Compiler-Level Explicit Cache for a GPGPU Programming Framework

Tomoharu Kamiya[1], **Takanori Maruyama**[1], **Kazuhiko Ohno**[1], **and Masaki Matsumoto**[2]
[1]Department of Information Engineering, Mie University, Tsu, Mie, Japan
[2]Medical Engineering Institute, Inc., Tsu, Mie, Japan

Abstract— *GPU is widely used for high-performance computing. However, standard programming framework such as CUDA and OpenCL requires low-level specifications, thus programming is difficult and the performance is not portable. Therefore, we are developing a new framework named MESI-CUDA. Providing virtual shared variables accessible from both CPU and GPU, MESI-CUDA hides complex memory architecture and eliminates low-level API function calls. However, the performance of current implementation is not sufficient because of the large memory access latency. Therefore, we propose a code-optimization scheme that utilizes fast on-chip shared memories as a compiler-level explicit cache of the off-chip device memory. The compiler estimates access count/range of arrays using static analysis. For mostly reused variables, code is modified to make copy on the shared memory and access the copy, using small shared memories efficiently. As the result of evaluation, our scheme achieved 13%–192% speedup in two of three programs.*

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

1. Introduction

The performance of Graphics Processing Unit (GPU) has been improved rapidly [1]. Therefore, recent GPUs are used as generic high-performance computing resources. Such GPU usage is called General Purpose computation on Graphics Processing Unit (GPGPU) [2]. However, current de facto GPGPU programming frameworks such as CUDA [3] and OpenCL [4] are still difficult to use. They provide APIs for low-level specifications such as memory allocation and data transfer. Although they enable the user to hand-optimize the performance of the program, it requires deep knowledge of GPU architecture. Furthermore, such optimization may not be portable to the different GPU models.

Therefore, we are developing a new framework named *MESI-CUDA* [5], [6] for easier GPGPU programming. MESI-CUDA is a CUDA variation which hides low-level GPU features. It provides virtual shared variables which can be accessed from both CPU and GPU. Explicit memory management or data transfer are not needed. The user can write MESI-CUDA program without low-level specifications expecting automatic optimization by the compiler.

However, current optimization is not sufficient. One reason is that current implementation uses only off-chip device memory. Fast on-chip GPU memories called shared memories are not used. Thus we propose an optimization scheme that automatically utilize the shared memories as compiler-managed caches of the device memory. Based on the result of static analysis, our scheme determines variables to cache so that device memory accesses are minimized. Then copying/writing back code is inserted and the code accessing the variables is modified. Thus the target program is optimized to use shared memories as explicit caches.

This paper is organized as follows: Section 2 gives a brief introduction of GPU/CUDA/MESI-CUDA and points out the current issue. In Section 3 we discuss the related works. Section 4 details the proposed scheme and Section 5 shows the evaluation results. In Section 6, we state 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. We do not discuss other memories, such as constant and texture memories, because the proposed scheme does not use them.

NVIDIA GPU architecture have been evolved in each generations Tesla/Fermi/Kepler changing specifications and introducing new features. Different models often have different specifications even if they belong to the same generation.

2.2 CUDA

CUDA (Compute Unified Device Architecture) [3], [7], [8] is a GPGPU programming framework using extended C/C++ or Fortran. Fig. 2 shows a (non-optimized) matrix multiplication program using CUDA. The additional code required for parallel programming is shown in bold font.

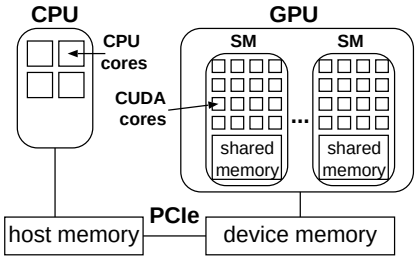

Figure 1: GPU Architecture

```
 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 transpose(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   transpose<<<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 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 executed on the device (Fig. 2 *l*. 5–13). The other functions (called *host functions* in this paper) are executed on the host (*l*. 14–32). To start computation on the GPU, any host function invokes a `__global__` kernel function (called *kernel invocation*) specifying the number of threads (*l*. 26). Then, the created GPU threads execute the kernel function. In this paper, we simply call 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. A kernel invocation creates a grid with the specified grid/block sizes, which are the numbers of total blocks and threads per block, respectively.

Table 1: CUDA Built-in variables

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

The grid/block sizes can be specified as integer values or 3D vectors using a built-in type `dim3`. Fig. 2 program creates a grid of `N/BX`×`N` blocks and each block consists of `BX` threads (Fig. 2 *l*. 18, 26). The grid/block sizes are not limited by the numbers of SMs and CUDA cores; blocks and threads are automatically mapped to the physical resources.

Grid/block sizes and block/thread indices can be obtained using built-in variables shown in Table 1. Using the variables in the index expressions of arrays, each thread can make the same computation on the different array element. In the kernel function `transpose()` 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/CUDA cores, respectively. To share data between CPU and GPU, memory allocations on both memories and data transfers between them 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 CUDA Optimization Techniques

In CUDA programming, hand-optimization considering the architecture-level features of the target GPU often largely contributes achieving high-performance.

2.3.1 Controlling Concurrency/Parallelism

Because each SM executes a *warp* of 32 threads in a SIMD manner, the thread block size should be an integral multiple of the warp size 32. It is better to have multiple warps in a block because the execution can be switched to hide the latency when the active warp is stalled on memory accesses. The number of blocks also should be large enough so that concurrent blocks run on each SM.

2.3.2 Optimizing Memory Usage

The parallel device memory accesses in a warp are coalesced if the requested data are in the same L2 cache line of 128 bytes, but otherwise they are serialized. Thus, the threads in a warp are better to access the neighboring data on the memory at the same time. [8].

Another memory-usage optimization is allocating frequently used data on the shared memories. Because their access latency is much smaller, they can be used as caches of the device memory. Although CUDA enables shared memories to be configured as L1 caches, the performance is

often not sufficient. Their size of 48KB [1] is too small and cached data tend to be not reused when threads are scanning large arrays. Thus explicit caching on the application layer is used as an optimization.

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 direct copying between the host and shared memories is not possible. Such variable is shared among all threads in the block, thus explicit synchronization calling `__syncthreads()` may be needed.

2.3.3 Reducing Data Transfer Overhead

The data transfer between host/device memories can be another bottleneck of performance. Avoiding fine-grained transfers or overlapping transfers/kernel executions using asynchronous transfer functions improves the performance.

2.4 MESI-CUDA

CUDA programming API is based on the complex GPU architecture. Although such low-level API enables hand-tuning considering hardware specifications, it is difficult and may not be efficient on other GPU models. Therefore we are developing an easier GPGPU programming framework *MESI-CUDA* [5], [6], hiding low-level features from the user.

In MESI-CUDA, 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. It would be unpractical to ignore such differences in high-performance computing using GPU.

On the other hand, we adopted a virtual shared memory model that all CPU/CUDA core share a single global memory (Fig. 3). 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*. This design is due to the following reasons:

1) GPU has no hardware/OS support to implement generic virtual shared memory. Virtual-sharing of the specified variables can be implemented at compiler-level, using static analysis and inserting appropriate data transfer code between host/device memories.
2) Because the cores are heterogeneous, the roles of host/device are clear. Many working variables are accessed only on either of the host/device. Thus explicit sharing of minimum variables is safe and also efficient.

Generally, shared memory based parallel programming requires synchronization and mutual exclusion. However, data access races are usually avoided in GPU programming because of the poor synchronization mechanism. Thus we adopt implicit synchronization that the shared values are made logically consistent on each kernel invocation.

[1]User-available size is currently 48KB of physical 64KB memory.

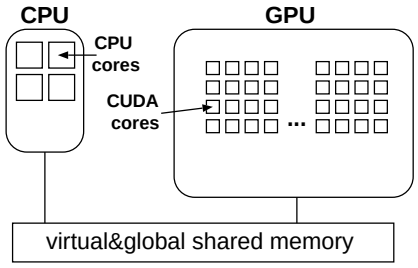

Figure 3: MESI-CUDA Programming Model

```
 1 #define N 1024
 2 #define BX 128
 3 __global__int ga[N][N], gb[N][N], gc[N][N];
 4 __global__
   void transpose(int a[][N], int b[][N], int c[][N]){
 5   int k;
 6   int row = blockDim.y*blockIdx.y+threadIdx.y;
 7   int col = blockDim.x*blockIdx.x+threadIdx.x;
 8   c[row][col] = 0;
 9   for(k = 0 ; k < N ; k++){
10     c[row][col] += a[row][k] * b[k][col];
11   }
12 }
13 void init_array(int d[N][N]){...}
14 void output_array(int d[N][N]){...}
15 int main(){
16   init_array(ga);
17   init_array(gb);
18   transpose<<<dimGrid, BX>>>
         ((int(*)[N]))ga,(int(*)[N]))gb,(int(*)[N]))gc);
19   output_array(gc);
20 }
```

Figure 4: MESI-CUDA Matrix Multiplication

Without low-level description of memory management and data transfer [2], the user can concentrate on device-independent parallel algorithm. For example, the matrix multiplication program in Fig. 2 can be simplified using MESI-CUDA as shown in Fig. 4. The additional code required for parallel programming in MESI-CUDA is shown in bold font. The arrays for 2D matrices are defined as VS variables and can be accessed from both host/kernel functions (Fig. 4 *l*. 3) [3]. We support variable-length array and dynamic allocation of VS variables [6], but in this paper we only discuss VS variables of static sizes.

The MESI-CUDA compiler is a translator to CUDA and generates low-level code for memory management and data transfer. Our research goal is to automatically apply the

[2]To make MESI-CUDA upper-compatible to CUDA, we did not remove low-level API functions. If the optimization of MESI-CUDA compiler is not sufficient, the user can hand-optimize like CUDA.

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

optimizations described in Section 2.3 and achieve high performance like hand-optimized CUDA programs.

3. Related Works

The latest CUDA 6 [7] and Kepler GPUs implemented *Unified Memory*, which enables to allocate *managed memory* by either statically defining a variable with `__managed__` qualifier or dynamically calling `cudaMallocManaged()`. Such memory can be accessed from both CPU and GPU.

The features are almost same with MESI-CUDA's VS variables; only user-specified data is logically shared and they are automatically copied between host/device memories. The large difference is that VS variables are implemented in compiler-level, while the managed memory is implemented in hardware/driver-level. Our advantage is that compile-time optimization is possible using static analysis. For example, asynchronous data copying code can be inserted where the data transfer and kernel executions are overlapped. Another example is the synchronizations between host and device. MESI-CUDA automatically inserts synchronization code to maximize their parallel execution, while CUDA 6 requires explicit synchronization calling `cudaDeviceSynchronize()` or setting a environment variable `CUDA_LAUNCH_BLOCKING` as 1 to automatically synchronize for every kernel invocation.

The main purpose of Unified Memory is easier GPGPU programming and hand-optimization using conventional low-level API is encouraged for high-performance. The goal of MESI-CUDA is to hide optimization under the compiler. However, generating code using new CUDA features may help to utilize hardware/driver supports for such features.

OpenACC [9] or OpenMP-to-CUDA translation [10], [11] are another GPGPU approach without low-level specifications. In these programming frameworks, a sequential program with some parallelizing directives is compiled into a parallel program executable on GPU. They have advantages on usability; abstract directives are easier than low-level API functions, sequential programs can be parallelized easily, and the program is portable to different GPU models or other heterogeneous multi-cores. However, their performance depends to the compiler optimization, which is usually worse compared with hand-optimized CUDA code [12]. As mentioned in Section 2.4, we consider explicit and heterogeneous parallel programming is necessary for high-performance.

For various input languages, schemes for automatic generation and optimization of CUDA low-level code are developed. CUDA-Lite [13] automatically generates memory access code from user specified annotations, optimizing accesses using shared memories. Yang, et al. [14] optimize memory accesses in CUDA kernel functions using shared memories for coalescing accesses to the device memory. Although our scheme is similar to these approaches, we do not assume additional annotations by the user and it is supposed to be a part of global optimization including mapping and scheduling in future.

4. Proposed Scheme

Current implementation of MESI-CUDA allocates area for virtual shared (VS) variables on host/device memories. Therefore, every access to the VS variables on GPU is a access to the device memory. By caching VS variables on shared memories, the memory access latency is largely reduced and the performance will be improved. Therefore, we propose a new scheme that the compiler automatically makes the optimization of explicit caching mentioned in Section 2.3.2.

For simplicity, we discuss the case that a kernel function f is invoked by the following statement, where S_g, S_b are integer values:

$$f\texttt{<<<}S_g\texttt{, }S_b\texttt{>>>(...);}$$

The values of `GridDim.x` and `BlockDim.x` will be S_g, S_b, respectively. If f calls other device functions, the analysis and code generation are extent to cover such functions. We also denote the size of available shared memory per block as C. On current GPU models, C=48KB but it may be changed in the future models. Furthermore, using a smaller value as C suppresses the shared memory usage per block, which can increase concurrent blocks per SM.

4.1 Caching Strategy

The scope of variables defined with `__shared__` qualifier is within the defined kernel function f. Thus the caching candidates are the variables which are on the device memory and accessed in f. Assuming that enough registers are available for local variables in f, VS variables and dereferences of pointer arguments will be the device memory accesses. We denote the list of caching candidate as V and include all such variables in V as the initial value.

Because each access latency is reduced for the cached variables, caching is more effective if the number of accessing the variable is larger. However, copying from/to the device memory causes another overhead which increases according to the variable size.

If the variable is an array, not all elements may be accessed in a block. So caching the set of accessed array elements is enough. However, static analysis may not obtain strict set of accessed elements. Furthermore, generating efficient access code is difficult for irregular access patterns. Therefore, we make static analysis to obtain the range of accesses on each dimension of the array. Instead of caching the whole array, the subarray of the obtained range is cached. For multi-dimensional arrays, the obtained range may not be a single continuous area on the device memory. In such cases, the required areas are packed into a continuous area on the shared memory, forming the subarray.

Our scheme gives higher priority of caching if a candidate variable has higher access count per byte. We first make static analysis to obtain the access counts, select variables to be cached, and finally generate code to cache the variables.

4.2 Static Analysis

We make static analysis on each kernel function and obtain access count of each variable in V. We also obtain required bytes for caching each variable. Here we expect that kernel functions satisfy the following assumptions:

1) All loop iteration numbers are fixed and known at compile time.
2) All array index expressions are first degree polynomials on all loop and built-in index variables (`threadIdx.x`, `blockIdx.x`, etc.). For example, `a[i*N+j]` or `a[threadIdx.x/N+i]` are acceptable but `a[i*j]` or `a[threadIdx.x/i]` are not.
3) Kernel functions may have conditional statements, but the branch probability is regard as 1/2; the access counts of `if`/`else` blocks are averaged and the access ranges are merged.

While most practical sequential programs will not satisfy these assumptions, many CUDA programs will satisfy. To prevent inefficient branch divergence, using `if`/`while` statements is tend to be avoided. Index expressions are commonly simple and linear because data elements should be divided equally to the threads preventing access races and balancing the load statically. Even if the assumptions are not satisfied for some candidate variables, we can just remove such variables from the candidate list V and apply our scheme to other variables.

Considering the assumption 1, 3, the access count of a candidate variable v, denoted as $\mathit{access}(v)$, is obtained as the sum of each access count of the variable occurrences. An access count of a variable occurrence is a product of loop iterations which include the occurrence. Considering the assumption 2, the access range of array elements is obtained by computing the index expression value with the minimum/maximum values of loop variables.

Suppose that a candidate variable v is a m-dimensional array and accessed in a kernel function f. The access range of v in the thread $t_{p,q}$ ($q = 0, \ldots, S_b - 1$) belonging to a block b_p ($p = 0, \ldots, S_g - 1$) is obtained as follows.

We denote occurrences of v in f as $v^1, \ldots, v^k$ and s-th index expression of v^r as $e_s(v^r)$. We compute the values of $e_s(v^r)$ on every combination of minimum/maximum values of loop variables. Considering the assumption 2, minimum/maximum of the computed values are the minimum/maximum values of $e_s(v^r)$, denoted as $\mathit{min}(e_s(v^r))$ and $\mathit{max}(e_s(v^r))$ [4]. We denote the range of an index expression value as $R_s(e_s(v^r)) = [\mathit{min}(e_s(v^r)), \mathit{max}(e_s(v^r))]$.

[4]It may not be true if the modulo operator `%` is used because the operation is not monotonic. We simply regard the minimum/maximum value of the term $e\%M$ in the expressions as 0 and $M-1$, respectively.

```
Set target variable set V_c empty
Sort candidate list V in the descending order of access(v)
while (V is not empty){
  Select first variable v_t in V and remove v_t from V
  if (byte(R(v_t, b_p)) ≤ C){
    V_c ← V_c ∪ {v_t}
    C ← C − byte(R(v_t, b_p))
  }
}
```

Figure 5: Algorithm Obtaining Variables to Cache

The access range of v^r is a m-dimensional range denoted as follows [5]:

$$R(v^r) \quad = \quad R_1(e_1(v^r)) \times \ldots R_m(e_m(v^r))$$

The access range of v in the thread $t_{p,q}$ and in the block b_p is obtained as follows:

$$R(v, t_{p,q}) \quad = \quad \bigcup_{r=1}^{k} R(v^r, t_{p,q})$$

$$R(v, b_p) \quad = \quad \bigcup_{q=0}^{S_b-1} R(v, t_{p,q})$$

We define the union of two ranges $R' \cup R''$ as a minimum range including R' and R''.

The required size (number of array elements) and memory bytes for caching v are computed as follows:

$$\begin{aligned}
\mathit{size}(R_s(v, b_p)) &= \mathit{emax}(v, b_p, s) - \mathit{emin}(v, b_p, s) + 1 \\
\mathit{size}(R(v, b_p)) &= \mathit{size}(R_1(v, b_p)) \times \ldots \mathit{size}(R_m(v, b_p)) \\
\mathit{byte}(R(v, b_p)) &= \mathit{size}(R(v, b_p)) \times \texttt{sizeof(}\textit{type of}\ v\texttt{)}
\end{aligned}$$

where

$$\begin{aligned}
R(v, b_p) &= R_1(v, b_p) \times \ldots R_m(v, b_p) \\
R_s(v, b_p) &= \texttt{[}\mathit{emin}(v, b_p, s)\texttt{,}\mathit{emax}(v, b_p, s)\texttt{]}
\end{aligned}$$

and $\mathit{emin}(v, b_p, s)$, $\mathit{emax}(v, b_p, s)$ are respectively minimum/maximum value of $e_s(v^r)$ for all q, r.

4.3 Optimization

Fig. 5 shows the algorithm of obtaining a set of variables to cache: V_c. The average access count per byte of a variable v_t is denoted as $\overline{\mathit{access}}(v_t)$, which is computed as follows:

$$\overline{\mathit{access}}(v_t) \quad = \quad \mathit{access}(v_t)/\mathit{byte}(R(v_t, b_p))$$

4.4 Code Generation

For each caching target $v_t \in V_c$, we apply the following code generation/modification in the kernel function f.

[5]This definition of range assumes that possible values of each index expression is continuous and the expressions are independent each other. The range is redundant in the cases of non-unit stride access patterns or dependent expressions like `a[i][i]`. Introducing more accurate range is the future work.

```
__shared__ type _s_v_t[S_m(v_t)]...[S_1(v_t)];
int _ix1, ..., _ixm;
for (_ixm = 0 ; _ixm < S_m(v_t) ; _ixm++){
    :
    :
for (_ix2 = 0 ; _ix2 < S_2(v_t) ; _ix2++){
for (_ix1 = 0 ; _ix1 < S_1(v_t) ; _ix1 += T ){
  _s_v_t[_ixm]...[_ix1]
     = v_t[_ixm+O_m(v_t)]...[_ix1+O_1(v_t)];
}}...}
__syncthreads();
```

Figure 6: Caching code for a variable v_t

4.4.1 Caching Variables

First, we insert the definition of a variable `_s_`v_t with `__shared__` qualifier. if v_t is an array, the size of the s-th dimension is *size*$(R_s(v, b_p))$. Next, we insert code for copying the initial values from the device memory and writing back the final values to the device memory to the head and tail of f, respectively.

The pseudo code defining `_s_`v_t and copying the initial values is shown in Fig. 6. For simplicity, we assume that the size of the first dimension $S_1(v_t)$ is an integral multiple of the number of copying threads $T =$ `blockDim.x`. We also use the following notations in Fig. 6.

$$\begin{aligned} S_s(v_t) &= \textit{size}(R_s(v_t, b_p)) \\ S(v_t) &= \textit{size}(R(v_t, b_p)) \\ O_s(v_t) &= \textit{emin}(v, b_p, s) \end{aligned}$$

Note that $S_s(v_t)$ and $S(v_t)$ are constant but $O_s(v_t)$ will be not. In most cases, it includes block indices such as `blockIdx.x` thus different on each block.

To copy the initial cache values, each array element of v_t within the caching range is assigned to the corresponding element of `_s_`v_t. Because the elements consecutive on the first dimension are consecutive in the device memory, coalesced accesses are expected on the parallel assignment of such elements. If later accesses in the block are not consecutive, they will cause non-coalesced device accesses without our cache. Using our scheme, the array is cached using coalesced accesses then shared memories are accessed later. Therefore, the access latency will be largely reduced. After copying, `__syncthreads()` must be called to ensure copying is completed before starting computations on them. If v_t is write-only in f, code for copying and synchronization is omitted.

The write-back code will be reverse copy of Fig. 6. The code can be omitted if v_t is read-only in f. Synchronization is not needed after the write-back, because the threads end immediately after that.

4.4.2 Accessing Cache

Each occurrence of v_t in f is replaced with `_s_`v_t. If v_t is an array and only its subset is cached, the index expressions must be modified as follows:

$$v_t[e_m]\ldots[e_1] \quad \rightarrow \quad \texttt{\_s\_}v_t[e_m\text{-}O_m(v_t)]\ldots[e_1\text{-}O_1(v_t)]$$

```
 4 __global__
   void transpose(int a[][N], int b[][N], int c[][N]){
 5   int k, _ix1;
 6   int row = blockDim.y*blockIdx.y+threadIdx.y;
 7   int col = blockDim.x*blockIdx.x+threadIdx.x;
 *   __shared__int _s_a[1][N];
 *   __shared__int _s_c[BX];
 *   for (_ix1 = threadIdx.x ; _ix1 < N ; _ix1 += BX){
 *     _s_a[0][_ix1] = a[row][_ix1];
 *   }
 *   __syncthreads();
 8   _s_c[threadIdx.x] = 0;
 9   for(k = 0 ; k < N ; k++){
10     _s_c[threadIdx.x] += _s_a[0][k] * b[k][col];
11   }
 *   c[row][col] = _s_c[threadIdx.x];
12 }
```

Figure 7: Optimized Kernel Function of Fig. 4 Program

Table 2: Evaluated Programs

name	description
`matmul`	matrix multiplication shown in Fig. 4
`dif`	single dimension diffusion equation solver using difference method
`ep`	EP (Embarrassingly Parallel) in NAS Parallel Benchmarks [15]

Fig. 7 is the result of applying our scheme to kernel function `transpose()` in Fig. 4. Modifications are shown in bold font.

5. Evaluation

To evaluate our scheme, we compared the execution time of MESI-CUDA programs shown in Table 2, applying/not applying the proposed optimization. The result is shown in Table 3. The columns 'normal' and 'opt' are the execution time of programs applying and not applying our optimization, respectively. The column 'speedup' is the inverse of the execution time ratio of 'opt' to 'normal'.

Our optimization achieved speedup on all GPU models for `matmul` and `dif`. As shown in Fig. 7, matrices A and C of $C = A \times B$ are cached in `matmul` and achieved 13% to 192% speedup when S_b is optimized to be the best. In `dif`, the required size for caching is only $S_b \times 4$ bytes and each array elements are shared between adjacent threads. Large block size is possible without reducing concurrent blocks.

As for the result of `ep`, our optimization achieved 23% to 98% speedup on C2050. When $S_b = 32$, it also slightly improved performance on other GPU models. However, the optimization caused slowdown for Kepler GPUs for larger S_b. Applied to `ep`, our optimization caches small arrays for random-accessed histogram but the main array for storing random numbers is too large to be cached. Therefore the contribution of reducing access latency is limited. In addition, large S_b reduces concurrent blocks because the required size of the histogram is $S_b \times 80$ bytes.

Table 3: Execution Time and Speedup using Proposed Scheme

Data Size	Block Size S_b	Tesla C2050 (Fermi) normal(s)	opt(s)	speedup	GeForce GTX 680 (Kepler) normal(s)	opt(s)	speedup	GeForce Titan (Kepler) normal(s)	opt(s)	speedup	Tesla K20 (Kepler) normal(s)	opt(s)	speedup
`matmul`													
1024^2	32	0.162	0.046	3.56	0.091	0.049	1.86	0.065	0.032	2.04	0.082	0.040	2.03
	64	0.089	0.026	3.47	0.051	0.026	1.95	0.036	0.018	2.00	0.046	0.022	2.12
	128	0.055	0.029	1.93	0.034	0.029	1.16	0.024	0.015	1.58	0.033	0.019	1.80
	256	0.045	0.038	1.18	0.034	0.030	1.13	0.024	0.015	1.62	0.033	0.018	1.89
	512	0.045	0.036	1.26	0.034	0.024	1.42	0.024	0.015	1.62	0.034	0.011	3.18
`dif`													
256K	32	6.12	4.13	1.48	4.03	2.47	1.63	2.85	1.66	1.72	4.15	2.39	1.73
	64	3.37	2.20	1.53	2.25	1.35	1.66	1.64	0.97	1.69	2.34	1.35	1.73
	128	2.17	1.35	1.60	1.71	0.91	1.88	1.26	0.75	1.68	1.72	1.00	1.73
	256	1.90	1.26	1.50	1.75	0.96	1.82	1.34	0.78	1.73	1.81	1.04	1.75
	512	2.06	1.43	1.45	1.88	1.09	1.73	1.45	0.87	1.66	1.96	1.14	1.71
512K	32	12.21	8.20	1.49	7.97	4.86	1.64	5.60	3.24	1.73	8.23	4.70	1.75
	64	6.70	4.37	1.53	4.41	2.63	1.68	3.17	1.84	1.72	4.62	2.64	1.75
	128	4.27	2.66	1.61	3.35	1.75	1.92	2.38	1.38	1.72	3.42	1.95	1.75
	256	3.74	2.48	1.50	3.42	1.86	1.84	2.51	1.44	1.74	3.60	2.04	1.77
	512	4.05	2.81	1.44	3.70	2.10	1.76	2.72	1.61	1.69	3.90	2.27	1.71
`ep`													
class B	32	1.60	1.25	1.29	2.04	1.88	1.09	1.36	1.30	1.05	1.17	0.99	1.18
	64	1.05	0.83	1.27	1.84	1.84	1.00	1.29	1.27	1.02	0.74	0.90	0.82
	128	1.65	0.83	1.98	1.81	1.92	0.94	1.30	1.30	1.00	0.94	0.99	0.95
	256	1.30	0.84	1.55	1.80	1.95	0.92	1.29	1.30	1.00	0.92	0.99	0.93
	512	1.03	0.83	1.23	1.79	1.90	0.95	1.33	1.32	1.01	0.93	1.01	0.92

6. Conclusion

Although GPGPU is widely used for high-performance computing, major programming frameworks like CUDA are difficult and the performance is not portable. Therefore, we are developing an easier programming framework MESI-CUDA. However, access latency of virtual shared variables is large, thus we proposed an automatic optimization scheme using on-chip shared memories as explicit cache.

To select variables of higher reused rate as the caching targets, we make static analysis to obtain the average access counts and accessed range in a block for each variable. The target variables are determined at compile time and code for explicit caching is automatically generated. Therefore, no support in hardware/driver-level is required and the dynamic overhead of cache management does not occur.

As the result of evaluations, our scheme achieved 13% to 192% speedup for `matmul`/`dif` programs but slowdown for `ep` program running on Kepler GPUs. Using shared memories reduces concurrent blocks on a SM thus the trade-off should be considered for applying our optimization.

As a future work, the result of current range analysis may be redundant and should be improved. Recognizing non-unit stride access patterns and packing required elements on caching will save the capacity of shared memories. Another issue is that our scheme tries to utilize the shared memories under the restriction of user-specified grids and blocks. The optimization may be far from the best. For example, specifying large block size may increase the access range of arrays and prevent their caching due to the lack of capacity. Another example is that the threads of common accessing range are distributed into different blocks, which prevents to share the cache value. Our next challenge is to develop optimization scheme of threads/data mapping which automatically controls block size and improve efficiency of data accesses and caching.

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*, vol. 26, no. 1, pp. 80–113, 2007.
[2] "Gpgpu.org," http://www.gpgpu.org/.
[3] "CUDA Zone," http://developer.nvidia.com/category/zone/cuda-zone.
[4] "OpenCL," http://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*, vol. 3, pp. 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*, 2012, pp. 122–131.
[7] *NVIDIA CUDA C Programming Guide*, 6th ed., NVIDIA Corporation, February 2014.
[8] *CUDA C Best Practices Guide*, NVIDIA Corporation, January 2012.
[9] "OpenACC Home," http://www.openacc-standard.org/.
[10] S. Lee, S. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, pp. 101–110, 2009.
[11] "OpenMP," http://openmp.org/.
[12] 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*. IEEE Computer Society, 2013, pp. 136–143.
[13] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu, "CUDA-Lite: Reducing GPU programming complexity," in *Languages and Compilers for Parallel Computing*, 2008, pp. 1–15.
[14] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," *SIGPLAN Not.*, vol. 45, pp. 86–97, 2010.
[15] "NAS parallel benchmarks," https://www.nas.nasa.gov/publications/npb.html.