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/

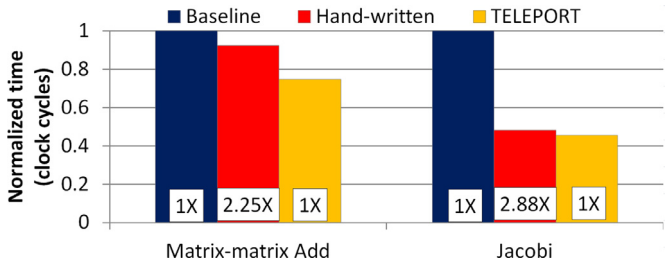

Fig. 1. Comparing three different implementations of Matrix-matrix Add and Jacobi iteration. Bars report kernel time and numbers below the bar indicate the development effort, normalized to Baseline (Effort is estimated by the number of lines of code.).

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.