International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 2, pages 253–271, July 2015

Automatic Optimization of Thread Mapping for a GPGPU Programming Framework

Kazuhiko Ohno, Tomoharu Kamiya, Takanori Maruyama
Department of Information Engineering, Mie University,
1577 Kurimamachiya-cho, Tsu, Mie, 514-8507, JAPAN

Masaki Matsumoto
Medical Engineering Institute, Inc.
3-141-1, Sakae-cho, Tsu, Mie, 514-0004, JAPAN

Received: February 15, 2015
Accepted: April 7
Communicated by Hidetsugu Irie

Abstract

Although General Purpose computation on Graphics Processing Units (GPGPU) is widely used for the high-performance computing, standard programming frameworks such as CUDA and OpenCL are still difficult to use. They require low-level specifications and the hand-optimization is a large burden. Therefore we are developing an easier framework named MESI-CUDA. Based on a virtual shared memory model, MESI-CUDA hides low-level memory management and data transfer from the user. The compiler generates low-level code and also optimizes memory accesses applying conventional hand-optimizing techniques. However, creating GPU threads is same as CUDA; the user specifies thread mapping, i.e. thread indexing and the size of thread blocks run on each streaming multiprocessors (SM). The mapping largely affects the execution performance and may obstruct automatic optimization of MESI-CUDA compiler. Therefore, the user must find optimal specification considering physical parameters. In this paper, we propose a new thread mapping scheme. We introduce new thread creation syntax specifying hardware-independent logical mapping, which is converted into optimized physical mapping at compile time. Making static analysis of array index expressions, we obtain groups of threads accessing the same or neighboring array elements. Mapping such threads into the same thread block and assigning consecutive thread indices, the physical mapping is determined to maximize the effect of memory access optimization. As the result of evaluation, our scheme could find optimal mapping strategies for five benchmark programs. Memory access transactions were reduced to approximately 1/4 and 1.4–76 times speedup is achieved compared with the worst mapping.

Keywords: GPGPU, CUDA, parallel programming, compiler, optimization

1 Introduction

General Purpose computation on Graphics Processing Unit (GPGPU) [1, 2] is widely used for the high-performance computing. However, current de facto programming frameworks such as CUDA [3] and OpenCL [4] provide APIs for low-level specifications such as memory allocation and data transfer. Although they enable the user to hand-optimize the program, deep knowledge of GPU architecture is required. Thus, GPGPU programming is complicated and achieving high-performance is very difficult. Moreover, such optimization may not be performance-portable to the different GPU models.

Therefore, we are developing a new programming framework named *MESI-CUDA* [5, 6, 7]. MESI-CUDA is a CUDA variation which aims to make GPGPU programming easier and more portable, but still achieving high-performance. Adopting simpler programming model, MESI-CUDA hides low-level GPU features. The low-level code is not needed in the user's program because such code is automatically generated and also optimized by the compiler.

Current MESI-CUDA hides complicated GPU memory architecture under a virtual-shared memory model, eliminating low-level code for memory management and data transfer. The compiler makes memory access optimizations such as using shared memories as explicit caches [7]. On the other hand, creating and mapping GPU threads are same as CUDA; the user defines kernel functions for GPU and invokes them as threads, explicitly mapping to physical resources. We regard explicit multi-threading is necessary for practical high-performance computing on GPU, but physical mapping should be hidden because its hand-optimization is very difficult and not portable between GPU models. Furthermore, thread mapping also affects the memory usage thus inappropriate mapping may obstruct our automatic memory access optimizations.

In this paper, we propose a new creation scheme of GPU threads. We introduce new syntax of thread creation specifying hardware-independent logical mapping. Such logical mapping is converted to CUDA's physical mapping at compile time. In our scheme, the user can create threads without considering low-level and device-dependent parameters such as the number of GPU cores. Because physical mapping is determined by the compiler, many automatic optimizations are possible. Based on static analysis of array index expressions, the compiler determines thread mapping improving the usage of physical resources and the effect of memory access optimization.

This paper is organized as follows: Section 2 gives an introduction of GPU, CUDA, MESI-CUDA and CUDA optimization techniques. Section 3, 4 details our logical mapping and how to convert it to optimized physical mapping. Section 5 shows the evaluation results and in Section 6 we discuss the related works. In Section 7 and 8, we state our future works and the conclusion.

2 Background

2.1 GPU Architecture

GPU is a collection of streaming multiprocessors (SM), which have certain number of CUDA cores. Although CUDA cores are simpler than typical CPU cores, a GPU has hundreds or thousands of CUDA cores. Thus the potential performance of a GPU is much higher than a CPU.

Fig. 1 shows a typical architecture of a GPU card installed on a PC. Similarly as the CPU cores share the main memory (called *host memory* in CUDA programming), all CUDA cores share a large off-chip *device memory*. Furthermore, each SM has a small on-chip memory called *shared memory*, which is shared by all CUDA cores in the SM.

NVIDIA GPU architecture has been evolved in each generations Tesla, Fermi [8], Kepler [9], and the latest Maxwell [10], introducing new features and changing many hardware specifications such as the numbers of SMs and CUDA cores. Different GPU models often have different specifications even if they belong to the same generation.

2.2 CUDA

CUDA (Compute Unified Device Architecture) [3, 11, 12] is a GPGPU programming framework provided by NVIDIA, using extended C/C++ or Fortran. Fig. 2 shows a matrix multiplication program using CUDA. The additional code required for parallel programming in CUDA is shown underlined.

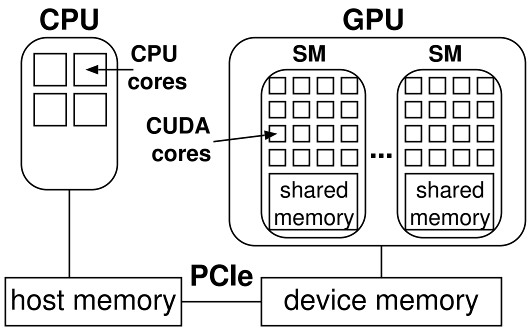

Figure 1: GPU Architecture

```
 1 #define N 1024
 2 #define BX 128
 3 #define S (N*N*sizeof(int))
 4 int ha[N][N], hb[N][N], hc[N][N];
 5 __global__ void matmul(int a[][N], int b[][N], int c[][N]){
 6   int k;
 7   int row = blockDim.y*blockIdx.y+threadIdx.y;
 8   int col = blockDim.x*blockIdx.x+threadIdx.x;
 9   c[row][col] = 0;
10   for(k = 0 ; k < N ; k++){
11     c[row][col] += a[row][k] * b[k][col];
12   }
13 }
14 void init_array(int d[N][N]){...}
15 void output_array(int d[N][N]){...}
16 int main(int argc, char *argv[]){
17   int *da, *db, *dc;
18   dim3 dimGrid(N/BX, N);
19   cudaMalloc(&da, S);
20   cudaMalloc(&db, S);
21   cudaMalloc(&dc, S);
22   init_array(ha);
23   init_array(hb);
24   cudaMemcpy(da, (int*)ha, S, cudaMemcpyHostToDevice);
25   cudaMemcpy(db, (int*)hb, S, cudaMemcpyHostToDevice);
26   matmul<<<dimGrid, BX>>>((int(*)[N]))da, (int(*)[N]))db, (int(*)[N]))dc);
27   cudaMemcpy((int*)hc, dc, S, cudaMemcpyDeviceToHost);
28   output_array(hc);
29   cudaFree(da);
30   cudaFree(db);
31   cudaFree(dc);
32 }
```

Figure 2: CUDA Program of Matrix Multiplication

In CUDA, CPU and GPU are called *host* and *device*, respectively. Functions, declared with the `__device__` or `__global__` qualifier, are called *kernel functions* and can be executed on the device (Fig. 2 *l.* 5–13). The other functions (called *host functions* in this paper) are executed on the host

Table 1: CUDA Built-in variables

`gridDim.x`, `gridDim.y`, `gridDim.z`	grid size (number of blocks)
`blockIdx.x`, `blockIdx.y`, `blockIdx.z`	block index (in the grid)
`blockDim.x`, `blockDim.y`, `blockDim.z`	block size (number of threads)
`threadIdx.x`, `threadIdx.y`, `threadIdx.z`	thread index (in the block)

(*l.* 14–32). To start computation on the GPU, any host function can invoke a `__global__` kernel function specifying the number of threads (*l.* 26). Then, the created GPU threads execute the kernel function. In this paper, we simply call such GPU threads as *threads*.

CUDA uses *grids* and *blocks* for controlling thread mapping to data and physical resources. A block is a group of threads executed on the same SM, and a grid is a group of blocks of the same size. On the invocation of a kernel function, execution configuration must be specified using an expression of the form `<<<`D_g, D_b`>>>`. D_g and D_b are the sizes of the grid and blocks, respectively. They should be values of integer or a built-in 3D vector type `dim3`. For example, Fig. 2 program creates a grid of `N/BX` $\times$ `N` blocks and each block consists of `BX` threads (Fig. 2 *l.* 18, 26). `BX` is used because the block size is limited (1024 at maximum for Kepler), and also may be tuned for the performance.

Built-in variables shown in Table 1 are available to obtain grid/block sizes and block/thread indices. Using them in the index expressions of arrays, each thread performs the same computation on the different array element. In the kernel function `matmul()` of Fig. 2 program, `row` and `col` are computed using block/thread indices so each thread computes different element of the array `c` (*l.* 7–12).

The host/device memories are only accessible from CPU/GPU cores, respectively. For the data sharing between CPU and GPU, memory allocations on both memories and data transfers are required. In CUDA programming, the user must explicitly describe such low-level behaviors calling API functions: memory allocation/deallocation calling `cudaMalloc()`/`cudaFree()` (*l.* 19–21, 29–31), and data transfer calling `cudaMemcpy()` (*l.* 24–25, 27).

2.3 Hand-Optimization of CUDA program

Various hand-optimizations are possible in CUDA programming [12, 13, 14], considering architecture-level features of the target GPU model. Because thousands of CUDA cores perform computation using data on a single device memory, hiding the memory access latency is the most important strategy.

2.3.1 Memory Access Optimization

A thread block is partitioned into groups of 32 threads called *warps* and a SM executes all threads of a warp in a SIMD style. When the current warp is stalled on memory accesses, the execution is switched to other active warps. The device memory accesses in a warp are *coalesced* if the requested data are in the same L2 cache line of 128 bytes. Therefore, the access transactions are largely reduced if the threads in a warp simultaneously access data of the same or neighboring memory addresses [12].

Another optimization is to allocate frequently-used data in the shared memories on each SM. Because their access latency is much smaller, they can be used as the explicit caches of the device memory. A local variable of a kernel function can be allocated on the shared memory using `__shared__` qualifier. However, explicit copy from/to the device memory is needed in the function because its lifetime is within the function. Such variables are shared among all threads in the block, thus explicit synchronizations calling `__syncthreads()` may be needed. Furthermore, the variables caching on the shared memories must be carefully determined because each available capacity is only 48KB [1]. Explicit caching of an array can also be used to convert non-coalesced accesses to coalesced

[1]Physical size of the shared memory is 64KB (96KB for the second generation Maxwell), but only 48KB is available for each thread block.

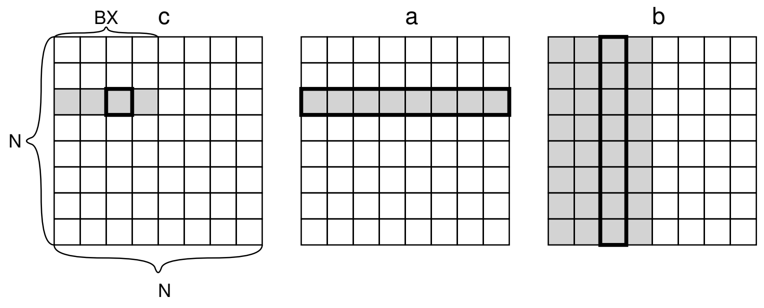

Figure 3: Thread Mapping for Matrix Multiplication

accesses; the threads in a block can cache the required segment of the array using coalesced accesses and later access to the shared memory.

2.3.2 Thread Mapping Optimization

The grid/block sizes are not limited by the numbers of SMs and CUDA cores; scheduling threads to the physical resources are managed by the hardware. However, physical mapping is not completely hidden. The execution configuration and the usage of build-in thread/block indices largely affect the performance.

The block size, determined by the execution configuration, should be an integral multiple of the warp size 32. Although the number of warps per block is 32 (=1024/32) at maximum, it is better to have many warps in a block so the SM can switch the current warp and hide the memory access latency. The number of blocks also should be large enough to keep all SMs busy. If multiple blocks are assigned to the same SM, some of the blocks may be executed concurrently to increase active warps on the SM. The block size also affects the possibility of explicit caching because the cached data size often depends on the number of threads in a block.

Mapping threads to data is determined by the user's usage of build-in thread/block indices. On the other hand, the threads in the same block are mapped to the same SM and the threads of consecutive `threadIdx.x` are mapped to the same warp. Thus, the indices usage determines if the coalesced accesses are possible. It also determines the access locality within a block and may influence explicit caching.

To make the optimization difficult, the user must consider the trade-off between utilizing the shared memories and assigning multiple warps/blocks to a SM. The larger block size increases active warps but may obstruct explicit caching because the required data size will also increase. Using shared memories suppresses concurrent execution of blocks because the total resources required by all concurrent blocks on a SM cannot exceed the physical resources. For example, only one (two for the latest Maxwell) block per SM is executed simultaneously if the usage of shared memory in the block is more than 32KB.

Fig. 3 shows the thread-data mapping of Fig. 2 program. Shaded and thick framed regions are the accessed elements in a block and in a thread of the block, respectively. In each thread, one element of the array `c` is updated `N` times but it can be optimized to use a temporal scalar variable and write back once. For the array `a`, all threads in a block access the same element for each loop iteration, and the accessed elements are consecutive in the memory. Therefore, the access will be coalesced and explicit caching is also available. On the other hand, each thread in a block accesses exclusive range of the array `b`. Caching is difficult because `N` $\times$ `BX` elements are needed in a block. Although the accesses in a block will be coalesced, the accesses are redundant because `N` blocks have a thread accessing the same column of `b` and their accesses cannot be coalesced.

A hand-optimization technique using a square partitioning [11] is shown in Fig. 4. All threads in the block accesses `B` rows/columns of `a`, `b`, respectively. On the loop iteration i, all threads access the element of column/row i of `a`, `b`, respectively. Thus the code is optimized to repeat caching `B`$\times$`B` elements of `a`, `b` (dark regions in Fig. 4) and executing the iteration `B` times. The optimized code

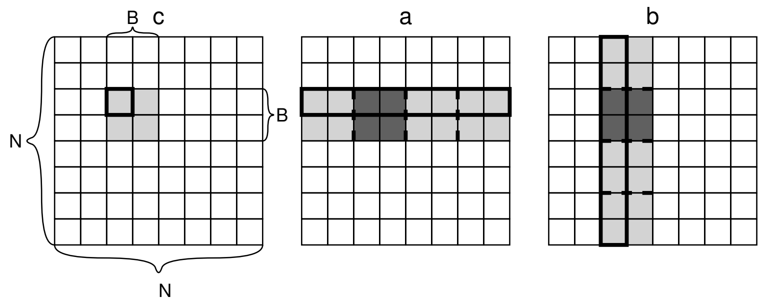

Figure 4: Optimized Thread Mapping for Matrix Multiplication

```
#define N 1024
#define B 16
       ⋮
__global__ void matmul(int a[][N], int b[][N], int c[][N]){
  int k,_ix1;
  int row = blockDim.y*blockIdx.y+threadIdx.y;
  int col = blockDim.x*blockIdx.x+threadIdx.x;
  int cval = 0;
  for (_ix1 = 0 ; _ix1 < N ; _ix1 += B){
    __shared__ int as[B][B];
    __shared__ int bs[B][B];
    as[threadIdx.y][threadIdx.x] = a[row][threadIdx.x+_ix1];
    bs[threadIdx.y][threadIdx.x] = b[threadIdx.y+_ix1][col];
    __syncthreads();
    for (k = 0 ; k < B ; k++){
      cval += as[threadIdx.y][k] * bs[k][threadIdx.x];
    }
    __syncthreads();
  }
  c[row][col] = cval;
}
       ⋮
int main(int argc, char *argv[]){
  int *da, *db, *dc;
  dim3 dimGrid(N/B, N/B);
  dim3 dimBlock(B, B);
       ⋮
  matmul<<<dimGrid, DimBlock>>>((int(*)[N]))da, (int(*)[N]))db, (int(*)[N]))dc);
       ⋮
```

Figure 5: Optimized CUDA Program of Matrix Multiplication

is shown in Fig. 5. The additional/modified code for the parallelization and optimization is shown underlined.

2.4 MESI-CUDA

CUDA programming API directly reflects the complex GPU architecture. Although such low-level API enables hand-tuning considering hardware specifications, it is difficult and may not be efficient

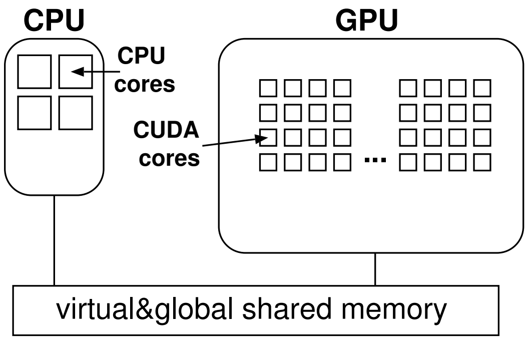

Figure 6: MESI-CUDA Programming Model

on other GPU models. Therefore we are developing an easier GPGPU programming framework *MESI-CUDA* [5, 6, 7].

MESI-CUDA adopts a virtual shared memory model that all CPU/CUDA cores share a single global memory (Fig. 6). Actually, only global variables defined with `__global__` qualifier are shared. To avoid confusing with the CUDA variables defined with `__shared__` qualifier, we call our shared variables as *virtual shared variables* or *VS variables.* The values of VS variables are made logically consistent on each kernel invocation, thus explicit synchronization/mutual exclusion are not needed.

On the other hand, basic parallelization scheme is same as CUDA: writing host/kernel functions for CPU/CUDA cores and invoking the latter from the former. We do not hide this explicit parallelization because the characteristics of CUDA cores are quite different from the CPU cores. For example, CUDA cores can run fine-grained threads with small overhead, but branch divergent code is inefficient. In high-performance computing using GPU, it would be impractical to write code ignoring such differences.

Using MESI-CUDA, the matrix multiplication program in Fig. 2 can be simplified as shown in Fig. 7. The additional code required for parallel programming in MESI-CUDA is shown underlined. The arrays for 2D matrices can be defined as VS variables and can be accessed from both host/kernel functions (Fig. 7 *l.* 3) [2]. The user can concentrate on parallel algorithm without low-level API functions for memory management and data transfer. However, MESI-CUDA is upper-compatible to CUDA and hand-optimizing using CUDA API is possible if the compiler's optimization is not sufficient.

The MESI-CUDA compiler is implemented as a translator to CUDA code. The low-level code such as memory management and data transfer is generated by the compiler. To achieve high-performance without user's hand-optimization, the compiler performs optimization based on static analysis. For this purpose, we have developed automatic optimization schemes such as overlapping thread execution/data transfer [5] and explicit cache using shared memories [7]. However, thread mapping cannot be optimized because current MESI-CUDA adopts physical mapping like CUDA. Moreover, user's mapping may obstruct existing optimizations by the compiler.

For the Fig. 7 program, our compiler caches the array `a`. However, applying the optimization shown in Fig. 5 is impossible because thread-data mapping is not modified.

3 Logical Grid of Threads

We propose a new logical thread mapping scheme for MESI-CUDA. It hides low-level GPU features from the user, device-dependent optimization can be left to the compiler, and more flexible automatic optimization is possible.

To keep the lower compatibility with CUDA, we introduce new execution configuration specified using an expression of the form `<[<`D_0, …, D_{m-1}`>]>`. The expression specifies a single thread grid

[2]If the input/output variables of multiplication is fixed to `ga`, `gb`, and `gc`, they can be directly accessed in kernel functions and do not need to be passed as function arguments.

```
#define N 1024
#define BX 128
__global__ int ga[N][N], gb[N][N], gc[N][N];
__global__ void matmul(int a[][N], int b[][N], int c[][N]){
  int k;
  int row = blockDim.y*blockIdx.y+threadIdx.y;
  int col = blockDim.x*blockIdx.x+threadIdx.x;
  c[row][col] = 0;
  for(k = 0 ; k < N ; k++){
    c[row][col] += a[row][k] * b[k][col];
  }
}
void init_array(int d[N][N]){...}
void output_array(int d[N][N]){...}
int main(int argc, char *argv[]){
  init_array(ga);
  init_array(gb);
  matmul<<<dimGrid, BX>>>((int(*)[N]))ga, (int(*)[N]))gb, (int(*)[N]))gc);
  output_array(gc);
}
```

Figure 7: MESI-CUDA Matrix Multiplication

Table 2: Built-in Variables for Logical Mapping

`lGrid.dim[`i`]` $(i = 0, \ldots, m-1)$	grid size (number of threads)
`lThread.Idx[`i`]` $(i = 0, \ldots, m-1)$	thread index (in the grid)
`lGridDim.x`, `lGridDim.y`, `lGridDim.z` `lThreadIdx.x`, `lThreadIdx.y`, `lThreadIdx.z`	alias to `lGrid.dim[`i`]` and `lThread.Idx[`i`]` $(i = 0, 1, 2)$

```
        ⋮
  int row = lThreadIdx.y;
  int col = lThreadIdx.x;
  c[row][col] = 0;
  for(k = 0 ; k < N ; k++){
    c[row][col] += a[row][k] * b[k][col];
        ⋮
int main(int argc, char *argv[]){
        ⋮
  matmul<[<N, N>]> ((int(*)[N]))ga, (int(*)[N]))gb, (int(*)[N]))gc);
        ⋮
```

Figure 8: MESI-CUDA Matrix Multiplication using Logical Mapping

of arbitrary dimensions, which size is $D_0 \times \ldots \times D_{m-1}$. In this paper, we call the configuration of new form and the specified grid as logical execution configuration and logical grid, respectively. We also introduce new built-in variables shown in Table 2.

The logical grid has no limitation on its dimension and size. They can be determined to naturally map the application's data structure. Thus the code will be independent from the physical resources. For example, the execution configuration and data mapping (i.e. accessed row/column of arrays) of Fig. 7 program are briefly specified as shown in Fig. 8.

4 Conversion to Optimized Physical Mapping

4.1 Basic Strategy

We make static analysis of array index expressions to obtain groups of threads accessing the same or neighboring elements. Here we define the following terminology:

- target variable

  Suppose that a m-dimensional logical grid of size $D = D_0 \times \ldots \times D_{m-1}$ is specified on invoking a kernel function, and an access to a n-dimensional array occurs within $k-$nested loops in the function. On analyzing each index expressions $e_0, \ldots, e_{n-1}$, we regard all built-in logical index variables `lThread.Idx[0]`, ..., `lThread.Idx[m-1]`, and all loop variables i_0, ..., i_{k-1}, as the target variables.

- normal form of index expression

  the normal form of an index expression e, denoted as $N(e)$, is in the form:

$$\begin{aligned} N(e) &= C^I_0 \times \texttt{lThread.Idx[0]} + \ldots + C^I_{m-1} \times \texttt{lThread.Idx[}m-1\texttt{]} \\ &+ C^L_0 \times i_0 + \ldots + C^L_{k-1} \times i_{k-1} \\ &+ C^c \end{aligned}$$

  The normal form must be a polynomial expression of first-degree terms of target variables. C^I_p, C^L_q are the coefficients of each term and must be loop-invariant. C^c is the constant term.

We also denote the sizes of warp and L2 cache line as S_w, S_l, respectively. As mentioned in Section 2.3.1, $S_w = 32$ and $S_l = 128$ for current GPU architectures.

We expect kernel functions satisfy the following assumptions:

1. All array index expressions can be transformed to the normal form.

2. All numbers of loop iterations are fixed and known at compile time.

Currently, we cannot handle non-linear index expressions. However, we expect most kernel functions can be analyzed because irregular access patterns are inefficient and tend to be avoided in GPU code. Omitting the expressions which does not satisfy the assumptions may cause non-optimal mapping, but code generation is still possible.

Unfixed numbers of loop iterations also make our analysis difficult. For the simplicity, here we assume all loop statements are in the following form:

$$\texttt{for } (i \texttt{ = } e_s \texttt{ ; } i \textit{ op } e_e \texttt{ ; } i \texttt{ += } c)\{\ldots\}$$

where e_s, e_e, and c are constant expressions and *op* is one of the following compare operators: `<`, `<=`, `>`, and `>=`. However, this assumption is practically too strict thus in Section 4.4.1 we discuss relaxing the limitation.

We statically analyze index expressions of each array variable occurrence, and obtain the relationship between threads and array indices. We also make range analysis of the index expressions [7] to obtain the accessed range in a thread for each array. Using the result, we make groups of threads maximizing the chance to apply memory access optimizations. The basic strategies are as follows:

1. Threads t_p, t_q access the same array element if each corresponding index expression $e_0, \ldots, e_{n-1}$ has the same value. Assigning t_p and t_q in the same warp makes their access for the occurrence coalesced.

2. If the values are the same for $e_0, \ldots, e_{n-2}$ and close to for e_{n-1}, assigning t_p and t_q in the same warp can expect (partially) coalesced access because they access neighboring addresses.

Testing the strategies for every thread pair consumes large computation time. Additionally, generating efficient code is difficult if logical/physical threads have nonlinear relationship. Therefore, we partition the m-dimensional logical grid to s-dimensional tiles and assign each tile to a CUDA thread block. To select preferred s dimensions from m logical dimensions, dimensions are given partitioning-priority value, which are denoted as $P_0, \ldots, P_{m-1}$. We also introduce notations P'_r, $P''_r (r = 0, \ldots, m-1)$, which represents the priority values derived from the basic strategy 1 and 2, respectively. We only examine threads which are adjacent on one of the m-dimensions and add priority values when they satisfy the strategy conditions.

Estimating advantage/disadvantage of explicit caching is very difficult because it reduces the memory access cost but suppresses concurrent blocks. Therefore, currently we only use coalescing detection for selecting s dimensions and explicit caching is considered on determining the block size.

4.2 Static Analysis

4.2.1 Building Normal Forms of Index Expressions

For each index expression in the kernel function, we transform it to the normal form. For non-target variables in the expression, we find the assignment whose value is valid at the occurrence of index expression. Then we replace the variable with the right hand side expression of the assignment. For example, the normal forms of index expressions for the occurrence of `a` in Fig. 8 *l*. 10 are `lThreadIdx.y` and `k`.

4.2.2 Detecting Access to Same Elements

To detect multiple accesses to the same element, index expressions are examined for each array variable occurrence. If all e_0, …, e_{n-1} do not have the term of `lThread.Idx[`r`]`, all thread with the same logical indices except dimension r access the same element and can be coalesced. For each such occurrence, we add $L \times S_w$ to P'_r, where L is the product of loop iteration numbers $L_0, \ldots, L_{k-1}$.

For example, all threads with the same `lThreadIdx.y` value access the same element on the occurrence of `a` in Fig. 8 *l*. 10 because `lThreadIdx.x` does not appear in the index expressions. Thus $\texttt{N} \times S_w$ is added to P'_0. Similarly, the same value is added to P'_1 examining the `b` occurrence.

4.2.3 Detecting Access to Neighboring Elements

Even if all index terms exist, some priority value should be given if the threads adjacent on logical grid accesses the neighboring element on the memory. For this purpose, the rightmost index expression e_{n-1} is examined. If only one index term `lThread.Idx[`r`]` appears and the coefficient C^I_r is a constant value, $L \times \lfloor S_l / \{C^I_r \times \texttt{sizeof(}\textit{type of } v_p \textit{ element}\texttt{)}\} \rfloor$ is added to P''_r. This expression evaluates the efficiency of coalescing, i.e. how many cache lines are needed for the accesses in a warp.

For example, threads of the consecutive `lThreadIdx.x` value access the consecutive elements on the occurrence of `b` in Fig. 8 *l*. 10, because the rightmost index expression is `col` (=`lThreadIdx.x`). Thus $\texttt{N} \times \lfloor S_l / \{1 \times \texttt{sizeof(int)}\} \rfloor$ is added to P''_0.

4.3 Thread Mapping

The execution configuration is specified for each thread invocation. Thus the possibilities of coalescing accesses and explicit caching are not influenced by the configurations of other invocations. Therefore, each thread mapping can be determined independently. However, using the shared memories for explicit caching may reduce the concurrent blocks of other invocations.

4.3.1 Mapping Logical Dimensions within CUDA Blocks

First, the partitioning-priority values $P_r (r = 0, \ldots, m-1)$ are determined using P'_r, P''_r obtained in the static analysis. Assuming the array element is word-size, each access to the same/consecutive elements adds $S_w = 32$ and $\lfloor S_l / \{1 \times 4\} \rfloor = 128/4 = 32$ to P'_r and P''_r, respectively. Thus we currently adopt a simple evaluation: $P_r = P'_r + P''_r$.

However, giving larger weight to P'_r may achieve better mapping result. Although P'_r and P''_r are *comparable* on evaluating the coalescing efficiency, they have different impact on the efficiency of explicit caching; accessing to the same element requires much smaller capacity. Determining the appropriate weighting factor is our future work.

Let $r_0, \ldots, r_{m-1}$ are dimensions of logical grid satisfying $P_{r_0} \geq \ldots \geq P_{r_{m-1}}$. The threads which have the same logical indices except `lThread.Idx[`r_0`]` have the highest chance to apply coalescing. Therefore, the threads aligned along the r_0-direction should be grouped into a CUDA block. We call such grouping r_0-fusion.

4.3.2 Determining Size of Thread Blocks

We select the block size B from 64, 128, 256, 512 and 1024. Larger size can provide more active warps. However, explicit caching may require larger size, which can exceed the size of shared memories or suppress concurrent execution of blocks. Because the candidate dimensions to assign within CUDA blocks are determined, possible explicit caching is also identified. So we compute the required shared memory size S_s for each block size and select the maximum size for B satisfying $S_s \leq 16$KB, which enables at least 3 concurrent blocks.

4.3.3 Mapping Logical Dimensions to CUDA Dimensions

Using the determined block size B, we fix the inner-block mapping. If D_{r_0} is greater than B, the threads aligned along the r_0-direction must be partitioned to multiple blocks. CUDA grid/block sizes are configured as $D_{r_0}/B \times D/D_{r_0}$ and $B \times 1 \times 1$, respectively. The logical index `lThread.Idx[`r_0`]` is converted to the expression `blockDim.x*blockIdx.x + threadIdx.x`. Other dimensions are all flattened to a single-dimension and the logical indices are converted to the block indices using `blockIdx.y`. We call this mapping r_0-fusion mapping.

If D_{r_0}, $D_{r_0} \times D_{r_1}$, or so on are smaller than B, the dimensions are mapped to `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. The grid/block sizes and the remaining logical indices are determined same as above.

To make optimization shown in Fig. 5, block mapping using 2-dimensional tiles is needed. The optimization technique utilizes the program's characteristic that the threads adjacent on x/y-dimensions both access to the same values thus explicit caching is efficient. Therefore, the threads are partitioned using two dimensions if $P'_{r_0} > 0$ and $P'_{r_1} > 0$. The block size B is selected from 64, 256, and 1024 in this case. CUDA grid/block sizes are configured as $D_{r_0}/b \times D_{r_1}/b \times D/(D_{r_0} \times D_{r_1})$ and $b \times b \times 1$, respectively, where $b^2 = B$. We call this mapping r_0r_1-fusion mapping.

4.4 Improvements of Mapping Conversion

For some code patterns of the target program, applying the following extensions helps to obtain better mapping result.

4.4.1 Computing Variable Number of Loop Iterations

The partition-priority value P_r shows the amount of coalesced accesses when r-fusion is selected. As we described in Section 4.2, the priority values depend to the occurrences of array accesses and the total iteration number of the statement including the occurrence. Therefore, under some conditions, non-constant expressions are acceptable in our scheme as the initialize and conditional expressions (e_s and e_e in Section 4.1) of `for` loops.

The iteration number of the loop can be computed at compile time if e_s and e_e are polynomial expressions of first-degree terms of the outer loop variables. Therefore, our scheme can select appropriate mapping in the case that the iteration number of the inner loop depends to the value of the outer loop variable. For example, the total iteration number within 2-nested loops in Fig. 9 (a) and (b) are $N * (N + 1)/2$ and $(N - 2) * 3$, respectively.

However, our scheme currently cannot generate efficient caching code for these cases. Although the range of the inner loop variable for each outer loop iteration can be computed at compile time,

```
for (i = 0 ; i < N ; i++){
  for (j = 0 ; j <= i ; j++){
         ⋮
  }
}
```

(a) j-Loop of Variable-Iterations

```
for (i = 1 ; i < N-1 ; i++){
  for (j = i-1 ; j <= i+1 ; j++){
         ⋮
  }
}
```

(b) j-Loop of Fixed-Iterations

Figure 9: Inner Loop Iterations Depending on Outer Loop Iterations

```
int a[N][N][N], b[N][N][N];
int main(){
  int x, y, z, t;
          ⋮
  for (t = 0 ; t < T ; t++){
    for (z = 0 ; z < N ; z++){
      for (y = 0 ; y < N ; y++){
        for (x = 0 ; x < N ; x++){
          b[z][y][x] += C0*a[z][y][x] + C1*a[z][y][x-1] + ...;
        }
      }
    }
    copyback(b, a);
  }
}
```

Figure 10: Sequential 3D Stencil Program

our scheme currently caches all required elements at the beginning of the kernel function and likely to overflow the available capacity. Replacing cached elements on each loop iteration can reduce the shared memory usage, but will increase the overhead because intra-thread synchronization is needed for each loop iteration.

4.4.2 Distributing Loop Iterations

Our scheme described in Section 4.2 and 4.3 only maps logical thread indices to appropriate physical indices. Therefore, the optimal mapping achieving coalesced accesses is not possible if the rightmost index expression of an array occurrence does not include thread indices.

We illustrate a typical case using a program shown in Fig. 10. The program is a sequential C version of 3D stencil computation which updates a 3-dimensional array for `T` times. Result of a natural parallelization using MESI-CUDA is shown in Fig. 11. Compared with the original version with 4-nested loops, the outmost `t` loop is left in the host function `main` because each iteration have dependency with calling `copyback()`. Although computing each `b` element is independent, the computation is extremely fine-grained while creating $\mathtt{N}^3$ threads may be excessive. In such cases, a standard parallelization is leaving the inner loop(s) in the kernel function as shown in Fig. 11, thus $\mathtt{N}^2$ threads perform $O(\mathtt{N})$ computation in parallel. Unfortunately in this code, consecutive elements of the array `a` are accessed in the same thread and the simultaneous accesses of threads cannot be coalesced.

If iterations of the loop have no dependencies, this code pattern can be optimized by swapping the loop variable and a logical thread index. For an array occurrence, suppose the normal forms of rightmost and next rightmost index expressions e_{n-1} and e_{n-2} are $C_p^I \times i_p + C_p^c$ and $C_q^I \times \mathtt{lThread.Idx}[q] + C_q^c$, respectively. If i_p-loop is in the form: `for (`i_p `= 0 ;` i_p `<` L_p `;` i_p `++)`, where

```
__global__ int a[N][N][N], b[N][N][N];
__global__ void stencil(){
  int x, y=lThread.Idx[0], z=lThread.Idx[1];
  for (x = 0 ; x < N ; x++){
    b[z][y][x] += C0*a[z][y][x] + C1*a[z][y][x-1] + ...;
  }
}
int main(){
  int t;
          ⋮
  for (t = 0 ; t < T ; t++){
    stencil<[<N, N>]>();
    copyback(b, a);
  }
}
```

Figure 11: MESI-CUDA 3D Stencil Program

Table 3: Evaluation programs

`matmul`	matrix multiplication (matrix size 8192^2)
`jacobi`	Jacobi kernel (matrix size 1024^2, 10000 iterations)
`fdtd`	FDTD kernel (matrix size 10240^2, 1000 iterations)
`poisson`	Poisson equation solver kernel using a point-Jacobi method [15] (grid size 128^3)
`hotspot`	2D transient thermal simulation [16] (grid size 512^2)

$L_p =$ `lGrid.dim[`q`]`, the value range of i_p and `lThread.Idx[`q`]` are equivalent. Thus all occurrences of i_p and `lThread.Idx[`q`]` in the kernel function are swapped each other. If such condition is not satisfied, i_p and `lThread.Idx[`q`]` are not *compatible*. In this case, the execution configuration and the conditional expression of the loop also must be changed to swap the value ranges of `lGrid.dim[`q`]` and L_p. Therefore, this optimization can be applied only when such swapping does not change the semantics of the program.

5 Evaluation

We evaluated our scheme using five benchmark programs shown in Table 3, running on the five environments shown in Table 4. The column 'CC' shows the compute capabilities of each GPU. Host PCs in all environments run CentOS 6.4.

For each program, we compared the performance of the execution applying our mapping optimization scheme with the execution using the worst mapping. Time elapsed for executing kernel threads and the speedup ratio are shown in Table 5 and Fig. 12, respectively. Although explicit caching is enabled for both mappings, the caching was not performed on `poisson` and `hotspot` because of insufficient shared memory capacity.

We also evaluated increase of the coalesced accesses on each program. Using a GPU profiler `nvprof` on GeForce GTX980, we obtained values of two metrics: `gld_transactions_per_request` and `gst_transactions_per_request`, which are the average numbers of transactions for each simultaneous global memory load/store of a warp. The result is shown in Table 6. Because the metrics values are obtained for each kernel function, the result of each function is shown for `fdtd`, `poisson`, and `hotspot`. For each simultaneous accesses of 32 threads in a warp, 32 transactions occur if every threads access to the different L2 cache line, while only 1 transaction occurs if all threads access to the same line. Therefore, the maximum value of these metrics are 32, which means no coalesced

Table 4: Evaluation Environments

CPU	Memory	GPU	Generation	CUDA	CC
Core i7 930 2.8GHz	6GB	Tesla C2075	Fermi	5.0	2.0
Core i7 3820 3.6GHz	16GB	GeForce GTX680	Kepler	5.0	3.0
Core i7 4820K 3.7GHz	16GB	Tesla K20c	Kepler	5.0	3.5
Core i7 930 2.8GHz	6GB	GeForce TITAN	Kepler	5.0	3.5
Xeon E5-1620 3.6GHz	16GB	GeForce GTX980	Maxwell	6.5	5.2

Table 5: Time Elapsed for Kernel Execution

	Tesla C2075		GeForce GTX680		Tesla K20c		GeForce TITAN		GeForce GTX980	
	worst(s)	opt(s)	worst(s)	opt(s)	worst(s)	opt(s)	worst(s)	opt(s)	worst(s)	opt(s)
`matmul`	355.94	5.93	343.41	4.52	139.32	5.13	84.54	3.56	80.55	1.63
`jacobi`	17.71	2.46	12.31	1.50	10.73	1.33	7.48	0.94	2.98	0.56
`fdtd`	809.73	52.39	424.85	30.14	347.00	31.27	236.62	20.84	210.87	22.40
`poisson`	1511.70	141.00	918.95	116.19	788.39	157.63	526.93	111.69	212.12	61.88
`hotspot`	7.90	1.19	8.47	1.08	3.01	0.53	2.15	0.75	0.92	0.68

access occurred, and will decrease approaching the value of 1 as the ratio of coalesced accesses increase.

In all five benchmarks, our scheme could select the best mapping strategy. Table 6 shows that most accesses are non-coalesced in the worst mapping and optimized mapping largely increased coalesced accesses, reducing total transactions in each program to approximately 1/4 [3]. As a result, the difference of thread mapping strategy largely affected the performance in all benchmark programs and GPU models (Fig. 12). The optimized mapping achieved 1.4–76 times speedup compared with the worst mapping, several to ten times speedup in most cases. The newer GPU tends to show smaller performance differences, perhaps due to the hardware improvement such as device memory bandwidth and the size of L2 cache (144GB/s : 768KB, 208GB/s : 1536KB, and 224GB/s : 1792KB for C2075, K20c, and GTX980, respectively). Furthermore, the user may specify better mapping strategy than the worst, even without deep knowledge of GPU architecture. However, the evaluation result shows that even on the newest GPU model, several times slowdown often occurs if the mapping strategy does not match with the low-level requirement of the hardware. Using our scheme, the user can specify device-independent mapping without worrying such risk.

6 Related Works

As the result of speeding up various application programs using CUDA [17, 18], many optimization techniques have been proposed [13, 14]. Although such techniques significantly improve the execution performance, it is also pointed out that determining appropriate tuning parameters and balancing trade-offs between conflicting techniques are complex and difficult issues [19]. Therefore, various schemes have been proposed to provide a new GPGPU programming framework better than conventional CUDA and OpenCL. The major approaches can be categorized to (a) automatic code transformation from sequential programs (with some parallel directives) [20, 21, 22], (b) automatic generation of low-level code for conventional GPGPU frameworks [23, 24], and (c) libraries hiding low-level features [25, 26].

6.1 Programming Framework

The approaches of the category (a) transform sequential programs to CUDA code. Although many schemes require parallelization directives [20, 22], the specification is much abstracted and simpler than CUDA API. Thus the low-level architecture is hidden to the user and GPU programming

[3]Note that this table does not show increase/decrease of total load/store.

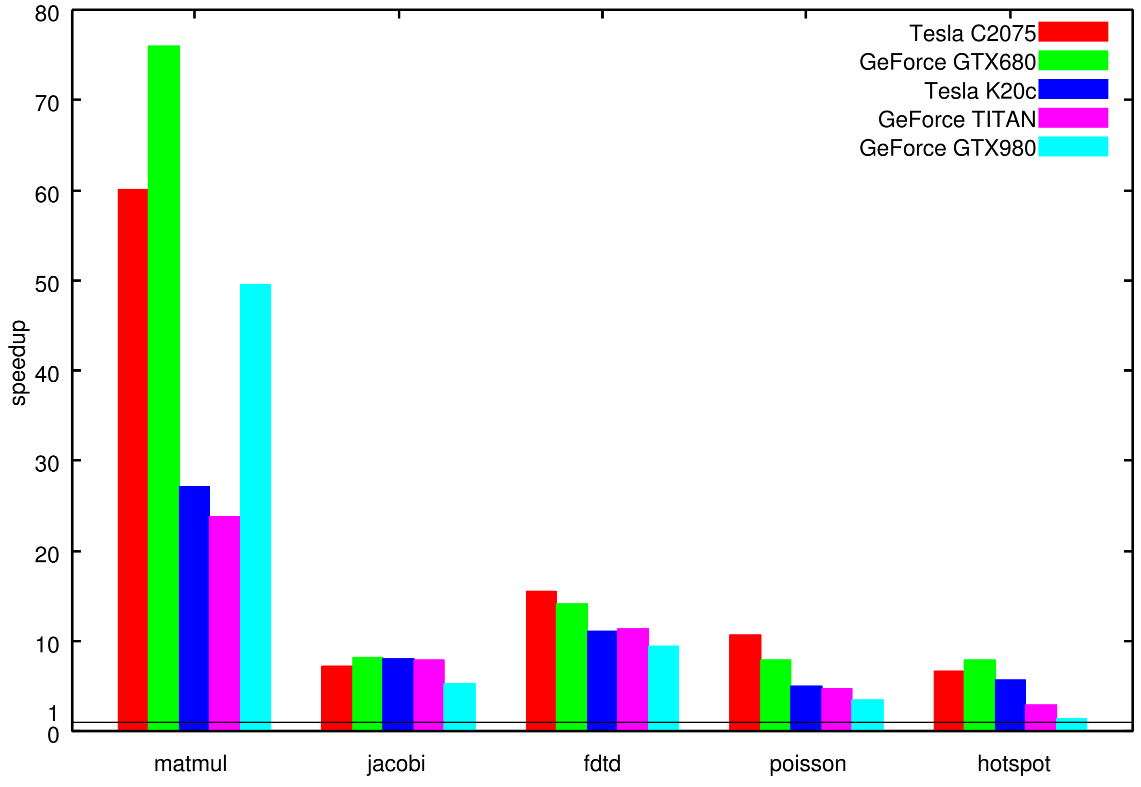

Figure 12: Speedup of Mapping Optimization Compared to the Worst Mapping

Table 6: Global Memory Access Transactions per Load/Store

benchmark	`matmul`	`jacobi`	`fdtd`		`poisson`			`hotspot`	
kernel			`wH`	`wEZ`	`calc`	`sum`	`update`	`calc`	`update`
gld (worst)	32.0	31.4	32.0	32.0	32.0	1.0	32.0	29.4	32.0
gst (worst)	32.0	32.0	32.0	8.0	32.0	1.0	32.0	32.0	32.0
gld (opt)	8.0	6.0	8.7	8.4	9.7	1.0	10.0	7.9	8.0
gst (opt)	4.0	5.0	4.5	4.0	5.0	1.0	5.0	4.0	4.0

is easier and highly-portable. Recently, OpenACC [22] is expected to be the standard of such GPU programming scheme. However, current performance of OpenACC is worse compared with hand-optimized CUDA code [27]. Nevertheless OpenACC provides pragmas for mapping control, such as `gang` and `vector`, hand-optimizing is difficult and needs low-level and hardware-dependent knowledge. Therefore, using such pragmas will spoil the advantage over CUDA programming.

The approaches of the category (b) try to hide low-level GPU features keeping the basic programming model of CUDA. hiCUDA [24] automatically generates CUDA code from sequential programs with directives. Although it is directive-based, the purpose is to simplify the programming based on the CUDA model, not to hide CUDA/GPU architecture. The directives specify low-level behaviors such as allocating variables to the shared memories and synchronization. CUDA-Lite [23] automatically generates memory access code from user-specified annotations, optimizing accesses using shared memories.

MESI-CUDA is also of the category (b). Our purpose is to obtain a practical GPGPU programming framework; easier than CUDA, but still achieving high-performance. Instead of mapping other language into GPU architecture like category (a) approaches, we introduced minimum language extensions into CUDA to hide low-level features, but basic CUDA programming model is retained. We regard explicit specification of host/kernel functions and kernel invocations are easy and portable enough but also sufficient to specify reasonable outline of the program's behavior on GPU environments.

New features hiding conventional low-level features have been added to newer GPUs and CUDA versions. CUDA 6 [11] and Kepler GPUs introduced *managed memory*, which works similar to MESI-CUDA's VS variables. The large difference is that CUDA 6 implements managed memory in hardware/driver-level, while VS variables are implemented in compiler-level as a source code translation. Our advantage is that compile-time optimization is possible using static analysis, such as memory access optimizations and also thread mapping optimization proposed in this paper. The main purpose of managed memory is easier GPGPU programming and using low-level API is encouraged for the high-performance. In contrast, the goal of MESI-CUDA is to hide optimization under the compiler.

The approaches of the category (c) is also practical. Template-base libraries achieve high reusability with small overhead. Instead of automatically generating CUDA code, this approach can provide highly hand-optimized code encapsulating its implementation details. For example, Thrust [26] provides some generic classes so that the user can efficiently handle complicated data structures on GPU.

The library-approach extends the programming framework without modifying the compiler. Although it has an advantage on the implementation cost, global optimization at compile time cannot be introduced. Dynamic optimization, i.e. tuning the library parameters on the fly, is restricted on GPU because using conditional branch in the kernel function may largely decline the performance.

6.2 Automatic Optimization

Many automatic optimization schemes have been proposed for GPGPU programming frameworks [28, 21, 23, 24, 29, 30].

Baskaran et al. [28, 21] use polyhedral model to analyze array accesses in nested loops and obtains affine relations between loop variables and array indices. Using the result, they make optimization on generating CUDA code from sequential C code; mapping loop iterations to CUDA threads is determined to improve memory access efficiency. Our scheme currently adopts simpler and low-cost analysis because our mapping scheme is not so flexible as theirs. The workload of each thread is specified by the user as a kernel function and cannot be determined by the compiler.

Yang et al. [29] propose a compiler which optimizes user's GPU kernel functions. Merging threads/blocks and using shared memories for coalescing accesses, they make a kind of mapping optimization to improve memory access efficiency like our scheme. However, their mapping is less flexible because they only optimize within kernel functions and do not change the thread mapping specified as the execution configuration. Furthermore, the user must specify physical mapping.

Hoshino, et al. [27] report that the layout of data structure largely affects the performance of GPGPU programs. Our scheme selects thread mapping to maximize the chance of optimizing memory accesses in the kernel function. However, the performance may not be sufficient if conflicting mappings are required in a kernel function. Thus introducing data layout optimization may improve our scheme. In Section 7.1 we discuss this issue.

7 Future Works

7.1 Introducing Data Layout Optimization

As we mentioned in Section 6.2, the layout of data structure affects the memory access. Similar as changing thread mapping, transposing multi-dimensional arrays also influences coalesced accesses. Changing array-of-structures (AoS) to structure-of-arrays (SoA) or array-of-structures-of-tiled-arrays (ASTA) increases data locality and enables coalesced accesses [14].

Although array transposing and our thread mapping intuitively seems to have the same effect, they have some practical advantages/disadvantages. In the case that multiple arrays in a kernel function requires different thread-data mapping to maximize the coalesced accesses, thread mapping cannot satisfy both demands, while transposing one of the array may resolve the conflict. AoS-to-SoA optimization is an another example that data layout optimization surpasses thread mapping. However, changing data layout during the execution requires notable cost. In the case that multiple host/kernel functions require different thread-data mapping for the same array, transposing between kernel invocations causes the overhead while specifying different thread mapping for each invocation does not need any cost. Therefore, our future work is to use both optimization scheme and apply appropriate one for each case.

7.2 Thread Mapping to multi-GPUs

If multiple GPU cards are installed on a PC, the GPUs can run kernel threads in parallel. In CUDA programming, API functions target the *current device*, which can be explicitly switched calling `cudaSetDevice()`. Therefore, to utilize multi-GPUs, the user must be aware of the individual GPU devices and (1) explicitly split a kernel invocation into multiple invocations for each device, and also (2) explicitly transfer required data to/from each device.

On splitting a kernel invocation, the user must load-balance on the devices; if different models are installed or the workload of each thread is not uniform, implementing a dynamic load-balancing mechanism will be needed. On such dynamic scheduling, efficiently copying array segments to the device on need will be also difficult and troublesome.

MESI-CUDA programming model, which the host and device virtually share a single memory, can be naturally extended to support multi-GPUs, in which the host and all devices virtually share a single memory. Although this model resolves the issue (2), conventional kernel invocation could not resolve the issue (1). Introduction of logical mapping proposed in this paper hides conventional execution configuration and physical mapping under logical execution configuration. Our future work is to design a scheme that user's kernel invocation using logical mapping is converted to multiple kernel invocations using physical mapping, thereby multi-GPUs can be utilized without any additional specifications.

8 Conclusion

We are developing a GPGPU programming framework MESI-CUDA for the high-performance computing easier than CUDA. Providing virtual shared variables, MESI-CUDA hides low-level memory management and data transfer. However, GPU thread creation and mapping are same as CUDA and device-dependent hand-optimization is needed for the performance. In this paper, we proposed a new thread creation/mapping scheme. We introduced logical mapping specification independent from the low-level architecture. The compiler converts the mapping to CUDA's physical mapping, optimizing memory accesses and controlling the concurrency of blocks and warps.

As the result of evaluation, our scheme selected the optimal mapping strategy for five benchmark programs. Compared with the worst mapping, The global memory transaction was reduced to approximately 1/4 and 1.4–76 times speedup was achieved. However, the current static analysis is still rough estimation. More evaluation and improving the scheme are our future works.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 24500060.

References

[1] J. D. Owens et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[2] Gpgpu.org. `http://www.gpgpu.org/`.

[3] CUDA Zone. `https://developer.nvidia.com/cuda-zone`.

[4] OpenCL. `https://www.khronos.org/opencl/`.

[5] K. Ohno, D. Michiura, M. Matsumoto, T. Sasaki, and T. Kondo. A GPGPU programming framework based on a shared-memory model. *Parallel and Distributed Computing and Networks*, 3:1–14, 2013.

[6] K. Ohno, M. Matsumoto, T. Kamiya, and T. Maruyama. Supporting dynamic data structures in a shared-memory based GPGPU programming framework. In *Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 122–131, 2012.

[7] T. Kamiya, T. Maruyama, K. Ohno, and M. Matsumoto. Compiler-level explicit cache for a GPGPU programming framework. In *Proc. The 2014 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 632–638, 2014.

[8] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[9] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[10] NVIDIA Corporation. *Whitepaper NVIDIA GeForce GTX980*, 2014.

[11] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, April 2012.

[12] NVIDIA Corporation. *CUDA C Best Practices Guide*, January 2012.

[13] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, 2008.

[14] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu. Optimization and architecture effects on GPU computing workload performance. In *Proc. Innovative Parallel Computing 2012*, InPar 2012, pages 1–10, 2012.

[15] Himeno benchmark. `http://accc.riken.jp/2444.htm`.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadoron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE Intl. Symp. on Workload Characterization*, IISWC 2009, pages 44–54, 2009.

[17] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proc. 2009 IEEE Intl. Symp. on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, 2009.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, October 2008.

[19] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc.6th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, CGO '08, pages 195–204, 2008.

[20] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44:101–110, 2009.

[21] M. Baskaran, Jj. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010.

[22] OpenACC Home. `http://www.openacc-standard.org/`.

[23] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[24] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78–90, 2011.

[25] N. Sundaram, A. Raghunathan, and S. T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. *Intl. Parallel and Distributed Processing Symp.*, 0:1–12, 2009.

[26] NVIDIA Corporation. *Thrust Quick Start Guide*, March 2012.

[27] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *CCGRID*, pages 136–143. IEEE Computer Society, 2013.

[28] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. 22nd Annual Intl. Conf. on Supercomputing*, ICS '08, pages 225–234, 2008.

[29] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45:86–97, 2010.

[30] M. Moazeni, A. Bui, and M. Sarrafzadeh. A memory optimization technique for software-managed scratchpad memory in GPUs. In *Proc. Symp. on Application Specific Processors*, SASP 2009, pages 43–49, 2009.