EPiC Series in Computing

Volume 58, 2019, Pages 168–179

Proceedings of 34th International Conference on Computers and Their Applications

A Compiler-assisted locality aware CTA Mapping Scheme

Lifeng Liu[1], Meilin Liu[1], and Chongjun Wang[2]

[1] Department of Computer Science & Engineering, Wright State University, Dayton, Ohio, U.S.A.
[2] Department of Computer Science and Technology, Nanjing University, Nanjing, China

Abstract

General purpose GPU (GPGPU) is an effective many-core architecture that can yield high throughput for many scientific applications with thread-level parallelism. However, several challenges still limit further performance improvements and make GPU programming challenging for programmers who lack the knowledge of GPU hardware architecture. In this paper, we design a compiler-assisted locality aware CTA (cooperative thread array) mapping scheme for GPUs to take advantage of the inter CTA data reuses in the GPU kernels. Using the data reuse analysis based on the polyhedron model, we can detect inter CTA data reuse patterns in the GPU kernels and control the CTA mapping pattern to improve the data locality on each SM. The compiler-assisted locality aware CTA mapping scheme can also be combined with the programmable warp scheduler to further improve the performance. The experimental results show that our CTA mapping algorithm can improve the overall performance of the input GPU programs by 23.3% on average and by 56.7% when combined with the programmable warp scheduler.

1 INTRODUCTION

General purpose GPU (GPGPU) is an effective many-core architecture for computation intensive applications both in scientific research and everyday life [6, 15, 2, 9, 8].Compared to the traditional CPUs such as Intel X86 serial CPUs, GPGPUs have significant advantages for certain applications [6, 15].

First, the computation power of GPGPUs is much higher than the traditional CPUs. As reported by Nvidia [6], the single precision GFlops of GeForce 780 Ti GPU chip is higher than 5000, which is more than 10 times higher than the top Intel CPUs. In addition, the double precision GFlops of Tesla K40 GPU chip reaches nearly 1500, which is also nearly 10 times compared to the top Intel CPUs. GPUs achieve the high computation power by putting hundreds or even thousands of parallel stream processor cores into one chip. Compared to the CPUs which have very strong single thread processing power, the computation speed of a single thread on the GPU systems is very modest. However, the massively parallel structure of GPUs, which enables thousands of threads work in parallel, improves the overall computation power.

Second, the memory access throughput of the GPUs is much higher than the traditional CPUs. The memory bandwidth of Geforce 780 Ti GPU and Tesla K40 GPU is nearly 6 times higher than the top Intel CPUs.

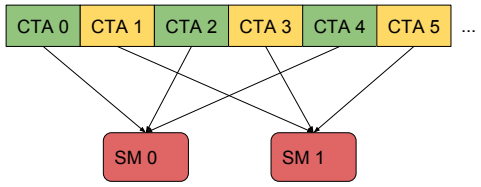

Figure 1: The default CTA mapping for 1D applications

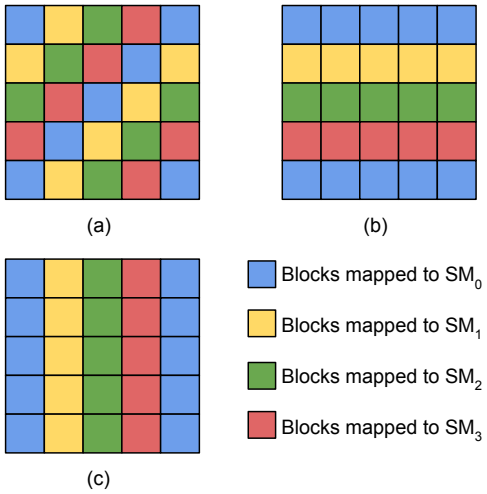

Figure 2: The CTA mapping.(a)Original mapping.(b)Mapping along the x direction (c)Mapping along the y direction

Third, compared to the supercomputers consisting of multiple CPU chips, GPUs could achieve the same computation power with lower power consumption and cost.

In addition, the development of GPU programming interfaces such as CUDA and OpenCL makes the programming on GPUs much easier [6, 5]. More and more developers have ported their applications from the traditional CPU based platforms to GPU platforms [3, 7, 13, 4].

Limited by the current DRAM technologies, accessing off-chip memory is very time consuming. In traditional CPUs, large L1 and L2 caches play an important role in bridging the speed gap between CPU cores and the off-chip DRAM memory. Both L1 and L2 caches are managed automatically by the hardware and are transparent to programmers. Programmers only need to consider a uniform continuous memory space. However, the memory hierarchy on GPU platforms is different. On each SM core of a GPU chip, a small high-speed shared memory is designed to cache small scale of frequently used data and the data shared among different threads in the same thread block.

When an SM has available CTA slots, a thread block will be selected and mapped to the SM in a round-robin manner [11]. The selection process does not consider the possible data locality or data reuses among the threads blocks mapped to the SM. Figure 1 shows a possible mapping result of a general scenario for a 1D applications. Assume the inter-CTA data locality exists among consecutive CTAs, the CTA mapping scheme shown in Figure 1 breaks the data locality, which increases the L1 cache footprint and degrades the overall performance.

The same problem can be observed for the 2D applications too. Assume we have four SMs and a grid of 5×5 thread blocks. The thread block mapping pattern with the original mapping strategy will break inter thread block data reuses along the x direction or the y direction as illustrated in Figure 2(a). The inter thread block data reuses can be preserved if we map the CTAs along the x direction as illustrated in Figure 2(b) because the CTAs having inter-warp data locality are all mapped to the same SM. Similarly, we can preserve the inter thread block data reuses along the y direction by mapping the CTAs along the y direction as illustrated in Figure 2(c).

In Chapter 4 of the dissertation [14], we analyze the intra-warp and inter-warp data reuses in the L1 data cache. The analysis is based on an assumption that the number of the warps is limited in the same thread block. However, in the real GPU systems the threads are grouped and executed as thread blocks or CTAs. The best size of the high priority warp group can be larger than the number of warps in a single thread block. If we prioritize the warps from different thread blocks without considering the data locality among them, the data locality might be broken and the overall performance would be degraded. To construct the high priority warp groups with the warps coming from the thread blocks with inter-CTA data reuses we must control the CTA mapping pattern according to the data reuse pattern among the thread blocks in the same kernel grid.

In [11], Lee et al. proposed a block CTA scheduling algorithm to preserve the inter-CTA locality. The block CTA scheduling algorithm can assign consecutive CTAs along the x direction to the same SM. However, without the inter-CTA reuse pattern detection mechanism, the algorithm cannot handle the inter-CTA reuses along the y direction. In addition, the algorithm cannot handle 2D applications.

In this paper, we present a compiler-assisted locality aware CTA mapping scheme that can enable the users to set the CTA mapping scheme before a GPU kernel is launched. The compiler-assisted CTA mapping scheme based on the polyhedron model can detect and control the CTA mapping pattern automatically. Then, we combine the CTA mapping scheme with the compiler-assisted programmable warp scheduler to further improve the performance of the GPU kernels.

The rest of this paper is organized as follows: in Section 2, we present the compiler-assisted locality aware CTA mapping scheme to detect the CTA mapping pattern. In Section 3, we illustrate how to combine the compiler-assisted CTA mapping scheme with the programmable warp scheduler. In Section 5, we evaluate the compiler-assisted locality aware CTA mapping scheme and the performance of the input benchmarks when they are optimized by the compiler-assisted CTA mapping scheme combined with the programmable warp scheduler. We conclude this paper in Section 7.

2 The CTA Mapping Pattern Detection

We design a CTA mapping control API to modify the CTA mapping strategy before a kernel is launched. The API will modify the value of a special register that controls the CTA mapping unit of each SM core. For performance-complexity trade-off, we only consider the three most common CTA mapping strategies: mapping along the x direction, mapping along the y direction and mapping in the round-robin manner. The default CTA mapping strategy is the round-robin mapping strategy.

Based on the CTA mapping API, we design a compiler-assisted locality aware CTA mapping scheme to insert the CTA mapping control APIs automatically based on the inter-CTA data reuse analysis. The basic rules of the compiler-assisted locality aware CTA mapping scheme is:

1. If there are inter thread block data reuses along the x direction, then the CTAs are mapped along the x direction.
2. If there are inter thread block data reuses along the y direction, then the CTAs are mapped along the y direction.
3. If there are inter thread block data reuses along both the x and the y directions, the CTAs are mapped along the direction that has larger reuse distance as the memory blocks can have more reuses along the direction having larger data reuse distance. For example, if the memory blocks accessed by a thread block can be reused by the following N thread blocks, then these memory blocks can be reused $N-1$ times during the execution of the GPU kernel. So the larger the N is, the more times the memory blocks can be reused.
4. Otherwise, use the round-robin CTA mapping strategy.

The inter thread block data reuses along the x direction can be formally defined as:
Inter-CTA data reuses along the x direction: During the execution process of a thread block, if there exists two memory accesses M and M' that meet all of the following conditions, then inter-CTA data reuses exist among the thread blocks along the x direction.

1. M and M' are issued by different thread blocks with the same thread block index in the y direction.
2. M' reuses the data in the L1 cache brought in by M.

The first constraint can be represented as

$$\begin{cases} 0 \leqslant bidy, bidy' < gdimy \\ 0 \leqslant bidx, bidx' < gdimx \\ 0 \leqslant tidy, tidy' < bdimy \\ 0 \leqslant tidx, tidx' < bdimx \\ bidy' = bidy \end{cases} \tag{1}$$

The second constraint can be represented as

$$\begin{cases} \vec{\alpha'}_i = \vec{\alpha}_i, i = 1...m-2 \\ \vec{\alpha'}_{m-1} = \beta' * B + \gamma' \\ \vec{\alpha}_{m-1} = \beta * B + \gamma \\ \beta' = \beta \\ -B + 1 \leqslant \gamma' - \gamma \leqslant B - 1 \\ 0 \leqslant \gamma' \leqslant B - 1 \\ 0 \leqslant \gamma \leqslant B - 1 \\ 0 \leqslant \beta' \\ 0 \leqslant \beta \end{cases} \tag{2}$$

Where B indicates the cache block size; β' and β indicate the cache block indexes of M' and M; γ' and γ indicate the offsets inside each cache block.
The target parameter that we are interested in is the reuse distance along the x direction:

$$\zeta_x = |bidx' - bidx| \tag{3}$$

Then, the problem of detecting inter thread block data reuses can be transformed into an integer linear programming problem:

$$\begin{array}{ll} max & \zeta_x \\ s.t. & \zeta_x = |bidx' - bidx| \\ & bidx', bidx \in G_x \end{array} \tag{4}$$

Where G_x is the polyhedron defined by (1) and (2). If the problem (4) has no solution or $\zeta_x = 0$, then it indicates that there is no inter thread block data reuses along the x direction.

Similarly, we can define inter-CTA data reuses along the y direction as:

Inter-CTA data reuses along the y direction: During the execution process of a thread block, if there exists two memory accesses M and M' that meet all of the following conditions, then inter-CTA data reuses exist among the thread blocks along the y direction.

1. M and M' are issued by different thread blocks with the same thread block index in the x direction.
2. M' reuses the data in the L1 cache brought in by M.

The first constraint is represented as

$$\begin{cases} 0 \leqslant bidy, bidy' < gdimy \\ 0 \leqslant bidx, bidx' < gdimx \\ 0 \leqslant tidy, tidy' < bdimy \\ 0 \leqslant tidx, tidx' < bdimx \\ bidx' = bidx \end{cases} \tag{5}$$

The second constraint is the same as (2). We can obtain the maximum inter thread block reuse distance along the y direction ζ_y in a similar way:

$$\begin{array}{ll} max & \zeta_y \\ s.t. & \zeta_y = |bidy' - bidy| \\ & bidy', bidy \in G_y \end{array} \tag{6}$$

Where G_x is the polyhedron defined by (5) and (2).

Algorithm 1 is presented to detect the CTA mapping direction. As illustrated in Algorithm 1, if there are multiple inter-CTA data reuses existing in a single GPU kernel, we record the maximum inter-CTA reuse distance along the x direction and the y direction separately (Line $1-5$). When there are inter thread block data reuses along both the x direction and the y direction, the CTAs are mapped along the direction that has larger reuse distance (Line $6-9$). Otherwise, the round-robin CTA mapping strategy would be used.

3 Combine the Programmable Warp Scheduler and the Locality Aware CTA Mapping Scheme

To take advantage of the inter thread block data locality and inter-warp data locality simultaneously, we combine the compiler-assisted programmable warp scheduler and the compiler-assisted locality aware CTA mapping scheme as illustrated in Algorithm 2.

The compiler-assisted locality aware CTA mapping scheme will not affect the intra-warp data reuse detection algorithm. However, we must modify the inter-warp data reuse detection

Algorithm 1: Detect the CTA mapping direction

Input: GPU kernel G
Output: *direction*, *maxResueDistance*

```
max_x = 0;
max_y = 0;
for each memory access pair M and M' in G do
    Get the reuse distance r_x along the x direction by solving problem 4;
    Get the reuse distance r_y along the y direction by solving problem 6;
    max_x = max(max_x, r_x);
    max_y = max(max_y, r_y);
end
if max_x==0 and max_y==0 then
    direction = round-robin;
else
    direction = max_y>max_x ? y : x;
    maxResueDistance=max(max_y, max_x);
end
```

Algorithm 2: Combine the CTA mapping scheme and the programmable warp scheduler

Input: GPU kernel G
Output: Optimized GPU kernel with both CTA mapping and programmable warp scheduler applied

```
Get CTA mapping direction through Algorithm 1;
Insert CTA mapping control API before G is launched;
for each loop L in G do
    Apply the Programmable Warp scheduling Algorithm [14] on L with the constrains
    modified by (7) and (8) to insert scheduler control instructions for L.
end
```

algorithm and the high priority warp group size detection algorithm. The constraints in Programmable Warp scheduling Algorithm [14] are modified by the extra constraints presented in (7) and (8). Also, we must remove the constraints of $w, w' < ceiling(bdimx * bdimy/32)$ because the warp IDs could exceed the thread block boundary. In addition, the check condition in Line 14 of Programmable Warp scheduling Algorithm [14] must be modified to "$\min(\xi_{min},\theta_{min}) < \#warpsInBlock * maxResueDistance$".

$$\begin{cases} bidy' = bidy & \text{If CTAs are mapped along the } x \text{ direction} \\ bidx' = bidx & \text{If CTAs are mapped along the } y \text{ direction} \end{cases} \tag{7}$$

$$\begin{cases} 32w \leqslant bdimx * tidy + tidx + (gdimx * bidy + bidx) * wpb < 32(w+1) \\ \quad \text{If CTAs are mapped along the } x \text{ direction or in the round-robin manner} \\ 32w \leqslant bdimx * tidy + tidx + (gdimy * bidx + bidy) * wpb < 32(w+1) \\ \quad \text{If CTAs are mapped along the } y \text{ direction} \end{cases} \tag{8}$$

Where $wpb = ceilling(bdimx * bdimy/32)$, which is the number of warps per thread block.

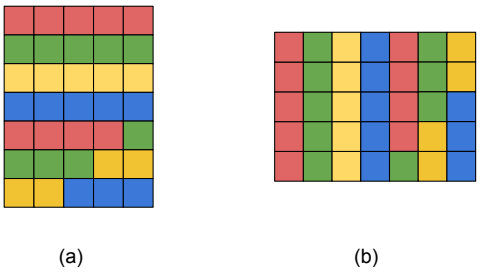

Figure 3: Balance the CTAs among SMs when mapping along the x direction (a) and the y direction (b)

The warp constraints for w' is modified in the same way.

4 Balance the CTAs Among SMs

We modify the CTA mapping units in GPGPU-sim to evaluate our compiler-assisted locality aware CTA mapping algorithm. The CTA mapping unit can be configured to map CTAs along the x direction, along the y direction or using the round-robin manner. The CTA mapping configuration can be set by the CTA mapping control API we developed, which is supported by the GPGPU-sim modified by us. The CTA mapping control API can be executed before the GPU kernel launch. To balance the number of CTAs assigned to an SM, and preserve the inter-CTA data reuses as much as possible when the mapping direction is selected, we use the following rules

1. Assign a whole row or column of CTAs to an SM as long as we can evenly distribute them among the SMs.
2. For the rest of the CTAs, we evenly distribute them along the x direction or the y direction.

The CTA mapping algorithm that follows these two rules is illustrated in Algorithm 3. In Algorithm 3, a, b, c, d and e are intermediate variables, *direction* is the CTA mapping direction that is detected by Algorithm 1. Figure 3 shows the mapping results of Algorithm 3 with $\#sms = 4$ for mapping along both the x direction and the y direction.

5 Evaluation

5.1 Evaluation Platform

We configure GPGPU-sim developed by Aamodt et al. [1, 2] to simulate the GTX480 GPU as the test platform to evaluate our CTA mapping technique for GPGPUs. The detailed configuration for the baseline GPU system is shown in Table 1. We use NVCC 4.2 to compile the output source code of our source-to-source compiler framework.

The benchmarks we selected to evaluate our CTA mapping algorithm are listed as follows:
(1) micro-benchmark: A simple GPU kernel designed to add neighboring blocks together. We can see that the same block in input will be reused among CTAs along the x direction.
(2) matmul: Block matrix multiplication kernel, whose input size is 4kx4k.
(3) conv: Two dimensional convolution algorithm, whose input size is 4kx4k.
(4) demosaic: Image demosaicing algorithm, whose input size is 8kx8k.

Algorithm 3: CTA mapping

```
Input: gdimx, gdimy, #sms, direction
Output: the CTAs mapped to each SM
1  if direction == x then
2      a = gdimy / #sms;
3      b = a * gdimx;
4      c = (gdimy - a) * gdimx;
5      d = (c + 1) / #sms + 1;
6      for each B = CTA(x, y) do
7          if y < b then
8              Assign B to SM core y / a;
           end
9          e = (x + y * gdimx) - b;
10         Assign B to the SM core e / d;
       end
   end
11 else if direction == y then
12     a = gdimx / #sms;
13     b = a * gdimy;
14     c = (gdimx - a) * gdimy;
15     d = (c + 1) / #sms + 1;
16     for each B = CTA(x, y) do
17         if x < b then
18             Assign B to the SM core x / a;
           end
19         e = (y + x * gdimy) - b;
20         Assign B to the SM core e / d;
       end
   end
21 else
22     assign the CTAs to the SMs in the round-robin manner;
   end
```

Module	Description
Number of SMs	15 (as Nvidia GTX480)
Number of integer processing units per SM	2
Number of floating point processing units per SM	1
SIMD width	32
Size of L1 data cache	16KB, 4 way associative
Size of L2 data cache	512KB, 8 way associative
Size of shared memory	48KB

Table 1: The baseline simulator configuration

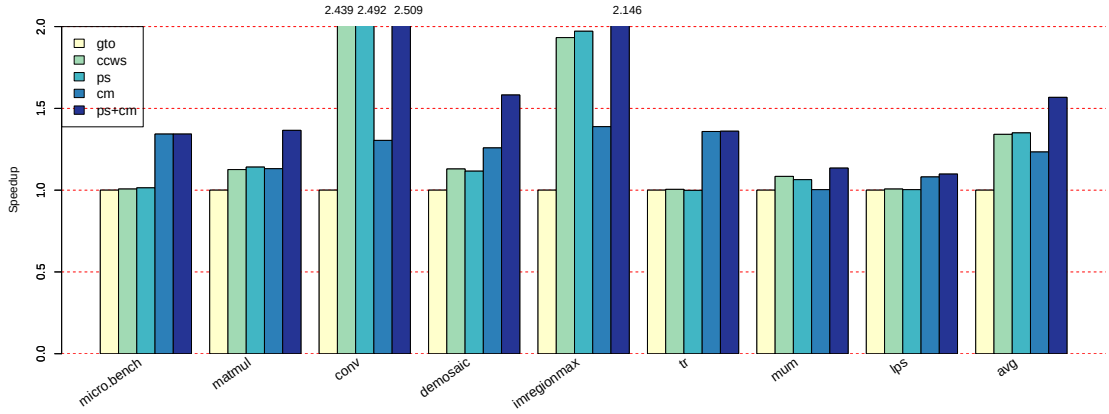

Figure 4: Speedups

(5) imregionmax:The algorithm to find the regional maximum for an image, whose input size is 4kx4k.
(6) tr: Matrix transpose algorithm, whose input size is 4kx4k.
(7) mum: Parallel local-sequence alignment program [2], whose input size is 50k.
(8) lps: A Laplace discretisation on a 3D structured grid [2], whose input size is 4M.

5.2 Experimental Results

To evaluate our compiler-assisted locality aware CTA mapping algorithm, we measure the performance of the GPU kernels optimized by the compiler-assisted locality aware CTA mapping algorithm (labeled as "cm" in Figure 4) and the performance of the GPU kernels optimized by the CTA mapping algorithm combined with the programmable warp scheduler discussed in Chapter 4 of the dissertation [14] (labeled as "cm+ps" in Figure 4). We compare the performance of the GPU kernels optimized by our compiler-assisted locality aware CTA mapping algorithm with the performance of the GPU kernels optimized by the GTO warp scheduler which is used as the baseline, the performance of the GPU kernels optimized by the CCWS warp scheduling algorithm proposed by Rogers et al. [17] and the performance of the GPU kernels optimized by the compiler-assisted programmable warp scheduler (labeled as "ps" in Figure 4).

The performance results of the GPU kernels are summarized in Figure 4. In Figure 4, the first column shows the normalized performance of the input benchmarks optimized by the GTO algorithm. The second column shows the normalized speedups of the input benchmarks optimized by the CCWS warp scheduling algorithm. The third column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted programmable warp scheduler. The fourth column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm. The fifth column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler.

As illustrated in Figure 4, our compiler-assisted locality aware CTA mapping algorithm can improve the performance of the selected benchmarks by 23.3% on average over the GTO warp scheduling algorithm. The GTO warp scheduler just considered the locality within each CTA, however, the inter thread block data reuses are ignored due to the round-robin CTA mapping

strategy that always assigns adjacent CTAs to different SMs.

Compared to the GTO warp scheduling algorithm, the CCWS warp scheduling algorithm and the compiler-assisted programmable warp scheduler can avoid self-evictions in the L1 cache and preserve intra-warp and inter-warp data reuses. The CCWS warp scheduling algorithm and compiler-assisted programmable warp scheduler improve the overall performance of the input benchmarks by 34.1% and 35.0% respectively on average over the GTO warp scheduling algorithm. However, The CCWS warp scheduling algorithm and the compiler-assisted programmable warp scheduler do not consider the data reuses among different CTAs, so they cannot further exploit the data reuses in the L1 cache.

The performance of the input benchmarks is improved by 56.7% on average when our compiler-assisted CTA mapping algorithm is combined with the compiler-assisted programmable warp scheduler, since the self-evictions in the L1 cache are avoided by limiting the total number of active warps and taking advantage of the inter CTA data reuses. In addition, our compiler-assisted CTA mapping algorithm can construct high priority warp groups across the CTA boundaries, which can completely hide long latency operations by feeding the warp scheduler with enough warps and taking advantage of the high data reuses in the L1 cache.

The benchmarks matmul, conv, demosaic, imregionmax and mum have both good intra-CTA and inter-CTA data locality, so the compiler-assisted programmable warp scheduler contributes most to the performance improvements (64% for matmul, 79% for conv, 55% for demosaic, 66% for imregionmax and 97% for mum) for these input benchmarks. While the rest of the benchmarks mainly obtained performance gain from the inter-CTA data reuses (100% for micro_bench, 99% for tr and 82% for lps). Both the compiler-assisted programmable warp scheduler and the CCWS warp scheduler cannot further improve the performance of these input benchmarks.

6 Related Work

In [12], Lee et al. proposed two optimized CTA scheduling algorithms to improve the performance of the input GPU kernels. The lazy CTA scheduling method can limit the total number of active CTAs assigned to each SM to improve the L1 cache hit ratio and reduce the resource competition, which is similar to the idea of the warp limiting algorithm introduced by the CCWS warp scheduler [17] when applied on thread blocks. They also proposed the block CTA scheduling strategy to take advantage of the data locality among different thread blocks. Compared to their work, we propose a compiler-assisted locality aware CTA mapping algorithm which uses a systematic way to control the CTA mapping pattern more accurately using the data reuse analysis based on the polyhedron model. In addition, the CTA limiting can also be performed in our framework by combining the compiler-assisted locality aware CTA mapping algorithm with the programmable warp scheduler, which can tune the number of active warps across the CTA boundaries.

Our compiler-assisted CTA mapping algorithm is also combined with the compiler-assisted programmable warp scheduler to exploit inter-CTA data reuses in addition to the intra-warp data reuses and the inter-warp data reuses, which exhibits more data reuses compared to the compiler-assisted programmable warp scheduling algorithms, such as the CCWS warp scheduling algorithm proposed by Rogers et al. [17] and the two-level warp scheduling algorithm proposed by Narasiman et al. [16]. Both of these two warp scheduling algorithms considered the data locality and the resource competition in the L1 cache among different warps on an SM, which can improve the overall performance by limiting the number of concurrent warps. However, just as we analyzed in this paper, random CTA assignment to SMs might break the data

locality and increase the L1 cache thrashing. Our proposed compiler-assisted CTA mapping scheme overcomes this drawback and further improves the overall performance.

In [10], the consecutive CTA mapping strategy is used to facilitate the design of optimized GPU memory prefetcher based on the two level warp scheduler proposed in [16]. Their prefetcher takes advantage of the spatial locality among consecutive CTAs assigned to each SM. However, only the benefit obtained from this CTA mapping strategy is analyzed and no inter-CTA data reuse analysis has been performed.

Yang et al. proposed a task scheduling algorithm considering data reuses in the cache, memory footprint and cache coherence for chip-multiprocessor systems [18]. They group tasks that have the maximum amount of data sharing into sharing groups. These tasks will be assigned to the same CPU core to maximize the data reuses in the L1 cache and minimize the cache coherence traffic, which is similar to the idea of our compiler-assisted CTA mapping algorithm that also assigns the thread blocks with data reuses to the same SM. The concept to enable the memory footprint fit in the shared cache is also similar to the warp limiting technique used in our compiler-assisted programmable warp scheduler. However, they did not apply the data reuse analysis and memory footprint estimation at compile time.

7 Summary

In this paper, we design a compiler-assisted locality aware CTA mapping scheme for GPUs to take advantage of the inter CTA data reuses in the GPU kernels. Using the data reuse analysis based on the polyhedron model, we can detect inter CTA data reuse patterns in the GPU kernels and control the CTA mapping pattern to improve the data locality on each SM. The compiler-assisted locality aware CTA mapping scheme can also be combined with the programmable warp scheduler to further improve the performance. The experimental results show that our CTA mapping algorithm can improve the overall performance of the input GPU programs by 23.3% on average and by 56.7% when combined with the programmable warp scheduler.

References

[1] Tor M. Aamodt and Wilson W.L. Fung. Gpgpu-sim 3.x manual. http://gpgpu-sim.org.

[2] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[3] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009.*, pages 44–54, Oct 2009.

[5] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide.* NVIDIA Corporation, 2007.

[6] NVIDIA Corporation. *NVIDIA CUDA (Computer Unified Device Architecture): Programming Guide, Version 7.5.* 2015.

[7] N. Devarajan, S. Navneeth, and S. Mohanavalli. Gpu accelerated relational hash join operation. In *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*, pages 891–896, Aug 2013.

[8] W.W.L. Fung, I. Sham, G. Yuan, and T.M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 407–420, Dec 2007.

[9] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. *SIGARCH Comput. Archit. News*, 39(3):235–246, June 2011.

[10] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[11] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271, Feb 2014.

[12] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271, Feb 2014.

[13] Jie Li, Vishakha Sharma, Narayan Ganesan, and Adriana Compagnoni. Simulation and study of large-scale bacteria-materials interactions via bioscape enabled by gpus. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12, pages 610–612, New York, NY, USA, 2012. ACM.

[14] Lifeng Liu. *An Optimization Compiler Framework Based on Polyhedron Model for GPGPUs*. PhD thesis, Wright State University, 2017.

[15] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, June 2010.

[16] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[17] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[18] Teng-Feng Yang, Chung-Hsiang Lin, and Chia-Lin Yang. Cache-aware task scheduling on multicore architecture. In *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, pages 139–142, April 2010.