single API call. Each row registers base pointer, data type size, minimum products, and maximum products.

3.2.2. TELEPORT controller unit

The proposed controller unit is shared among all thread blocks and resides in the thread block dispatching unit. This unit reads every valid row of preload table to load the data region associated with the ready-for-dispatch thread blocks. Upon dispatching every thread block, the controller reads each row of the preload table, calculates the boundary of data region of the row using Eqs. (2) and 3, and issues a burst of requests per row to load the data region. Requests are issued from the thread block dispatcher unit to the memory controllers. In each request, the controller 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 loaded data is stored in buffer.

3.2.3. Data buffer

TELEPORT extends GPU cores with a logical buffer to store the data loaded via the TELEPORT controller unit. The buffer stores the loaded data for outstanding thread blocks of the core. The buffer is also responsible to count the number of received packets for each newly scheduled yet-stalled thread block and signal the warp scheduler to activate the thread block once all the packets from the TELEPORT controller unit are received at the core's end. Physically, this buffer can be a dedicated cache or any of the already available caches on the core (e.g. data cache or software-managed cache). In this work, we use the unused space of the existing CUDA shared memory as the buffer. An extra tag array is maintained along the shared memory to create a set-associative cache out of shared memory. The LRU cache replacement policy is used when needed[7] We investigate performance under TELEPORT for various cache configurations in Section 5.3. We also investigate the impact of having an ideal implementation of the buffer in Section 5.2.

3.2.4. Load concurrency

There might be few instructions ready to execute between the time that it takes for TELEPORT to load the entire thread block's data and when a thread actually demands the data. The GPU core can execute these instructions (in parallel to the pending memory requests issued by TELEPORT's controller unit) and hide the TELEPORT's loading delay. However, since overlapping the thread block progress and TELEPORT's loading increases the hardware complexity, we decided to simply stall the entire thread block until loading is completed. Later, in Section 5.2, we show the performance potential behind interleaving the execution of thread blocks and loading.

4. Experimental methodology

Modeling TELEPORT. We develop a static analyzer that highlights precalculable array indexes. We implemented the static analyzer standalone in Python. Analyzer parses one kernel at a time and identifies all array accesses and determine the static predictability state of every array index within the kernel. This provides essential input for injecting cudaSetCTATracker calls. While this can be accomplished automatically by the compiler, currently we manually reform this information to the proposed API calls and inject them to the benchmarks' source code. We use GPGPU-sim 3.2.2 [2] for modeling both hardware and software sides of TELEPORT. We model a hardware similar to NVIDIA GTX 480 as the baseline GPU of this study [26]. Simulation details are listed in Table 2. Warp scheduler and cache replacement policies are fixed and the same for both TELEPORT and the baseline.

Table 2
GPGPU-sim configurations for modeling GTX 480.

GPU chip	
GPU cores	15
Memory controllers	6
Sub partition / memory controller	2
GPU core	
L1 Cache	16KB, 32 sets, 4-way
Shared Memory	48KB, 32 banks
# of Threads	1536
Maximum concurrent thread blocks	8
# of registers	32,768 32-bit
Warp scheduler	gto
Memory controller	
L2 Cache / sub partition	64KB, 64 sets, 8-way
DRAM scheduler	FR-FCFS

Table 3
Benchmark details.

Abbr.	Name	Source
BFS	Breadth First Search	Rodinia
BKP	Backpropagation	Rodinia
BPT	B+ Tree	Rodinia
EDS	Edge detection (Sobel filter)	Third-party
FLD	Fluide Animate	PARSEC
FWL	Fast Walsh Hadamard Transform	NVIDIA SDK
HSP	Hotspot Simulation	Rodinia
JAC	Jacobi Iteration	PGI Compiler
LPS	Laplace 3D	GPGPU-sim
MMA	Matrix-matrix add	Third-party
MUM	MUMmerGPU	GPGPU-sim
NNC	Nearest Neighborhood	Rodinia
NN	Neural Network	GPGPU-sim
PTF	Pathfinder	Rodinia
RDC	Reduction	NVIDIA SDK
SRD	Speckle Reducing Anisotropic Diffusion	Rodinia

Benchmarks. Table 3 lists the benchmarks we used in this work. We used 16 benchmarks from Rodinia [7], GPGPU-sim [2], NVIDIA GPU Computing SDK [28], PGI Compiler [40], PARSEC [4], and applications that we wrote (edge-detection by sobel filter (EDS) and matrix-matrix add (MMA)) in this work. We selected these 16 benchmarks as they are not merely compute-bounded and show tangible performance improvement under the ideal zero-latency memory machine. We run all the benchmarks to completion. We model an ideal machine by assuming a perfect L1 cache which has a hit rate of 100%. Hereafter we limit evaluations to the 12 benchmarks that show strong precalculability in Section 2.2 (excluding FLD, FWL, LPS, and MUM benchmarks).

Evaluations. We measure performance in execution time (clock cycles) when comparing different implementations of the same benchmark (whenever different implementations come with unequal number of instructions.) Otherwise, we use IPC (instructions per cycle) as the performance metric. We assume a 64-entry preload table and a 48KB 96-way associative cache tag for TELEPORT, unless stated otherwise.

[7] We leave investigation of alternative replacement policies to future work.

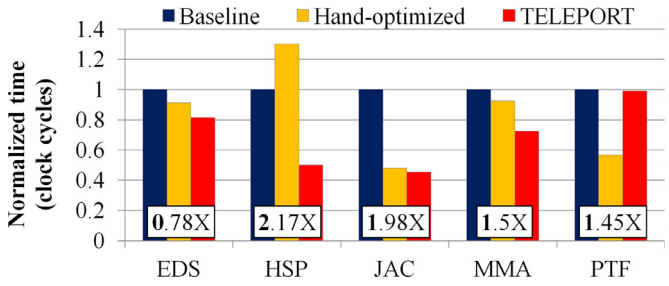

Fig. 5. Occupancy is 100% in all benchmarks, except Hand-written HSP, which is 50%. The numbers below the bar group show the ratio of dynamic instructions under Hand-written over TELEPORT.

5. Experimental results

In Section 5.1, we present performance improvements and development effort savings under TELEPORT's baseline configuration. In Section 5.2, we use machine models to investigate performance advantages, if TELEPORT hardware is optimized toward an ideal machine. In Section 5.3, we investigate the sensitivity of our findings under various hardware alternatives. In Section 5.4, we compare runtime of TELEPORT and a hardware prefetching scheme [19].

5.1. Baseline configuration

In this Section, we split the benchmark set of this study into two sets. We use five of our benchmarks to investigate both performance and development effort of TELEPORT. This study is presented in Section 5.1.1. Performance of remaining seven benchmarks are evaluated in Section 5.1.2 (here TELEPORT uses the unused space of shared memory as the buffer.)

5.1.1. Shared memory

We investigate performance and development effort advantages of TELEPORT, comparing three implementations of the benchmarks (comparing TELEPORT to two other implementations):

- Baseline: This implementation does not use software-managed cache. This should be prone to inefficient memory accesses.
- Hand-written: This implementation exploits software-managed cache to minimize off-chip memory accesses.
- TELEPORT: Built on top of Baseline, static analyzer parses the kernel and extracts precalculable array accesses. Using this information, *cudaSetCTATracker()* calls are injected before the kernel launch call to set preload table in hardware.

We limit evaluations of this section to EDS, HSP, JAC, MMA, and PTF benchmarks. It should be noted that evaluating each benchmark requires excessive amount of development effort to develop a Hand-written implementation. Therefore we limit our study to the five benchmarks listed above. Table 5 compares development effort of TELEPORT to Hand-written. We measure development effort in code lines. As reported, development effort improvements range from 1.46X to 3.4X. Below, we investigate the performance aspect.

Fig. 5 compares execution time of these benchmarks under Baseline, Hand-written, and TELEPORT implementations. Baseline and TELEPORT execute the exact same kernel code. The number below the bar group reports the ratio of dynamic instructions saved by Baseline and TELEPORT, compared to Hand-written.

As shown, the proposed approach consistently improves performance compared to Baseline. This improvement is minor under PTF and significant in other benchmarks.

Generally, TELEPORT has four performance advantages over Hand-written implementation. Firstly, it executes less number of *dynamic instructions*, since TELEPORT controller unit removes explicit read/writes from the shared memory space. Secondly, the controller unit issues the burst of memory requests in advance, effectively lowering average *memory access latency*. Thirdly, the controller unit issues requests for the same DRAM row back-to-back, potentially improving *DRAM row locality*. Finally, TELEPORT delivers higher *GPU core occupancy* (compared to Hand-written), since it does not demand allocating shared memory space statically (unlike Hand-written). Below we discuss each benchmark separately.

Under **EDS**, TELEPORT outperforms both Baseline and Hand-written. Most of the speedup comes from memory latency improvements. The input data for the entire thread block are fetched altogether, adequately earlier than the real demand. We found that DRAM row locality of TELEPORT is 1.9X and 2.7X greater than Hand-written and Baseline, respectively. Meanwhile, Hand-written executes lower dynamic instructions than TELEPORT. This might seem ironic since Hand-written executes more load/stores from/to shared memory. This is explained by observing the kernel code of these two implementations. Comparing kernel code of TELEPORT (and also Baseline) to Hand-written, the latter removes large number of logical and control-flow instructions (Hand-written returns dark pixel from shared memory, instead of checking the boundary and assuring the index falls within the range of the tile in global memory, as Baseline and TELEPORT do).

Under **HSP**, each thread block reads two 2D input tiles and writes one 2D output tile. Hand-written performs slower than other implementations as its occupancy is fairly low (50%), limited by the thread block registers usage. Compiler uses extra registers for loading/storing from/to shared memory. Also TELEPORT has advantages in executing lower dynamic instructions than Hand-written (by 2.17X).

Under **JAC**, TELEPORT delivers 2.2X speedup over Baseline. TELEPORT also outperforms Hand-written by 6% for executing 1.98X lower dynamic instructions.

Under **MMA**, Hand-written and TELEPORT both improve the efficiency of memory accesses using software-managed cache. In MMA, every thread works on 12 bytes of data (three 4-byte words; two input words and one output word). Assuming 256 threads per block, every thread block requires 3KB of data. Every GPU core runs 6 thread blocks, demanding 18KB in total. This is below the L1 cache capacity (16KB). Hand-written and TELEPORT use shared memory space and decrease this demand and prevent early evictions and improve the performance. Comparing these two shared memory version, Hand-written executes 50% more dynamic instructions. Although these extra instruction only involve two loads from shared memory and two stores to shared memory (plus shared memory addressing calculations), this degrades performance by a significant amount since the instruction sequence of threads is relatively short. Table 4 compares the instruction mix of TELEPORT and Hand-written versions of MMA. As reported, Hand-written kernel includes 54% more static instructions: 2 extra loads (from shared memory space), 2 extra stores (to share memory space), 1 thread block sync, and 9 other instructions for shared memory addressing. (Notice that static PTX sequence will be translated to SASS during runtime and SASS determines dynamic instructions.)

Under **PTF**, Baseline and TELEPORT perform close. Hand-written improves performance of Baseline through algorithmic modifications. PTF is 1D stencil kernel iterating for 36 times. Hand-written reduces the total number of iterations through ghost zone optimizations [24]. This amortizes several operations in one kernel launch, which significantly improves performance.

5.1.2. Shared memory free

Fig. 6 compares performance of TELEPORT to Baseline under BFS, BKP, BPT, NN, NNC, RDC, and SRD. Significant performance

Table 4
Comparing instruction mix of TELEPORT and Hand-written version of MMA benchmark (instructions are static PTX instructions).

Instruction	TELEPORT	Hand-written	description
ld	6	8	load from global/shared/param spaces
bar	0	1	thread block sync
cvt	3	5	datatype conversion
mov	4	6	move
st	1	3	store to global/shared
add	7	10	addition
mul	4	6	multiply
exit	1	1	terminate thread
Total	26	40	sum of all instructions

Table 5
Comparing development effort of TELEPORT to Hand-written shared memory version. Development effort is measured in code lines.

	TELEPORT	Hand-written	Improvement
EDS	13	19	1.46
HPS	21	51	2.43
JAC	8	23	2.88
MMA	4	9	2.25
PTF	10	34	3.4

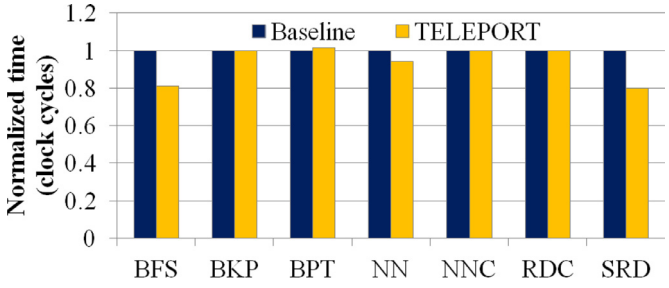

Fig. 6. Comparing performance of TELEPORT and Baseline.

improvement can be seen under BFS and SRD. TELEPORT slightly impacts performance of BKP, BPT, NN, NNC, and RDC. BKP, NN, and NNC are not memory-bounded kernels, as we found ideal zero-latency memory improves performance of these benchmarks by less than 20%. For BPT and RDC, TELEPORT is not effective in improving performance, since precalculable memory accesses account for a small share. In the next section, we investigate these benchmarks in detail using machine models.

5.2. Machine models

TELEPORT's performance is prone to two limiting overheads. The first overhead is excessive memory bandwidth usage (caused by loads from TELEPORT controller unit). This may increase average memory access latency since loading data tiles may slowdown the memory subsystem. The second overhead is stalling thread blocks till the data is completely loaded. This may harm performance if there are not enough concurrent thread blocks to hide the latency.

In this section we introduce three machine models to isolate and investigate the effect of these overheads. We also use machine models to investigate (i) the share of memory accesses that are covered by TELEPORT, (ii) DRAM row locality, and (iii) total number of DRAM accesses. For the evaluation that follows, we assume 64-entry preload table and an ideal unlimited fully-associative data buffer. The machine models:

- **I-Machine** is an ideal implementation resolving both overheads. First, no memory request is issued by TELEPORT controller unit, so TELEPORT does not impact the memory bandwidth. Second, the controller unit instantly loads the data of each thread block into the destination buffer, implementing zero-latency TELEPORT. Hence, thread blocks are not stalled till the load completes. 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 loads from controller unit but assuming a zero-latency TELEPORT. The controller unit issues a burst of memory requests to load 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, under real bandwidth restrictions.
- **R-Machine** is the non-ideal implementation modeling both overheads; the controller unit memory bandwidth usage and stalling thread blocks until completion of the TELEPORT loads. The only difference between this machine and the realistic implementation studied (in Section 5.1) is the ideal buffer employed by this machine has.

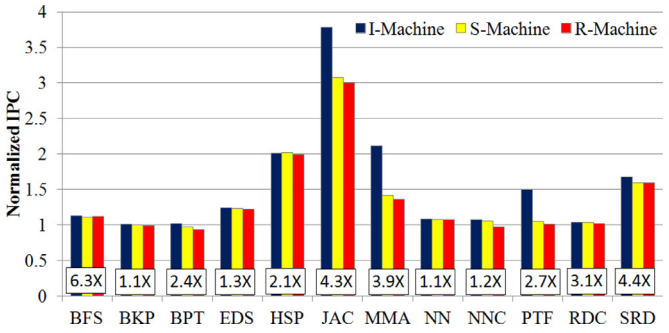

Fig. 7. Comparing performance of ideal TELEPORT machine models. Numbers are normalized to Baseline without TELEPORT. At the bottom of bars, the number indicates the speedup from zero-latency memory machine.

5.2.1. Performance

Fig. 7 compares performance of machine models under various workloads. Numbers are normalized to Baseline machine without TELEPORT nor software-managed cache usage. I-Machine reports the full performance potential behind TELEPORT, unleashed from all the overheads. Performance improvements range between 1% (in BKP) to 278% (in JAC) over the baseline.

Performance improvement is significant in BFS, EDS, HSP, JAC, MMA, PTF, and SRD. In these cases, the buffer covers a large portion of dynamic memory accesses (up to 90% in JAC). Moving from I-Machine to S-Machine, a portion of performance improvement is lost by modelling realistic memory bandwidth usage. On average, the memory bandwidth usage of TELEPORT impacts the performance of I-Machine by less than 6%. This is significant in PTF,

where the performance improvement of S- and R-Machine narrow down to 5% and 1%, respectively.

Slight performance improvement can be seen under NN and NNC. Under NN, all machines sustain the improvement of 8% which is close to the performance of ideal zero-latency memory machine (10% improvement, as reported by the label below the bar group). Under NNC, I- and S-Machine improve performance, however, R-Machine negates this. Since NNC runs under very low occupancy (8%; thread block size is 16 and 8 thread blocks are executed concurrently), stalling threads till TELEPORT loading completes 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 1.1X (10%, reported by the label below the bars in the figure). This leaves little room for improving the performance with TELEPORT. Under **BPT**, firstly, precalculable memory accesses account for a small portion of the total dynamic memory accesses. We found that only 15% of runtime memory requests are covered by TELEPORT. BPT has many non-precalculable memory accesses due to indirect dependencies[8] Secondly, the data type of the data is an *struct* of two 512-element arrays of *int* (nearly 4K per thread block). TELEPORT conservatively fetches entire bytes of these two arrays. BPT kernel, however, is divergent and does not access the whole array. One of the arrays is used thoroughly by all threads of the thread block and the accesses to the other array is control-dependent and therefore the array is partially accessed. Compared to Baseline, the extra memory accesses that the controller unit issues explains why TELEPORT performs poorly under BPT. In **RDC**, there are two kernels and 50% of array indexes are precalculable in total (as reported earlier in Fig. 4). Indexes from one kernel are all precalculable while indexes from the other kernel are all non-precalculable, due to control dependencies. Despite this seemingly promising share, the precalculable kernel contributes less than 20% to the total execution time of RDC. This leaves little room for improving performance.

Comparing S- to R-Machine, reveals the overhead of stalling thread blocks till loading completes. This overhead may harm performance in applications with low occupancy, e.g. BPT (66% occupancy) and NNC (8% occupancy). However, if the occupancy of an application is high, it can tolerate this overhead. For example, in SRD, the occupancy is 100% and R-Machine is able to improve performance by 59%. BFS, EDS, HPS, JAC, and MMA also have occupancies of 100% which allows tolerating the latency of stalling thread blocks and sustaining the performance improvement of the S-Machine.

5.2.2. Detailed analysis

The controller unit generates a burst of memory requests for a contiguous data region. This traffic pattern can improve DRAM row locality by mitigating row changes. However, this will not necessarily turn into a faster DRAM, since TELEPORT may simultaneously increase the total number of memory requests. This is the case when (i) the percentage of memory accesses covered by TELEPORT is low or (ii) the controller unit loads the entire data range while the thread block sparsely accesses the data. Generally, to have a faster DRAM with TELEPORT, we aim to (i) keep memory demand as low as the baseline, (ii) deliver high row locality at DRAM and (iii) deliver high hit rate at the TELEPORT buffer. Below we inspect these aspects of TELEPORT.

DRAM row locality. To increase exploitable locality, we use the following two techniques in hardware. First, the controller unit issues requests from single data region (or single entry of preload table or *cudaSetCTATracker()* call) back-to-back, avoiding early interleaving. Second, the memory controller prioritizes the controller unit requests over the requests coming from GPU cores. We found that the combination of these two techniques lowers DRAM row changes[9] Fig. 8 reports the average DRAM row locality of R-Machine and 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 is as low as Baseline in most cases (except BPT and JAC). This number for each benchmark is shown above R-Machine bars in Fig. 8, ranging from −5% (in NNC benchmark) to 134% (in BPT). In BPT, the higher memory reads are explained by the conservative approach which loads entire range of data, while the kernel sparsely accesses the data. In JAC, TELEPORT loads two extra rows and two extra columns for each tile to cover boundaries. Although all the data is not required by the threads, this covers all input data that thread block demands. The extra data fetches increase DRAM requests under TELEPORT by 11%, compared to Baseline.

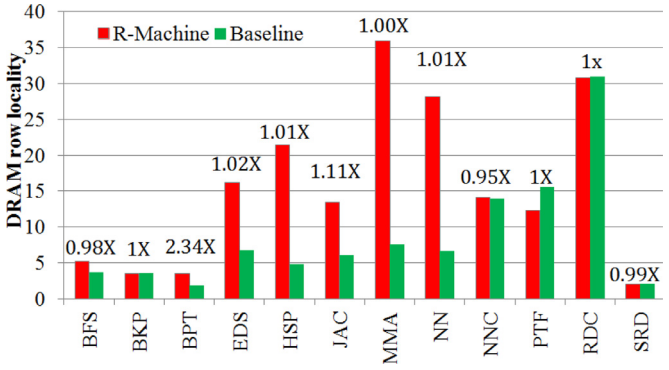

Fig. 8. Comparing average DRAM row locality of R-Machine to the baseline. Numbers above the bars indicate the ratio of the total DRAM accesses under R-Machine to the baseline.

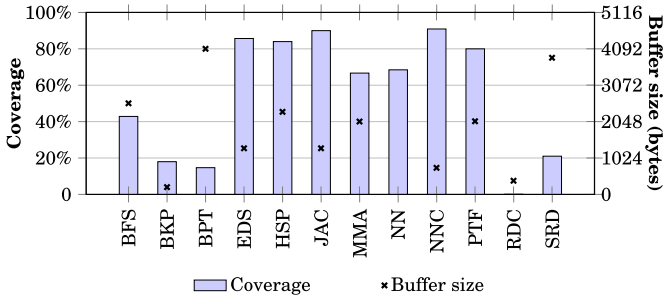

Fig. 9. Maximum buffer size for a thread block and the percentage of memory accesses that is captured by the buffer. Buffer size for NN is 10KB (not shown in the figure.).

Coverage of the data buffer. We define *coverage* as the percentage of the demand memory requests that hit in the TELEPORT buffer.[10] As reported in Fig. 9, coverage ranges from nearly zero in RDC to 90% in NNC. The figure also reports the maximum amount of data that is loaded into the buffer for a thread block in each benchmark. This suggests an upper-bound for the size of the buffer; one of the design parameters of TELEPORT. The demand from most of the benchmarks stays below 5 KB per thread block (except 10KB in NN). Assuming eight thread blocks per GPU core,

[8] Indirect dependency is explained in Section 2.2.

[9] In Section 5.3, we investigate alternatives.

[10] This includes both read or write requests.

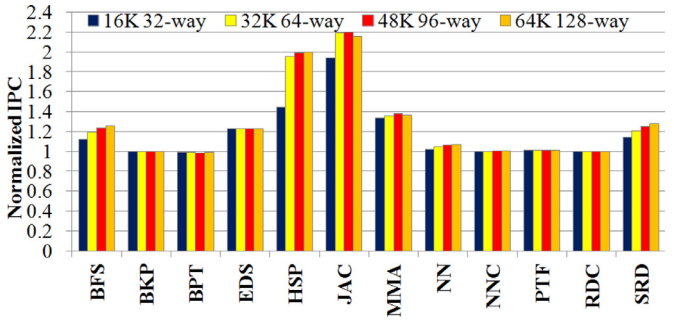

Fig. 10. Comparing performance of various buffer sizes, ranging from 16KB to 128KB. Numbers are normalized to Baseline without TELEPORT.

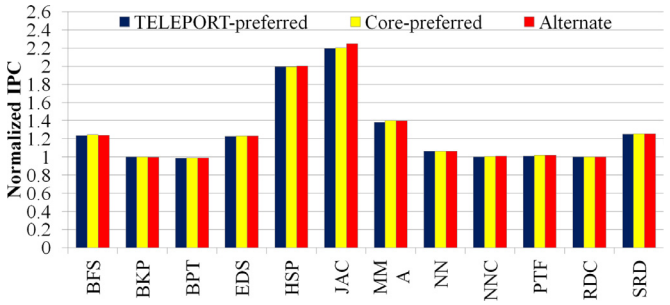

Fig. 11. Comparing performance of various network arbitration policies. Numbers are normalized to Baseline without TELEPORT.

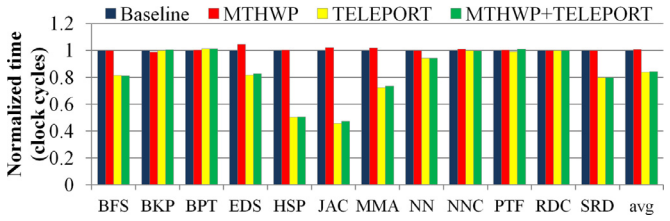

Fig. 12. Comparing performance of TELEPORT and MTHWP to the baseline.

the entire data ($8 * 5KB = 40KB$) would fit in the existing 48 KB shared memory in most cases.

5.3. Design alternatives

Below we analyze the sensitivity of our findings under various configurations for data buffer and also network policies for arbitrating the controller unit memory requests and core memory requests.

Data buffer. Fig. 10 compares performance of various buffer configurations, ranging from 16 KB (32-way associative) to 64 KB (128-way associative), while the line size is fixed to 128-byte. As shown, 32KB or 48KB of cache is large enough in most cases. Increasing the capacity and associativity improves performance. The optimal cache size large enough to maximize performance depends on the benchmark (specifically, size of the loaded data per thread block as shown in Fig. 9). For instance, while a 16 KB cache is large enough for EDS, 64 KB cache is not large enough for SRD. SRD is a heavily memory-bounded benchmark which comes with significant performance improvement potential under TELEPORT. SRD demands a very large amount of data to be loaded by the controller unit which requires large cache resources. To maximize performance for a specific cache size, the programmer can tune cudaSetCTATracker calls and maintain the total data size below the hardware capacity. If the fetched data's size exceeds the buffer size, then lines are evicted as suggested by a simple LRU replacement policy. Sophisticated replacement policies, e.g. coordinated with the thread-block pace, can be used to maximize performance and we leave this as future work.

The controller unit and core memory request arbitration. In the evaluations of this paper, we assigned higher priority to the controller unit requests than the core traffic during network arbitration. Here we compare performance of three different arbitration policies. *TELEPORT-preferred* always assigns higher priority to the controller unit requests. *Core-preferred* assigns higher priority to requests from cores. *Alternate* tries to establish a fair arbitration by switching the priority every cycle. Alternate proceeds with TELEPORT-preferred policy in even cycles and core-preferred policy in odd cycles. As we show in Fig. 11, the arbitration policies perform very close (less than 1% difference on average). In CUDA kernels, thread blocks usually write at the time near completion. Accordingly, TELEPORT-preferred may lower the throughput, since a large number of memory writes may stay behind the controller unit requests. This is the case in JAC and Alternate resolves this by time-sharing between the controller unit and cores traffic.

5.4. Compare with Hardware Prefetching

We have implemented Many-Thread Aware Hardware Prefetching (MTHWP) mechanism [19] to compare TELEPORT against a well-known GPU hardware prefetching approach. Fig. 12 compares runtime of MTHWP, TELEPORT, and the combined MTHWP+TELEPORT mechanisms to the baseline.

MTHWP is attached to L1 cache and has three prefetching tables (PWS, GS, and IP) to arbitrate and decide the next address to prefetch. PWS table is trained per warp per instruction to learn the stride. When few warps have the same stride on the same instruction, this entry is promoted to GS table to amortize the training overhead. In other words, GS table is similar to PWS table, but only trained per instruction. IP table is trained to prefetch an address for another warp while executing the current warp. Arbitration assigns highest priority to GS table. If GS table fails to predict an address to prefetch, priority is given to IP and then PWS tables. There is an adaptive throttling approach on top, throttling the prefetching when it is not beneficial. We configure MTHWP with 32, 32, and 128 entries per PWS, GS, and IP tables, respectively. We also set the throttling parameters to the numbers originally used by the authors [19].

As shown in Fig. 12, runtime advantages of MTHWP, either when deployed stand-alone or along with TELEPORT (MTHWP+TELEPORT), is very limited in our benchmark set. We explain this by the behavior of the benchmarks and structure of the prefetching tables. PWS table will have a chance to be trained and make accurate predictions if warps are executing the same instruction multiple times. In most of our benchmarks, each load instruction is only executed once (e.g. EDS, MMA, JAC, and NNC), preventing PWS from training. This automatically stops GS table, since GS table stores promoted entries from PWS. To have IP table making fair predictions, the target warp should have been dispatched on the same GPU core as the prefetcher (so the prefetched data can be fed into the warp from L1 cache). However, IP table does not have the information on where the target warp is dispatched. In this case, the prefetching might only help by returning the data from L2 cache (if the timeliness condition holds). The reasons above limit MTHWP in improving the runtime of our benchmarks here. Our findings are close to the evaluations in [29]. As shown in Fig. 12, MTHWP prefetcher has a negligible impact on the performance. While TELEPORT and MTHWP both have a similar hardware overhead, TELEPORT is a more efficient worth of hardware investment for runtime improvement.

6. Hardware complexity

We used CACTI 6.5 [25] in 40 nm to estimate the size of cache tag arrays and global preload table. We estimated the area of a 4-set 96-way tag array to be 0.009 mm^2. Multiplying this number by GPU cores, the total area overhead of cache tag arrays is 0.136 mm^2 (15*0.009). Assuming 21 bytes per entry for preload table (1-byte for datatype size, 4-byte for pointer, 8X 2-byte for min/max product/offsets) and 64 entries per table, preload table's size is nearly 1344 bytes. Using CACTI, we found the area of preload table 0.004 mm^2.

Also TELEPORT controller unit needs 6 16-bit integer multipliers and 6 16-bit integer adders to calculate minimum and maximum range of data for one thread block in single cycle. Using high-performance ALUs proposed by [23], each ALU occupies 0.071 mm^2 in 90 nm. This ALU occupies roughly 0.0144 mm^2 in 40 nm, scaled with 0.198 $((\frac{40nm}{90nm})^2)$ scaling factor. Assuming 12 ALUs of this kind, the controller unit area is 0.173 mm^2.

Compared to GTX480 die size (529 mm^2) [39], preload table, cache tag arrays, and the controller unit impose less than 1% overhead.

7. Related work

7.1. Prefetching

Cache fetch algorithms are either demand fetch or prefetch methods [36]. Demand requests are actual memory requests needed by the application. Prefetch requests speculate actual accesses and load them into the cache in advance. Prefetching mechanisms can solely be a hardware approach, not modifying the application code, or accept hints from the software. While prefetching mechanisms are independent from the application flow, they are still tightly related to and can be imagined as a concurrent process predicting memory accesses of the main process. Overviewing the prefetching literature since 1978[11], four major challenges should be addressed for prefetching to achieve performance improvment: prefetching timeliness (distance), accuracy, excessive bandwidth usage, and cache pollution. (i) timeliness: a prefetch request should be initiated with proper timing so it can lower the overall memory access latency. If initiated too early, data might be evicted in the cache before being used, returning no benefit. If initiated too late, and near the time that demand fetch happens, prefetching will not reduce memory latency. To address this, previous work propose to set the prefetching distance statically for the next Nth access [6,11] or adjust the distance adaptively [21,44]. (ii) accuracy: a prefetching mechanism should assure the addresses (of future demand fetches) are being predicted with high accuracy. This might be trivial for simple patterns (e.g. iterating through an array sequentially [16]), but involves a complicated compiler pass on sophisticated patterns (e.g. pointer-chasing patterns [22].) (iii) excessive bandwidth: prefetching increases the memory bandwidth usage of the application significantly. This can negate performance, if prefetching is wasting the bandwidth with wrong predictions [1]. Prior work suggest various heuristics to estimate prefetching usefulness and drop/filter useless prefetching out [8,9,18,20,37,45]. (iv) cache pollution: If the prefetched data share cache space with the demand fetch data, there is a risk that prefetching may evict lines from the working set of the application. To address this, using a dedicated prefetch buffer or cache partitioning are suggested [12,30,32].

Prefetching in GPU computing has also been investigated. Ryoo et al. [31] investigated software prefetching in the matrix multiplication test case. They found prefetching advantageous so long register pressure does not degrade occupancy. Lee et al. [19] evaluate several hardware prefetching mechanisms. Generally, they found that memory patterns are highly predictable. They also found out that a significant challenge stems from excessive memory bandwidth usage and lack of prefetch timeliness. They introduced a threshold-based heuristic to address these challenges. Jog et al. [15] report how integrating the warp scheduler into the prefetching mechanism unchains the real performance potential behind prefetching in GPGPUs. They show that conventional warp schedulers keep warps at a close pace, accessing nearby cache blocks during short intervals. If warps are scheduled far apart, then warps can potentially prefetch for each other. They show this careful warp scheduling combined with a simple prefetcher can yield significant speedup. Sethia et al. [33] used prefetching as a technique to improve energy-efficiency. They exploited prefetching to improve memory latency hiding. In addition, they lowered thread-level parallelism to save energy, while maintaining performance. Jeon et al. [13] found that the memory access pattern of the threads within the same thread block is strongly strided. In order to predict memory accesses of the thread block, their main challenge is to find the base address and stride value. To calculate base addresses, the warp scheduler prioritizes a single warp (from each thread block) to run ahead of the rest of the warp. The stride value is calculated by subtracting memory addresses that are issued from two successive warps upon executing the same instruction. Lakshminarayana and Kim [17] propose an approach for predicting the address of load instructions that depend on another load instruction. They tune the work for GPGPUs and address cache pollution by using spare registers' of the threads for prefetching peace requirements.

The approach taken by TELEPORT is fundamentally different from prefetching, as unlike prefetching, TELEPORT deals with *precalculating (absolute accuracy)* and not *speculation*. Accuracy is not a concern in TELEPORT as it precisely precalculates the range of data that will be accessed within each thread block. This is achieved in two steps: a software step that finds the range as a function of thread block identifier and a hardware step that evaluates the range by assigning values to thread block identifiers.

Prefetching and TELEPORT are orthogonal techniques and therefore can be employed in the same system. Under such circumstances, the prefetching mechanism can search TELEPORT's buffer before issuing a prefetch request. This lowers prefetching memory bandwidth usage as the prefetch request might have hit in the buffer already.

7.2. Development effort

Fang et al.[10] proposed ELMO APIs to lower OpenCL development effort in utilizing local memory. They explained challenges in introducing a high-level API for local memory fetch, writeback, and communication. They also investigated performance of various implementation alternatives. They found 1.3X to 3.7X performance improvement over the baseline (without local memory). Using the proposed API, programmer may save 5 to 81 lines of code (22 to 30 lines on average).

CUDA programmers control threads of thread blocks to optimize both data and computation patterns. This can fail under scenarios where optimizing data and computation patterns simultaneously is not possible. CudaDMA API [3] aims to improve both performance and productivity by decoupling data and computation patterns. CudaDMA allows splitting thread block into compute and DMA warps. DMA warps manage the movement of data between shared memory and DRAM and compute warps are responsible for computation.

[11] We refer readers to study [35] for work prior to this.

Silberstein et al. [34] propose an efficient solution to sum-products problem on GPUs. They characterize the problem as memory-bounded and find software-managed cache essential for achieving high performance on GPU. The challenge is that the memory pattern of the problem is not known up until run-time. They suggest a CPU-side preprocessing step to compensate and dynamically adapt to the dataset. The preprocessing step describes the pattern in metadata and then this metadata is passed to the GPU to configure the software-managed cache. Our approach is different in three ways: Firstly, our approach is not application specific. Secondly, building on top of the baseline implementation that is not using software-managed cache, our approach does not impose any development effort to the programmer (while their solution requires explicit implementation of data fetch, retrieve, and replacement by the programmer.) Finally, while their approach uses CPU time for preprocessing at runtime, our approach statically analyze the GPU kernel at compile-time.

7.3. DRAM efficiency

Yuan et al. [43] studied DRAM row locality in GPGPUs. They found that the memory traffic generated by cores has a very high row locality. However, these requests are reordered on the network on chip (NoC) that connects GPU cores to memory controllers. Built on this observation, they suggest NoC optimizations to deliver a performance close to that of complex DRAM schedulers.

Jog et al. [14] observed that when a DRAM row is open, sooner or later, most columns are read from the row. They suggest prefetching more columns than the demand (adaptively set between 8 to 16) into the L2 cache when a row is open. They show that while this can increase the latency of demand fetch stream, the overall impact on performance is positive as row-conflicts are reduced.

8. Conclusion

CUDA programmers exploit shared memory space to reduce off-core traffic. Although shared memory may deliver huge performance improvement, it imposes significant development effort. In this paper, we proposed TELEPORT as a hardware/software scheme for addressing performance and productivity in GPGPUs. TELEPORT is motivated by our observation on the precalculability of memory accesses in CUDA kernels. We presented our motivation that a large share of memory accesses in CUDA kernels is statically precalculable, at the thread block granularity. This means that data tiles assigned to thread blocks can be exactly determined by knowing thread block identifiers. On average, and compared to hand-written programs, TELEPORT improves performance by 32% and lowers development effort by 2.5X.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.micpro.2018.09.004.

References

[1] A.R. Alameldeen, D.A. Wood, Interactions between compression and prefetching in chip multiprocessors, in: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007, pp. 228–239, doi:10.1109/HPCA.2007.346200.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, Analyzing cuda workloads using a detailed gpu simulator, ISPASS 2009, 2009, doi:10.1109/ISPASS.2009.4919648.

[3] M. Bauer, H. Cook, B. Khailany, Cudadma: optimizing gpu memory bandwidth via warp specialization, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, in: SC '11, ACM, New York, NY, USA, 2011, pp. 12:1–12:11, doi:10.1145/2063384.2063400.

[4] C. Bienia, Benchmarking Modern Multiprocessors, Princeton University, 2011 PhD Dissertation at.

[5] Bill Dally, Challenges for future computing systems, Keynote speech at The 10th HiPEAC. Available: http://www.cs.colostate.edu/~cs575dl/Sp2015/Lectures/Dally2015.pdf(2015).

[6] D. Callahan, K. Kennedy, A. Porterfield, Software prefetching, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS IV, ACM, New York, NY, USA, 1991, pp. 40–52, doi:10.1145/106972.106979.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, IISWC 2009, 2009, doi:10.1109/IISWC.2009.5306797.

[8] E. Ebrahimi, O. Mutlu, C.J. Lee, Y.N. Patt, Coordinated control of multiple prefetchers in multi-core systems, in: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO 42, ACM, New York, NY, USA, 2009, pp. 316–326, doi:10.1145/1669112.1669154.

[9] E. Ebrahimi, O. Mutlu, Y.N. Patt, Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems, in: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, 2009, pp. 7–17, doi:10.1109/HPCA.2009.4798232.

[10] J. Fang, A. Varbanescu, J. Shen, H. Sips, Elmo: A user-friendly api to enable local memory in opencl kernels, in: Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on, 2013, pp. 375–383, doi:10.1109/PDP.2013.61.

[11] I. Ganusov, M. Burtscher, Future execution: a hardware prefetching technique for chip multiprocessors, in: 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), 2005, pp. 350–360, doi:10.1109/PACT.2005.23.

[12] B.S. Gill, D.S. Modha, Sarc: Sequential prefetching in adaptive replacement cache, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, in: ATEC '05, USENIX Association, Berkeley, CA, USA, 2005. 33–33

[13] H. Jeon, G. Koo, M. Annavaram, Cta-aware prefetching for gpgpu, Computer Engineering Technical Report CENG-2014-08, 2014.

[14] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A.K. Mishra, M.T. Kandemir, O. Mutlu, R. Iyer, C.R. Das, Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance, SIGPLAN Not. 48 (4) (2013) 395–406, doi:10.1145/2499368.2451158.

[15] A. Jog, O. Kayiran, A.K. Mishra, M.T. Kandemir, O. Mutlu, R. Iyer, C.R. Das, Orchestrated scheduling and prefetching for gpgpus, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, in: ISCA '13, ACM, New York, NY, USA, 2013, pp. 332–343, doi:10.1145/2485922.2485951.

[16] N.P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: Proceedings of the 17th Annual International Symposium on Computer Architecture, in: ISCA '90, ACM, New York, NY, USA, 1990, pp. 364–373, doi:10.1145/325164.325162.

[17] N.B. Lakshminarayana, H. Kim, Spare register aware prefetching for graph algorithms on gpus, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 614–625, doi:10.1109/HPCA.2014.6835970.

[18] C.J. Lee, O. Mutlu, V. Narasiman, Y.N. Patt, Prefetch-aware dram controllers, in: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO 41, IEEE Computer Society, Washington, DC, USA, 2008, pp. 200–209, doi:10.1109/MICRO.2008.4771791.

[19] J. Lee, N.B. Lakshminarayana, H. Kim, R. Vuduc, Many-thread aware prefetching mechanisms for gpgpu applications, MICRO '43, 2010, doi:10.1109/MICRO.2010.44.

[20] F. Liu, Y. Solihin, Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors, in: Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, in: SIGMETRICS '11, ACM, New York, NY, USA, 2011, pp. 37–48, doi:10.1145/1993744.1993749.

[21] J. Lu, H. Chen, P.-C. Yew, W.-C. Hsu, Design and implementation of a lightweight dynamic optimization system, J. Instruction-Level Parallelism 6 (2004).

[22] C.-K. Luk, T.C. Mowry, Compiler-based prefetching for recursive data structures, SIGOPS Oper. Syst. Rev. 30 (5) (1996) 222–233, doi:10.1145/248208.237190.

[23] S. Mathew, M. Anders, B. Bloechel, T. Nguyen, R. Krishnamurthy, S. Borkar, A 4-ghz 300-mw 64-bit integer execution alu with dual supply voltages in 90-nm cmos, Solid-State Circuits, IEEE Journal of 40 (1) (2005) 44–51, doi:10.1109/JSSC.2004.838019.

[24] J. Meng, K. Skadron, Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus, in: Proceedings of the 23rd International Conference on Supercomputing, in: ICS '09, ACM, New York, NY, USA, 2009, pp. 256–265, doi:10.1145/1542275.1542313.

[25] N. Muralimanohar, R. Balasubramonian, N. Jouppi, Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO 40, IEEE Computer Society, Washington, DC, USA, 2007, pp. 3–14, doi:10.1109/MICRO.2007.30.

[26] NVIDIA Corp., GeForce GTX 480, Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications[accessed 2016-05-05] (2015).

[27] NVIDIA Corp., Cuda c programming guide, Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/[accessed 2016-05-05](2016).

[28] NVIDIA Corp., Cuda c programming guide, Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ [accessed 2016-05-05](2016).

[29] Y. Oh, K. Kim, M.K. Yoon, J.H. Park, Y. Park, W.W. Ro, M. Annavaram, Apres:

Improving cache efficiency by exploiting load characteristics on gpus, in: Proceedings of the 43rd International Symposium on Computer Architecture, in: ISCA '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 191–203, doi:10.1109/ISCA.2016.26.
[30] P. Reungsang, S.K. Park, S.-W. Jeong, H.-L. Roh, G. Lee, Reducing cache pollution of prefetching in a small data cache, in: Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on, 2001, pp. 530–533, doi:10.1109/ICCD.2001.955085.
[31] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.-Z. Ueng, J.A. Stratton, W.-m.W. Hwu, Program optimization space pruning for a multithreaded gpu, in: CGO '08, 2008, pp. 195–204, doi:10.1145/1356058.1356084.
[32] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P.B. Gibbons, M.A. Kozuch, T.C. Mowry, Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks, ACM Trans. Archit. Code Optim. 11 (4) (2015) 51:1–51:22, doi:10.1145/2677956.
[33] A. Sethia, G. Dasika, M. Samadi, S. Mahlke, Apogee: Adaptive prefetching on gpus for energy efficiency, PACT '13, 2013.
[34] M. Silberstein, A. Schuster, D. Geiger, A. Patney, J.D. Owens, Efficient computation of sum-products on gpus through software-managed cache, in: Proceedings of the 22Nd Annual International Conference on Supercomputing, in: ICS '08, ACM, New York, NY, USA, 2008, pp. 309–318, doi:10.1145/1375527.1375572.
[35] A.J. Smith, Sequential program prefetching in memory hierarchies, Computer 11 (12) (1978) 7–21, doi:10.1109/C-M.1978.218016.
[36] A.J. Smith, Cache memories, ACM Comput. Surv. 14 (3) (1982) 473–530, doi:10.1145/356887.356892.
[37] S. Srinath, O. Mutlu, H. Kim, Y.N. Patt, Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers, in: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007, pp. 63–74, doi:10.1109/HPCA.2007.346185.
[38] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, W.-M.W. Hwu, Parboil: a revised benchmark suite for scientific and commercial throughput computing, Center for Reliable and High-Performance Computing (2012), 2012.
[39] Tech PowerUp, Nvidia geforce gtx 480, Available: https://www.techpowerup.com/gpudb/268/geforce-gtx-480(2011).
[40] I. The Portland Group, Pgi accelerator compilers with openacc directives, Available: https://www.pgroup.com/resources/accel.htm (2018).
[41] V. Volkov, Better performance at lower occupancy, in: GPU Technology Conference 2010 (GTC 2010), 2012. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
[42] V. Volkov, J. Demmel, Benchmarking gpus to tune dense linear algebra, in: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, 2008, pp. 1–11, doi:10.1109/SC.2008.5214359.
[43] G.L. Yuan, A. Bakhoda, T.M. Aamodt, Complexity effective memory access scheduling for many-core accelerator architectures, in: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO 42, ACM, New York, NY, USA, 2009, pp. 34–44, doi:10.1145/1669112.1669119.
[44] W. Zhang, B. Calder, D.M. Tullsen, A self-repairing prefetcher in an event-driven dynamic optimization framework, in: Proceedings of the International Symposium on Code Generation and Optimization, in: CGO '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 50–64, doi:10.1109/CGO.2006.4.
[45] X. Zhuang, H. h. S. Lee, Reducing cache pollution via dynamic data prefetch filtering, IEEE Trans. Comput. 56 (1) (2007) 18–31, doi:10.1109/TC.2007.250620.

Ahmad received his PhD in Computer Engineering at University of Victoria, BC, Canada in August 2017. For completing his PhD, he worked on developing hardware/software optimizations for accelerators. He developed IPMACC framework for translating OpenACC applications to CUDA, OpenCL, and ISPC backends. The framework is publically available at GitHub.