

# hiCUDA: High-Level GPGPU Programming

Tianyi David Han, *Student Member, IEEE*, and Tarek S. Abdelrahman, *Senior Member, IEEE*

**Abstract**—Graphics Processing Units (GPUs) have become a competitive accelerator for applications outside the graphics domain, mainly driven by the improvements in GPU programmability. Although the Compute Unified Device Architecture (CUDA) is a simple C-like interface for programming NVIDIA GPUs, porting applications to CUDA remains a challenge to average programmers. In particular, CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU memory. Practical experience shows that the programmer needs to make significant code changes, often tedious and error-prone, before getting an optimized program. We have designed *hiCUDA*, a high-level directive-based language for CUDA programming. It allows programmers to perform these tedious tasks in a simpler manner and directly to the sequential code, thus speeding up the porting process. In this paper, we describe the *hiCUDA* directives as well as the design and implementation of a prototype compiler that translates a *hiCUDA* program to a CUDA program. Our compiler is able to support real-world applications that span multiple procedures and use dynamically allocated arrays. Experiments using nine CUDA benchmarks show that the simplicity *hiCUDA* provides comes at no expense to performance.

**Index Terms**—CUDA, GPGPU, data-parallel programming, directive-based language, source-to-source compiler.

## 1 INTRODUCTION

GRAPHICS Processing Units (GPUs) have recently gained wide popularity among researchers and developers as accelerators for applications outside the domain of traditional computer graphics. This trend, known as General-Purpose computing on the GPU (or GPGPU), largely results from the great improvements in GPU programmability. The traditional fixed-function graphics pipeline has evolved into a much more flexible and unified many-core architecture [1], and nongraphics data-parallel languages have been introduced to program these cores [2], [3], [4].

The Compute Unified Device Architecture (CUDA) is such a programming language specifically designed for NVIDIA GPUs [2]. As a simple extension to C, CUDA has quickly become popular and attracted more nongraphics programmers to port existing applications to CUDA. However, experience shows that this porting process remains challenging [5]. In particular, CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host memory and various GPU memories, and of manually optimizing the utilization of the GPU memory. Further, the complexity of the underlying GPU architecture demands that the programmer experiments with many configurations of the code to obtain the best performance. The experiments involve different schemes of partitioning computation among GPU threads, of optimizing single-thread code, and of utilizing the GPU memory. As a result, the programmer has to make significant code changes, possibly many times,

before achieving desired performance. Practical experience shows that this process is very tedious and error-prone. Nonetheless, many of the tasks involved are mechanical, and we believe can be automated by a compiler.

Therefore, we have defined a directive-based language called *high-level CUDA (hiCUDA)* for programming NVIDIA GPUs. It provides a programmer with high-level abstractions to carry out the tasks mentioned above in a simple manner, and directly to the original sequential code. The use of *hiCUDA* directives makes it easier to experiment with different ways of identifying and extracting GPU computation, and of managing the GPU memory. We have designed and implemented a prototype compiler that translates a *hiCUDA* program to an equivalent CUDA program. Our experiments with nine CUDA benchmarks show that the simplicity and flexibility *hiCUDA* provides come at no expense to performance. For each benchmark, the execution time of the *hiCUDA*-compiler-generated code is within 2 percent of that of the handwritten CUDA version. Furthermore, our compiler includes various analyses that provide support for real-world applications, which typically span multiple procedures (possibly in multiple files) and use dynamically allocated arrays.

This paper describes the *hiCUDA* directives and the compiler support needed to translate them into CUDA code. It is organized as follows: Section 2 provides background on CUDA programming. Section 3 introduces the *hiCUDA* directives using a simple example. Section 4 specifies these directives in detail. Section 5 describes the design and implementation of the *hiCUDA* compiler. Section 6 gives an experimental evaluation of *hiCUDA* based on our prototype compiler. Section 7 reviews related work. Finally, Section 8 presents concluding remarks and directions for future work.

- The authors are with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada. E-mail: {han,tsa}@eecg.toronto.edu.

Manuscript received 23 Sept. 2009; revised 1 Feb. 2010; accepted 1 Mar. 2010; published online 31 Mar. 2010.

Recommended for acceptance by D.A. Bader, D. Kaeli, and V. Kindratenko.

For information on obtaining reprints of this article, please send e-mail to: [tpds@computer.org](mailto:tpds@computer.org), and reference IEEECS Log Number TPDISS-2009-09-0442.

Digital Object Identifier no. 10.1109/TPDS.2010.62.

## 2 CUDA PROGRAMMING

The Compute Unified Device Architecture provides a programming model that is ANSI C, extended with several keywords and constructs [2]. The programmer writes a

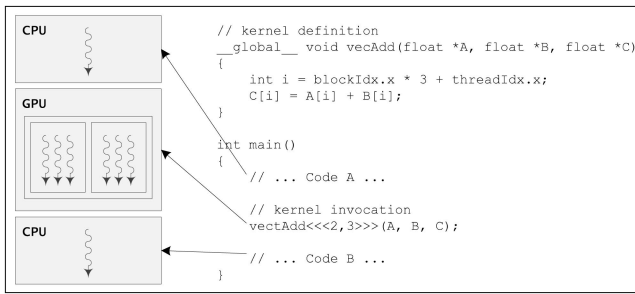


Fig. 1. Kernel definition and invocation in CUDA.

single source program that contains both the host (CPU) code and the device (GPU) code. These two parts are automatically separated and compiled by the CUDA compiler tool chain [6].

CUDA allows the programmer to write device code in C functions called *kernels*. A kernel is different from a regular function in that it is executed by many GPU threads in a Single-Instruction Multiple-Data (SIMD) fashion. Each thread executes the entire kernel once. Fig. 1 shows an example that performs vector addition in GPU. Launching a kernel for GPU execution is similar to calling the kernel function, except that the programmer needs to specify the space of GPU threads that execute it, called a *grid*. A grid contains multiple *thread blocks*, organized in a 2D space. Each block contains multiple threads, organized in a 3D space. In the example (Fig. 1), the grid contains  $2 \times 1$  blocks, each containing  $3 \times 1 \times 1$  threads, so the kernel is executed by six threads in total. Each GPU thread is given a unique *thread ID* that is accessible within the kernel, through the built-in variables `blockIdx` and `threadIdx`. They are vectors that specify an index into the block space (that forms the grid) and the thread space (that forms a block), respectively. In the example, each thread uses its ID to select a distinct vector element for addition. It is worth noting that blocks are required to execute independently because the GPU does not guarantee any execution order among them. However, threads within a block can synchronize through a *barrier* [2].

GPU threads have access to multiple GPU memories during kernel execution. Each thread can read and/or write its private *registers* and *local memory* (for spilled registers). With single-cycle access time, registers are the fastest in the GPU memory hierarchy. In contrast, local memory is the slowest in the hierarchy, with more than 200-cycle latency. Each thread block has its private *shared memory*. All threads in the block have read and write access to this memory, which is as fast as registers. Globally, all threads have read and write access to the *global memory*, and read-only access to the *constant memory* and the *texture memory*. The three memories have the same access latency as the local memory, and are the only GPU memories accessible from the host. Since the GPU threads cannot access the host memory, the data needed by a kernel must be transferred to these GPU memories before it is launched.

To write a CUDA program, the programmer typically starts from a sequential version and proceeds through the following steps:

1. Identify a kernel, and package it as a separate function.

2. Specify the grid of GPU threads that executes the kernel, and partition the computation among these threads, by using `blockIdx` and `threadIdx` inside the kernel function.
3. Manage data transfer between the host memory and the GPU memories (global, constant, and texture), before and after the kernel invocation.
4. Perform memory optimizations in the kernel, e.g., caching data in the shared memory and coalescing accesses to the global memory [2], [5].
5. Perform other optimizations in the kernel in order to achieve an optimal balance between single-thread performance and the level of parallelism [7].

Note that the above steps must be applied to each kernel in the program.

Most of the above steps in the procedure involve significant code changes that are tedious and error-prone, not to mention the difficulty in finding the “right” set of optimizations to achieve the best performance [7]. This not only increases development time, but also makes the program difficult to understand and to maintain. Consider the matrix multiply code in CUDA, shown in Fig. 2. It is nonintuitive to picture the kernel computation as a whole through explicit specification of what each thread does. Also, management and optimization on data in GPU memories involve heavy manipulation of array indices, which increases the likelihood of coding mistakes and prolongs program development and debugging.

### 3 *hiCUDA* THROUGH AN EXAMPLE

We illustrate the use of *hiCUDA* directives with the popular matrix multiply code shown in Fig. 3. The code computes the product  $64 \times 32$  matrix *C* of two matrices *A* and *B* of dimensions  $64 \times 128$  and  $128 \times 32$ , respectively. We map the massive parallelism, available in the triply nested loops (*i,j,k*) (lines 10-18), onto the GPU. The resulting *hiCUDA* program is shown in Fig. 4.

The loop nest (*i,j,k*) is surrounded by the kernel directive (lines 13 and 33 of Fig. 4). The directive gives a name for the kernel (`matrixMul`) and specifies the shape and size of the grid of GPU threads. More specifically, it specifies a 2D grid, consisting of  $4 \times 2$  thread blocks, each of which contains  $16 \times 16$  threads.

The parallelism in the matrix multiply code is exploited by dividing the iterations of the *i* and *j* loops among the threads. The `loop_partition` directives (lines 15 and 17 of Fig. 4) are used for this purpose. With the `over_tblock` clause, the iterations of the *i* and *j* loops are distributed over the first and second dimension of the thread-block space (i.e., the  $4 \times 2$  thread blocks), respectively. Thus, each thread block executes a  $64/4 \times 32/2$  or a  $16 \times 16$  tile of the iteration space of loops *i* and *j*. Furthermore, with the `over_thread` clause, the iterations of loop *i* and *j* that are assigned to each thread block are distributed over the first and second dimension of the thread space (i.e., the  $16 \times 16$  threads), respectively. Thus, each thread executes a  $16/16 \times 16/16$  tile or a single iteration in the  $16 \times 16$  tile assigned to the thread block. This partitioning scheme is shown in Fig. 5a.

The arrays *A*, *B*, and *C* must be allocated in the global memory of the GPU device. Further, the values of *A* and *B*,

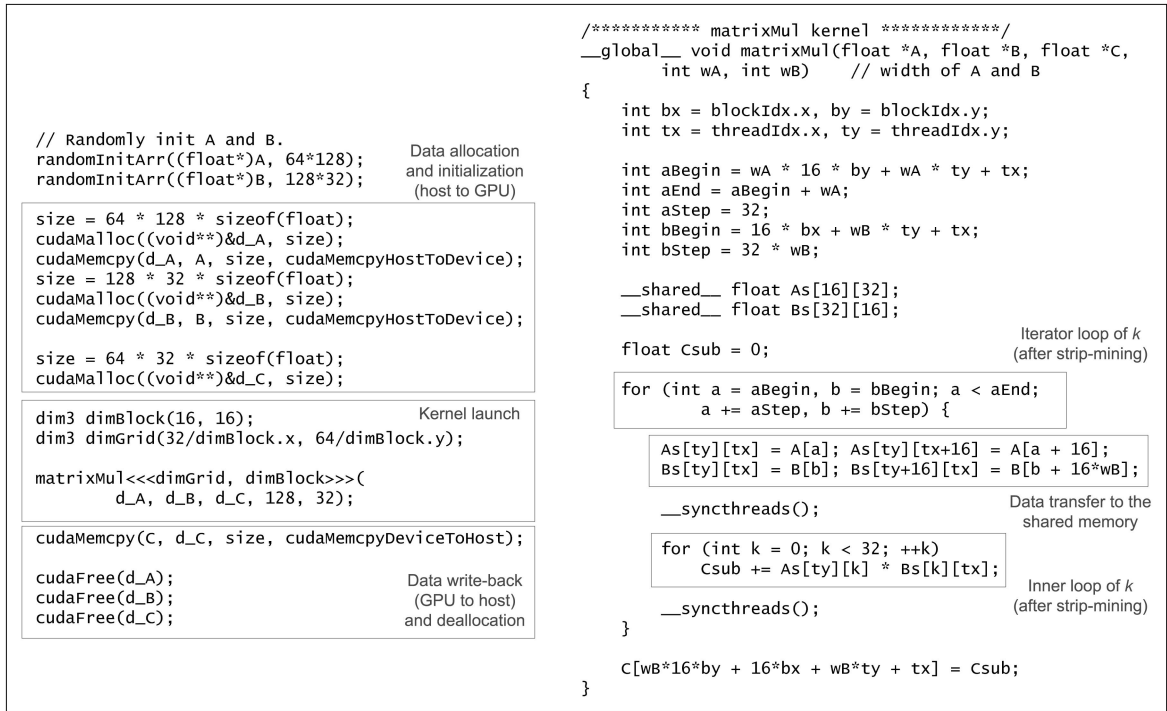


Fig. 2. Matrix multiply in CUDA.

initialized on the host, must be copied to their corresponding global memory locations before the kernel is launched. Similarly, the results computed in C must be copied out of the global memory back to the host memory after the kernel is done. All of this is accomplished using the `global` directive (lines 9-11 and 35-37 of Fig. 4). The `alloc` clause in the directive for each array specifies that an array of the same size is allocated in the global memory. The `copyin` clauses in the directives for A and B (line 9 and 10) indicate that the arrays are copied from the host memory to the global memory. Similarly, the `copyout` clause in the directive for C (line 35) copies C from the global memory back to the host memory.

The performance of the `matrixMul` kernel can be improved by utilizing the shared memory on the GPU [2]. The data needed by all threads in a thread block (i.e., 16 rows of A and 16 columns of B) can be loaded into the shared

memory before they are used, reducing access latency to memory. Since this amount of data is too large to fit in the shared memory at once, it must be loaded and processed in batches, as illustrated in Fig. 5b. This scheme can be implemented in two steps. First, loop k is strip-mined so that the inner loop has 32 iterations. Second, two shared directives (lines 21 and 22 of Fig. 4) are inserted between the resulting loops to copy data from the global memory to the shared memory. The sections of A and B specified in the

```

1 float A[64][128];
2 float B[128][32];
3 float C[64][32];
4
5 // Randomly init A and B.
6 randomInitArr((float*)A, 64*128);
7 randomInitArr((float*)B, 128*32);
8
9 // C = A * B
10 for (i = 0; i < 64; ++i) {
11     for (j = 0; j < 32; ++j) {
12         float sum = 0;
13         for (k = 0; k < 128; ++k) {
14             sum += A[i][k] * B[k][j];
15         }
16         C[i][j] = sum;
17     }
18 }
19
20 printMatrix((float*)C, 64, 32);

```

Fig. 3. The original matrix multiply program.

```

1 float A[64][128];
2 float B[128][32];
3 float C[64][32];
4
5 // Randomly init A and B.
6 randomInitArr((float*)A, 64*128);
7 randomInitArr((float*)B, 128*32);
8
9 #pragma hcuda global alloc A[*][*] copyin
10 #pragma hcuda global alloc B[*][*] copyin
11 #pragma hcuda global alloc C[*][*]
12
13 #pragma hcuda kernel matrixMul tblock(4,2) thread(16,16)
14 // C = A * B
15 #pragma hcuda loop_partition over_tblock over_thread
16 for (i = 0; i < 64; ++i) {
17     #pragma hcuda loop_partition over_tblock over_thread
18     for (j = 0; j < 32; ++j) {
19         float sum = 0;
20         for (kk = 0; kk < 128; kk += 32) {
21             #pragma hcuda shared alloc A[i][kk:kk+31] copyin
22             #pragma hcuda shared alloc B[kk:kk+31][j] copyin
23             #pragma hcuda barrier
24             for (k = 0; k < 32; ++k) {
25                 sum += A[i][kk+k] * B[kk+k][j];
26             }
27             #pragma hcuda barrier
28             #pragma hcuda shared remove A B
29             C[i][j] = sum;
30         }
31     }
32 }
33 #pragma hcuda kernel_end
34
35 #pragma hcuda global copyout C[*][*]
36
37 #pragma hcuda global free A B C
38
39 printMatrix((float*)C, 64, 32);

```

Fig. 4. The *hi*CUDA matrix multiply program.

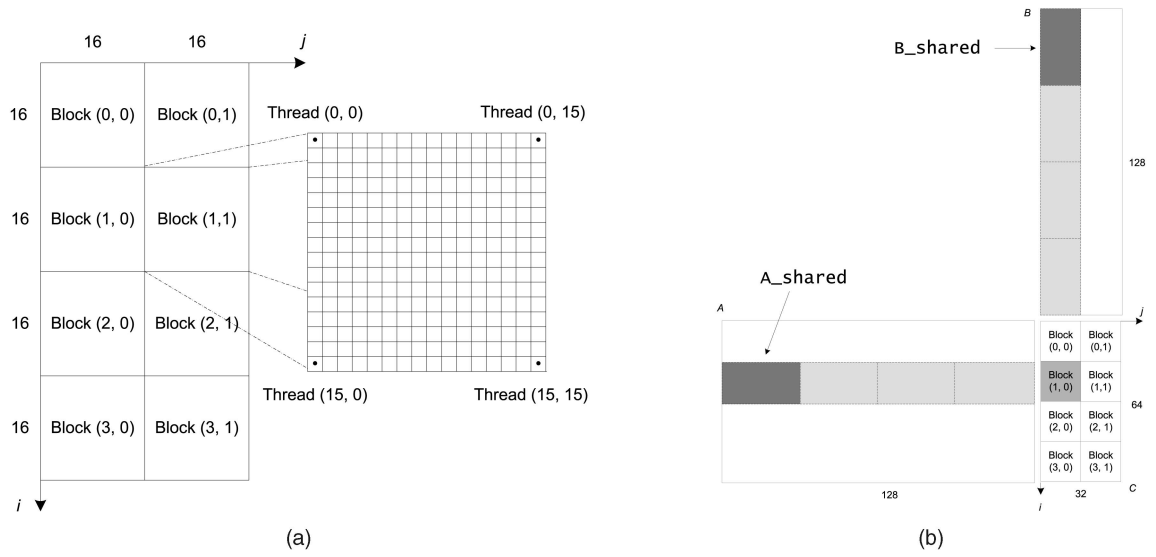


Fig. 5. The scheme of accelerating the matrix multiply code on GPU. (a) Partitioning the computation. (b) Utilizing the shared memory.

directives (i.e., a  $1 \times 32$  tile of A and a  $32 \times 1$  tile of B) represent the data to be brought into the shared memory for *each iteration* of the loop nest ( $i, j, kk$ ). Based on this information, the *hiCUDA* compiler determines the actual size and shape of the shared memory variable to be allocated, taking into account the fact that *multiple* iterations are executed *concurrently* by the threads in a thread block. In this case, it allocates one variable for holding a  $16 \times 32$  tile of A and another for holding a  $32 \times 16$  tile of B. The details of the shared directive appear in Section 4.2.

When compared with the handwritten CUDA version for matrix multiply (Fig. 2), we believe that the *hiCUDA* code is simpler to write, to understand, and to maintain. Although the programmers can further “clean up” the CUDA version by using macros and templates, they still have to extract the kernel code from the host code and use explicit thread indices to partition computations, which is much simpler to do in *hiCUDA*. Further, the *hiCUDA* code maintains the structure of the sequential version (i.e., the loop nest), making it easier to comprehend the code and reason about its correctness. Also, using *hiCUDA* directives eliminates the possibility of some CUDA-specific bugs (e.g., incorrect array indexing due to complex work partitioning among the threads), thus reducing the burden of parallel debugging on the programmers. Most importantly, *hiCUDA* supports the same programming paradigm already familiar to CUDA programmers.

The matrix multiplication example illustrates only the basic use of *hiCUDA* directives. The directives allow for more complex partitioning of computations and for more sophisticated movement of data. For example, the data directives support transfer of array sections between the global (or constant) memory and the host memory, which is tedious to express in CUDA.

## 4 THE *hiCUDA* DIRECTIVES

*hiCUDA* presents the programmer with a *computation* model and a *data* model. The computation model allows

the programmer to identify code regions that are intended to be executed on the GPU and to specify how they are to be executed in parallel. The data model allows programmers to allocate and deallocate memory on the GPU and to move data back and forth between the host memory and the GPU memory.

The *hiCUDA* directives are specified using the pragma mechanism provided by the C and C++ standards [8]. Each directive starts with `#pragma hicuda` and is case-sensitive. Preprocessing tokens following `#pragma hicuda` are subject to macro replacement. Variables referenced inside a *hiCUDA* directive must be visible at the place of the directive.

### 4.1 Computation Model

*hiCUDA* provides four directives in its computation model: `kernel`, `loop_partition`, `singular`, and `barrier`.

The programmer identifies a code region for GPU execution by enclosing it with two kernel directives, as shown below:

```
#pragma hicuda kernel kernel-name \
    thread-block-clause thread-clause [nowait]
    sequential-code
#pragma hicuda kernel_end,
```

where *kernel-name* is the name of the kernel function to be created, and *thread-block-clause* and *thread-clause* specify a virtual grid of GPU threads with the following format:

```
tblock(dim-sz{, dim-sz} *)
thread(dim-sz{, dim-sz} *),
```

where *dim-sz* is an integer expression that represents the size of a dimension in the virtual thread-block or thread space.<sup>1</sup>

1. In the specification of *hiCUDA* directives, terminal symbols are shown in typewriter font. [...] enclose optional tokens. | means OR. {...} considers the enclosed token(s) as a group. {...} \* represents zero or more repetition of the token group. {...} + represents one or more repetition of the token group.

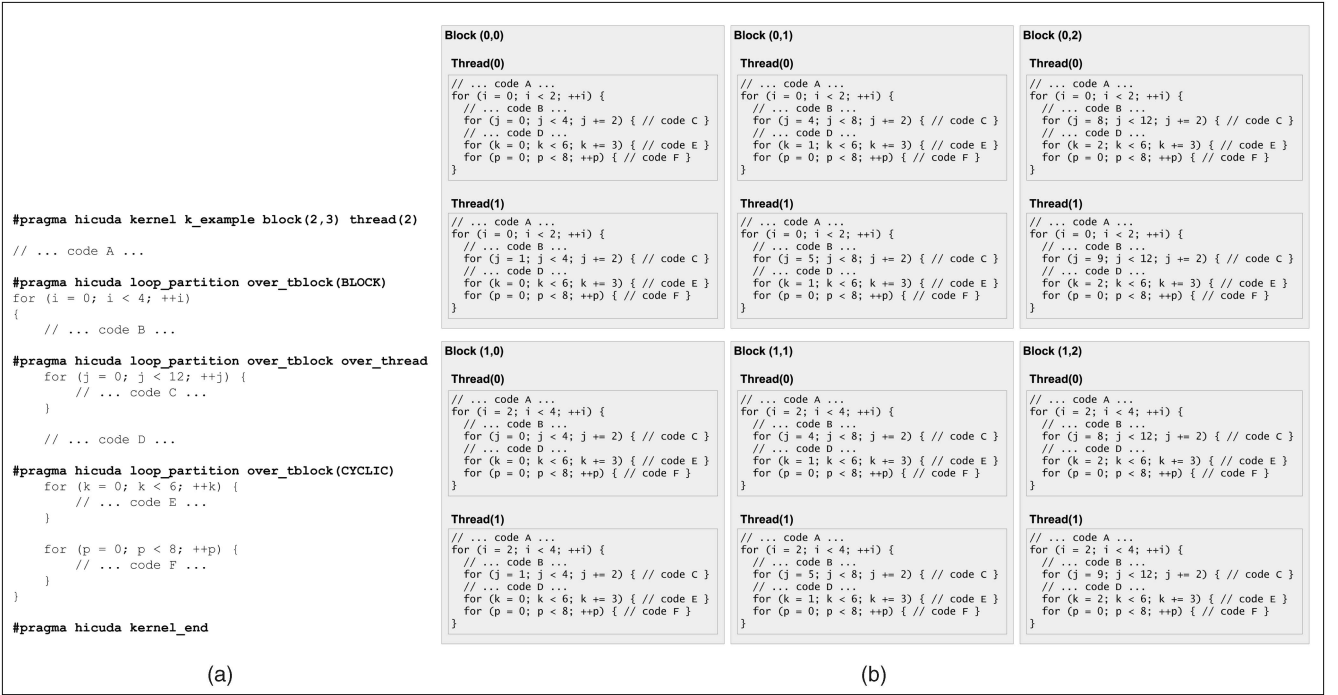


Fig. 6. Example use of the `loop_partition` directives. (a) A kernel in a *hiCUDA* program. (b) Code executed by each GPU thread.

The `kernel` directive specifies that *sequential-code* is to be extracted in a kernel function named *kernel-name* and replaced by an invocation to this kernel. The grid of GPU threads that executes the kernel contains  $B_1 \times B_2 \times \dots \times B_n$  thread blocks, where  $B_i$  is the  $i$ th *dim-sz* specified in *thread-block-clause*. Similarly, each thread block contains  $T_1 \times T_2 \times \dots \times T_m$  threads, where  $T_i$  is the  $i$ th *dim-sz* specified in *thread-clause*. The integers  $n$  and  $m$  are the dimensionality of the virtual thread-block and thread space, respectively, and can be arbitrarily large. By default, the host thread waits for the kernel to finish before executing code after the kernel region. If the `nowait` clause is present, the host thread proceeds asynchronously after launching the kernel, and waits for it to finish at the next *hiCUDA* directive immediately after the kernel region.

With the `kernel` directive, the kernel in its entirety is executed by each thread. To exploit parallelism in the kernel, the programmer must divide its computations among GPU threads. This can be done using the `loop_partition` directive, which distributes loop iterations. It has the following syntax:

```
#pragma hicuda loop_partition \
    [over_tblock [(distr-type)]] [over_thread]
for loop.
```

At least one of the `over_tblock` and `over_thread` clauses must be present. In the `over_tblock` clause, *distr-type* specifies one of the two strategies of distributing loop iterations: blocking (BLOCK) and cyclic (CYCLIC). If it is omitted, the default distribution strategy is BLOCK. The `over_thread` clause does not have such an option: the distribution strategy is always CYCLIC. The rationale behind this restriction is explained later in the section.

The loops associated with `loop_partition` directives can be arbitrarily nested within each other. For each

directive with the `over_tblock` clause, its *thread-block nesting level* (or *LB*) is defined to be the nesting level (starting from 1) with respect to enclosing directives that also have the `over_tblock` clause. Similarly, for each directive with the `over_thread` clause, its *thread nesting level* (or *LT*) is defined to be the nesting level (starting from 1) with respect to enclosing directives that also have the `over_thread` clause. Fig. 6 shows an example of how the `loop_partition` directive works. The directives for loop *i*, *j*, and *k* have *LB* defined, which are 1, 2, and 2, respectively.

The `over_tblock` clause specifies that the iteration space of *for loop* is distributed over the *LB*th dimension of the virtual thread-block space (specified in the enclosing `kernel` directive). The thread blocks are divided into  $B_{LB}$  groups according to the index in the *LB*th dimension, where  $B_i$  is defined previously. Each group is assigned a subset of loop iterations to execute, determined by the distribution strategy specified in the clause. These iterations are executed by every thread block in the group. Similarly, the `over_thread` specifies that the loop iterations assigned to each thread block are distributed over the *LT*th dimension of the virtual thread space. More details of the clauses' semantics can be found in [9]. In the example (Fig. 6), the `loop_partition` directive for loop *j* has *LB* = 2 and *LT* = 1. Therefore, the iteration space for *j* is first distributed over the second dimension of the thread-block space, whose size is 2. Based on the blocking distribution strategy, the thread blocks (0, \*), (1, \*), and (2, \*) are responsible for iterations 0-3, 4-7, and 8-11, respectively. Within each thread block, its assigned loop iterations are cyclically distributed over two threads. Thus, each thread executes 4/2 or two iterations of loop *j*. It is worth noting that the `loop_partition` directive supports uneven distribution of iterations over the threads. The *hiCUDA* compiler automatically generates "guard" code to ensure the correct number of iterations being executed.

In designing the `loop_partition` directive, we restrict the distribution strategy for the `over_thread` clause to be cyclic. This ensures that *contiguous* loop iterations are executed concurrently. Since contiguous iterations tend to access contiguous data, this strategy allows for various memory optimizations, such as utilizing the shared memory, and coalescing accesses to the global memory.

By default, any code that is not partitioned among the threads is executed by *every* thread. An example would be loop `p` in Fig. 6a. Sometimes, however, this redundant execution could cause incorrect result or degraded performance. *hiCUDA* allows the programmer to identify kernel code to be executed *only once* in a thread block, by enclosing it with two singular directives:

```
#pragma hicuda singular
    sequential-kernel-code
#pragma hicuda singular_end.
```

Note that *sequential-kernel-code* cannot be partitioned, i.e., it cannot contain any `loop_partition` directives. If loop `p` in Fig. 6a were surrounded by the singular directives, only Thread(0) in each thread block executes this loop (for each iteration of loop `i` assigned). It is worth noting that this directive does not guarantee that *sequential-kernel-code* is executed once across all thread blocks. For example, the same loop `p` is executed by all Block(0,\*). This behavior meets the common scenarios a singular directive is used for, i.e., for initialization or “summary” code. They usually have to be executed once in *each* thread block, because the thread blocks are independent and do not share data.

Finally, the `barrier` directive provides barrier synchronization for all threads in each block, at the place of the directive.

## 4.2 Data Model

*hiCUDA* provides four main directives in its data model: `global`, `constant`, `texture`, and `shared`. Each directive manages the life cycle of variables in the corresponding GPU memory.

Since data management in the device memory happens before and after kernel execution, the `global`, `constant`, and `texture` directives must be placed outside kernel regions. In contrast, the `shared` memory is explicitly managed within the kernel code, so the `shared` directive must be placed inside a kernel region. All four directives are stand-alone, and the associated actions happen at the place of the directive.

To support the use of dynamically allocated arrays, *hiCUDA* also provides a `shape` directive, allowing the user to specify the dimension sizes of these arrays:

```
#pragma hicuda shape ptr-var {[dim-sz]}+,
```

where *ptr-var* refers to a pointer variable in the sequential program, and *dim-sz* is the size of an array dimension. The directive is not associated with any actions, but allows pointer variables to be used in the main data directives. The visibility rule for a `shape` directive with respect to main data directives is the same as that for a local C variable with respect to statements that refer to this variable.

The `global` directive takes one of the following three forms:

```
#pragma hicuda global alloc variable \
    [copyin [variable]][clear]
#pragma hicuda global copyout variable
#pragma hicuda global free var-sym+
    variable := var-sym[start-idx:end-idx]*,
```

where *var-sym* refers to a variable in the sequential program, and *variable* consists of a *var-sym* followed by a section specification if the variable is an array (static or dynamic). The index range of each dimension of the section is contiguous (i.e., with unit stride) and includes both ends. Apart from the standard form `[start-idx : end-idx]`, *hiCUDA* supports two short forms:

- `[start-idx]`, which is equivalent to `[start-idx : start-idx]`
- `[*]`, which represents the entire dimension range, i.e., `[0 : dim-sz-1]`.

The first form of the `global` directive specifies that a copy of *variable* in the `alloc` clause is to be allocated in the global memory. If the `copyin` clause is present, the content of the *variable* in this clause (or the *variable* in the `alloc` clause if omitted) is to be copied to the corresponding portion of the newly allocated global memory variable. Note that the two *variables* must refer to the same *var-sym*. If the `clear` clause is present instead, the allocated global memory region is to be initialized to 0. The second form of the directive specifies that the global memory region corresponding to *variable* in the `copyout` clause is to be copied to the host memory region for this *variable*. The third form of the directive specifies that the global memory variable corresponding to each *var-sym* is to be deallocated. In both the `copyout` and the `free` clauses, the global memory variable corresponding to *variable* (or *var-sym*) refers to the one created by the matching `global` directive (i.e., for *variable*) in its first form. This matching directive must appear in the same *lexical scope* as the one with `copyout` or `free` clause. It is worth noting that the *global directive never exposes any global memory variable to the programmer*. This reduces the programming burden and facilitates automatic management by the compiler. This is consistent over all *hiCUDA* data directives.

In many applications, it is not necessary to transfer an entire array to/from the global memory, so all data directives provide a way to specify rectangular array sections. Fig. 7 shows an example: the *jacobi* benchmark. The array `A` stores the initialized data while `B` is a temporary array. Since the peripheral elements of `B` are not used in the computation, they need not exist in the global memory. Also, the peripheral elements of `A` never change during the computation, so they do not need to be copied back to the host memory. These optimizations can be achieved through simple changes in the array specification in the `global` directives (lines 7 and 23 of Fig. 7).

The `constant` and `texture` directives are similar to the `global` directive, and are described in [9].

The `shared` directive also looks similar to the `global` directive, with one of the following forms:

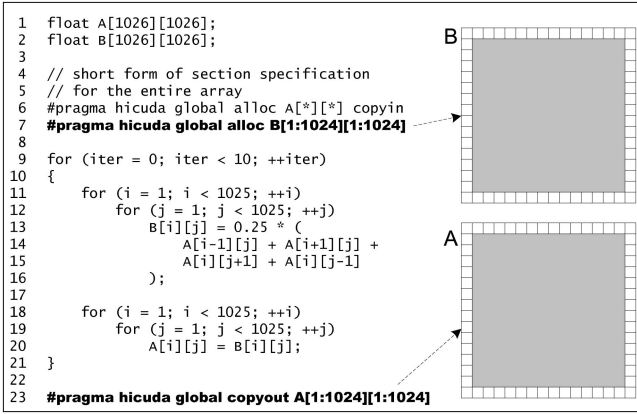


Fig. 7. Array section specification in the `global` directives in the `jacobi` program.

```

#pragma hcuda shared alloc variable \
    [copyin [(nobndcheck)] [variable]]
#pragma hcuda shared \
    copyout [(nobndcheck)] variable
#pragma hcuda shared remove {var-sym} + .

```

The semantics of the `shared` directive are different from those of the `global` directives, in determining the type and shape of the GPU memory region to be allocated, initialized, or written-back. In both directives, *variable* in the `alloc` clause specifies the array section to be allocated in the `global` or `shared` memory when *sequential* execution reaches the place of the directive. Since `global` directives are placed outside kernels, where the execution model is still sequential, *variable* directly implies the global memory variable to be created. However, `shared` directives are placed within kernels, in which multiple loop iterations are executed by multiple threads concurrently. In this case, the shared memory variable must be big enough to hold the *variables* for all concurrently executed iterations. Consider the example shown in Fig. 8a: a `shared` directive is put inside a simple kernel loop that is distributed over three threads (assuming that only one thread block is used to execute the kernel). This directive specifies that `A[i - 1 : i + 1]` should be loaded into the shared memory at the beginning of each iteration of loop `i`. Since the threads concurrently execute three contiguous iterations at any given time, the `hiCUDA` compiler merges `A[i - 1 : i + 1]` with respect to a three-value range of `i : [ii : ii + 2]`, where `ii` is the iterator for batches of contiguous iterations. Thus, the shared memory variable `As` is a five-element array.

Not only is the shared memory variable obtained by merging *variable* in the `alloc` clause, the region of this variable to be initialized (or written-back) is also a “merged” version of *variable* in the `copyin` (or `copyout`) clause. In all these clauses, the array region specified in *variable* does not have to be within the array bound in all cases. For example, in Fig. 8a, the `shared` directive attempts to load `A[-1]` (when `i = 0`) and `A[6]` (when `i = 5`). By default, the `hiCUDA` compiler automatically generates code that guards against invalid accesses if necessary. Since the decision made by the compiler can be too conservative, `hiCUDA` allows the programmer to optimize code generation by disabling array-bound check through a `nobndcheck` option in the `copyin` or `copyout` clause.

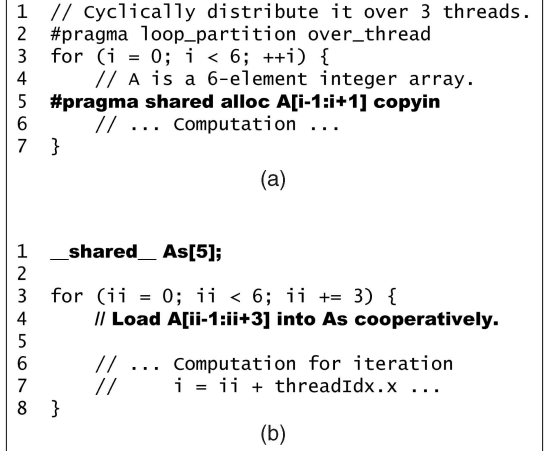


Fig. 8. Semantics of the `shared` directive. (a) A kernel loop containing a `shared` directive. (b) Code executed by each GPU thread.

## 5 THE *hiCUDA* COMPILER

The `hiCUDA` compiler takes as input a sequential C program with `hiCUDA` directives and produces an equivalent CUDA program. The compiler supports code with regular array accesses and that contains multiple functions. It can handle kernels with function calls in their bodies as well as with related data directives in different functions. The compiler also supports arrays that are dynamically allocated, but ignores aliasing among such arrays since GPU-friendly applications typically do not have such aliasings.

The `hiCUDA` compiler performs a number of key steps in the translation process:

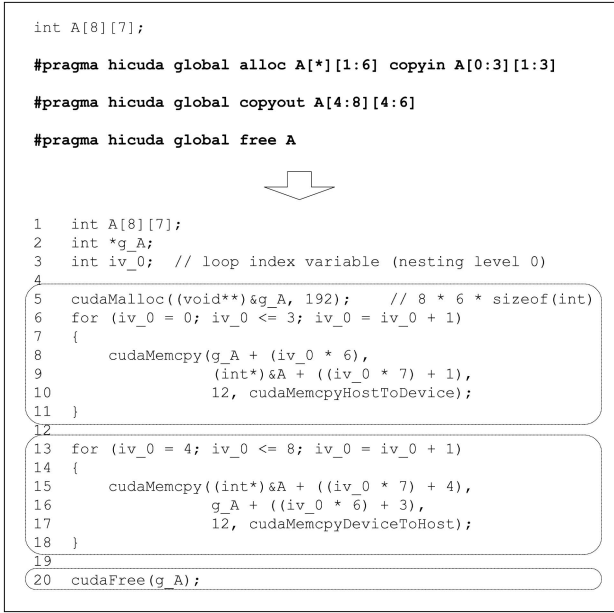
**GPU data management.** The compiler is responsible for generating code that allocates data on device memory as also for transfer of data between the host and device memories, as specified by the `global` and `constant` directives.

**Kernel data access redirection.** In order for a kernel region to be executed on the GPU, the compiler must redirect data accesses made in the region to corresponding GPU memory variables (associated with `global` or `constant` directives). Before doing the actual access redirection, the compiler must perform two analyses: one is to determine all `global` and `constant` directives that *reach* this kernel region, or *reaching directives analysis*; the other is to collect all scalar and array accesses made in the region, or *data access analysis*.

**Kernel loop partitioning.** The compiler translates each `loop_partition` directive in a kernel function, by modifying the bounds and step of the associated loop so that they represent the iterations executed by each GPU thread.

**The use of shared memory.** The compiler generates code that caches global memory data in the shared memory, as specified by `shared` directives. As you will see, this translation process is more complicated than that of `global` directives.

**Outlining of kernel region.** Once the code of a kernel region is transformed into what each GPU thread executes, the compiler extracts it into a separate kernel function and replaces the original region with an execution configuration to this kernel function. Performing this transformation in a robust and scalable fashion is nontrivial [10].



```

int A[8][7];

#pragma hicuda global alloc A[*][1:6] copyin A[0:3][1:3]

#pragma hicuda global copyout A[4:8][4:6]

#pragma hicuda global free A

1  int A[8][7];
2  int *g_A;
3  int iv_0; // loop index variable (nesting level 0)
4
5  cudaMalloc((void**)&g_A, 192); // 8 * 6 * sizeof(int)
6  for (iv_0 = 0; iv_0 <= 3; iv_0 = iv_0 + 1)
7  {
8      cudaMemcpy(g_A + (iv_0 * 6),
9                (int*)&A + ((iv_0 * 7) + 1),
10               12, cudaMemcpyHostToDevice);
11 }
12
13 for (iv_0 = 4; iv_0 <= 8; iv_0 = iv_0 + 1)
14 {
15     cudaMemcpy((int*)&A + ((iv_0 * 7) + 4),
16               g_A + ((iv_0 * 6) + 3),
17               12, cudaMemcpyDeviceToHost);
18 }
19
20 cudaFree(g_A);

```

Fig. 9. Translation of global directives.

In the following sections, we first describe the details of each step in an intraprocedural context, and then, in Section 5.7, focus on the additional analyses required (in each step) to support real-world applications that can span multiple functions and use dynamically allocated arrays.

## 5.1 GPU Data Management

The global and constant directives are directly translated into CUDA code in place. Fig. 9 shows an example of the translation of global directives.

For each global directive with an alloc clause, the compiler creates a *global memory variable*  $g\_H$  corresponding to the host variable  $H$  (in the alloc clause).  $g\_H$  is declared as a local pointer variable (line 2 in Fig. 9). The compiler dynamically allocates global memory for this variable by inserting a call to the CUDA runtime library: `cudaMalloc` (line 5). The size of the allocated memory is based on the section specified in the clause. Similarly, for a global directive with the free clause, the compiler inserts a call to `cudaFree` (line 20). To generate code that transfers data between the host and global memory, as specified by a `copyin` or `copyout` clause, the compiler inserts a call to `cudaMemcpy` (lines 10 and 17). Since this call can only transfer a *contiguous* chunk of data, the compiler must generate code that invokes the call multiple times (i.e., inside a loop nest) in order to transfer a noncontiguous data section. In the example (Fig. 9), the biggest contiguous section that can be “copied-in” using one call of `cudaMemcpy` is `A[iv_0][1:3]`, where `iv_0 = 0..3`.

The handling of constant directives is slightly different from that of global directives because a constant memory variable must be declared and allocated statically. Since this memory region persists outside the directive scope, it can be reused for other constant directives with nonoverlapping scopes. A graphic-coloring algorithm (similar to the one used in register allocation) is employed to determine the smallest amount of constant memory to be allocated that can satisfy all constant directives. Section 5.6 gives the details of the algorithm.

## 5.2 Kernel Data Access Redirection

In order to redirect data accesses made in a kernel region to GPU memory variables, the compiler first performs reaching directives analysis and kernel data access analysis.

In reaching directives analysis, the compiler determines all GPU memory variables that are visible to each kernel region. The scope of a GPU memory variable is bounded by the associated `global` or `constant` directive in the first form and the matching directive with the `free` clause. Since these two directives must be in the same *lexical* scope, it is straightforward for the compiler to check whether or not a kernel region is within the scope of a GPU memory variable.

In kernel data access analysis, the compiler determines the sections of arrays accessed in a kernel region by *projecting* each array access onto index variables of enclosing loops up to the kernel region boundary. Note that this analysis cannot handle irregular accesses like `A[B[i]]` and must conservatively assume that the entire array is accessed in such cases. Data flow analysis is used to identify scalar variables accessed (or modified) by the kernel region, i.e., those with at least one def-use chain [11] going across the region boundary. With the data access summary of a kernel region in place, the compiler then determines how each access is redirected to the GPU memory. The targets of redirection are GPU memory variables associated with reaching global and constant directives, which are matched with kernel data accesses by variable symbol. If possible, the compiler checks that each access is covered by the data brought into the GPU memory by the matched directive. The only exception is read-only scalar accesses, which can be redirected to parameters of the kernel function to be created, if no matched global or constant directive is found. Note that, during this matching process, the compiler emits an error when ambiguity occurs, e.g., when a variable accessed in a kernel region is covered by both a global and a constant directive.

The actual access redirection is straightforward. Consider array  $A$  in Fig. 9. Any access `A[i][j]` in a kernel region is replaced with `g_A[i][j - 1]`. Since  $g\_A$  is a dynamically allocated array, the actual code generated has a 1D access offset, i.e., `g_A[i * 6 + (j - 1)]`.

## 5.3 Kernel Loop Partitioning

For each `loop_partition` directive in a kernel region, the compiler modifies the bounds and step of the associated loop so that it represents the iterations executed by each GPU thread. This process is straightforward when the distribution strategy is `CYCLIC` (among either thread blocks or threads). For example, distributing a loop `for (i = 0; i < 5; ++i)` among three threads cyclically results in a new loop `for (i = threadIdx.x; i < 5; i += 3)`. However, code generation for a loop distributed among thread blocks in a `BLOCK` fashion is more complicated. In such a case, the compiler has to determine the number of consecutive iterations assigned to each thread block, and insert guard code to prevent certain thread blocks to execute extra iterations if it is not certain that the distribution is even. For example, if the previous loop were distributed among three thread blocks in a `BLOCK` fashion, the resulting loop would be transformed into:



```

i_end = 2 * blockIdx.x + 2;
if (i_end > 5) i_end = 5;
for (i = 2 * blockIdx.x; i < i_end; ++i)

```

As an optimization, the compiler eliminates the loop when each thread executes at most one iteration. It generates a guard if some threads do not have any iterations to execute. Details of this optimization are described in [9].

Finally, the number of thread blocks and threads to which a loop is distributed is determined by the geometry of the thread block and thread spaces of the enclosing kernel region, and the nesting level of the `loop_partition` directive. The construction of these quantities, in terms of `blockIdx` and `threadIdx`, involves mapping the virtual thread block and thread spaces onto the physical ones supported by CUDA, which is described in Section 5.5.

#### 5.4 The Use of Shared Memory

*hiCUDA* allows the programmer to cache global memory data in the shared memory by using a `shared` directive. There are two challenges that make the translation of a `shared` directive more complicated than that of a `global` or a `constant` directive. First, a `shared` directive specifies the section to be allocated (or transferred) in *sequential execution* while the kernel code is executed by multiple threads *concurrently*. Therefore, the compiler must merge the section with respect to all iterations of the enclosing loop nest that are concurrently executed in a thread block, which depends on how these loops are partitioned as specified by `loop_partition` directives. Consider the `shared` directive for `A` in the matrix multiply example (Fig. 4). To determine the array section to be allocated in the shared memory (for each thread block), the compiler projects the section `A[i][kk:kk+31]` onto the range of each enclosing loop's index variable (i.e., `i` and `j`) that represents concurrently executed iterations in the thread block. The range for `i` can be expressed as `[i-threadIdx.y:i-threadIdx.y+15]`, where `i` is the loop index variable *after* its `loop_partition` directive is handled. Therefore, the merged section is `A[i-threadIdx.y:i-threadIdx.y+15][kk:kk+31]`, a  $16 \times 32$  section.

The second challenge of handling a `shared` directive is that data transfer between the shared memory and the global memory can be done *cooperatively* by all threads (in a thread block). In order to achieve optimal performance of both memories during the transfer, the compiler must generate code that respects the access pattern requirements imposed by CUDA. More specifically, the writes to the shared memory (by a half-warp of threads) must be free of bank conflicts, and the reads from the global memory (by a half-warp of threads) must be *coalesced*, i.e., must form a contiguous segment with proper alignment, which must also be accessed by these threads in order [2]. To meet these constraints, the compiler divides the section to be transferred into segments of  $W = 32$  elements that are contiguous in both the shared and the global memory, where  $W$  is the size of a thread warp. The compiler generates code that assigns these segments to all warps (in a thread block) in a cyclic fashion. Within a segment, each element is assigned to each thread in the warp in the order of the thread index. Each segment is guaranteed to have the proper alignment required for

coalescing; thus, some of the segments may partially fall outside the section to be transferred. In such cases, the compiler generates guard code to ensure that these “extra” elements are not transferred. This scheme clearly achieves coalescing reads from the global memory. It also avoids bank conflicts in shared memory because consecutive elements (accessed by a half-warp) are mapped to different banks and the shared memory has exactly  $W/2 = 16$  banks.

#### 5.5 Kernel Outlining

Once a kernel region has been transformed into the code each thread executes, it is ready to be outlined into a separate kernel function. Note that, before this process, the compiler maps `singular` and `barrier` directives to CUDA code, which is straightforward.

The outlining process consists of three steps. First, the compiler determines the parameters of the new kernel function, which consists of GPU memory variables the kernel region needs and scalar variables it reads (that do not exist in any GPU memories). Second, the compiler creates a kernel function (with the `__global__` attribute) that contains the kernel region as the function body. Last, the compiler replaces the original kernel region with an invocation to the kernel function.

Two local variables of CUDA runtime type `dim3` are created in the procedure that contains the kernel region, to hold the geometry of the CUDA thread block and thread spaces, respectively. Their values are determined by mapping the virtual thread block and thread spaces (specified in the `kernel` directive) to the physical ones supported by CUDA. This process starts from the leftmost (or outermost) dimension of the virtual thread block (or thread) space and assigns each dimension to `gridDim.x` (or `blockDim.z`), `gridDim.y` (or `blockDim.y`), and `gridDim.z` (or `blockDim.x`) in sequence. If the virtual space's dimensionality exceeds the physical one's, `gridDim.x` (or `blockDim.x`) holds a collapsed version of *all* remaining virtual dimensions.

#### 5.6 Optimization of GPU Memory Allocation

When creating the shared memory variable associated with a `shared` directive, the compiler statically allocates a *new* region in the shared memory. This scheme clearly does not utilize the memory efficiently because the live range of a shared memory variable is often shorter than the entire kernel and its memory region can be reused for other variables with nonoverlapping live ranges. The same problem exists for constant memory variables because they are also statically allocated.

Therefore, we develop an optimized scheme, in which a *single* shared memory variable `smem` is declared globally. It is allocated dynamically at the launch time of each kernel, possibly with different sizes. The shared memory variable associated with each `shared` directive points to a particular offset of this common region. The scheme for constant memory variables is similar.

The optimization problem the compiler needs to solve is to determine the placement of individual variables' memory regions in `smem` that *minimizes* the size of `smem` (for each kernel region), while still respecting the constraint that the memory regions for any two variables with overlapping live ranges do not overlap. This is a standard *compile-time*

memory allocation problem and can be formulated as an interval coloring problem on the (weighted) interference graph [12], similar to the graph coloring approach in register allocation. We adopt a heuristic algorithm proposed by Clementson and Elphick [13].

## 5.7 Support for Real-World Applications

A real-world application typically contains multiple procedures (possibly in multiple files), and uses dynamically allocated array. Furthermore, directives that affect a kernel may appear in different procedures. In this section, we describe the *hiCUDA* compiler that supports such applications.

### 5.7.1 Support of Dynamically Allocated Arrays

*hiCUDA* provides a shape directive so that the programmer can specify the shape of dynamically allocated arrays in global and constant directives. The compiler uses the shape information to promote accesses of these arrays, in the form of `*(ptr + offset)`, to regular array accesses `(*ptr)[i1][i2]...[in]` (within the scope of the shape directive). This promotion process involves factoring a 1D access offset into multidimensional offsets, which must be verified to be within the corresponding array bounds using a linear programming solver [9]. This allows the kernel data access analysis and access redirection described above to be used without any modifications.

### 5.7.2 Interprocedural Support

Most of the above-mentioned phases must be extended to support the use of functions. More specifically, since a data directive “needed” by a kernel region can be placed in a different procedure than the region, the compiler must “propagate” this directive interprocedurally to the kernel region in reaching directives analysis. Since a kernel region can contain function calls, accesses made within these functions must be considered during kernel data access analysis, and then redirected to corresponding GPU memory variables as well. Further, `loop_partition` and `shared` directives may be placed inside these functions, necessitating a phase that interprocedurally propagates the context of a kernel region (e.g., the thread block and thread spaces, and enclosing loops that are partitioned) to these functions.

All interprocedural propagations added to the *hiCUDA* compiler are built upon a common framework that uses function cloning [9]. In this framework, annotations are attached to each function, each of which contains relevant information propagated to this function from a particular chain of function calls. As an example, in reaching directives analysis, the annotation carries information about the global or constant directive (if any) associated with each formal parameter of the function. The propagation starts with an “empty” annotation (i.e., no directive for any formal parameter) in the main function. When propagating across a function call, a new annotation is constructed for the callee. Each of its formal parameters is associated with a data directive, if the directive 1) is for the corresponding actual parameter, and 2) reaches the function call. Note that this directive could either appear locally in the caller function or in the caller’s annotation. The newly constructed annotation is added to the callee only if it is unique among the existing ones in the callee. The propagation terminates when no function

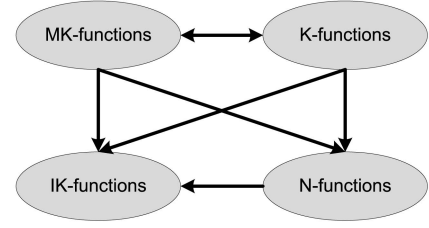


Fig. 10. Call flow directions among the four classes of functions.

gets any new annotation. Then, the compiler performs cloning on each function that has more than one annotation. Thus, at the end, each function (involved in the propagation) has exactly one annotation. From this point on, the compiler can perform reaching directive analysis in each procedure as before, taking into account the propagated directives in the annotation. We adopt this interprocedural framework over function inlining because the latter could lead to dramatic increase in kernel code size, which will have a great impact on the performance of the GPU’s instruction cache.

To facilitate the implementation of these interprocedural analyses, we add a preprocessing phase in the compiler that classifies each function in the input code into one of four classes:

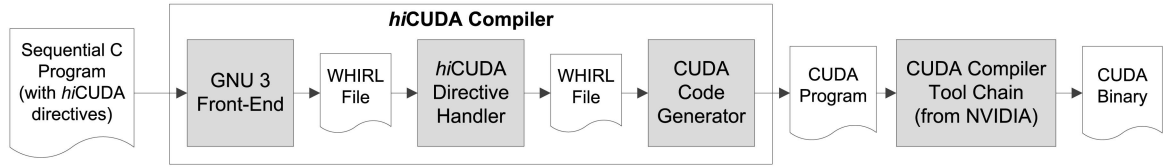
- **K-function:** those that directly contain kernel regions.
- **MK-function:** those that do not directly contain kernel regions but may lead to one; that is, there is at least one call path from this function to a K-function.
- **IK-function:** those that may be inside a kernel region; that is, there is at least one call path from within a kernel region to this function.
- **N-function:** none of the above.

Fig. 10 shows the call graph partitioned into the four classes and illustrates the call flow directions among them. With this classification scheme in place, checking kernel-related errors (e.g., nesting of kernel regions) becomes straightforward, and the compiler can perform the interprocedural propagations on specific classes of functions. For example, data directive propagation in reaching directives analysis is performed among K-functions and MK-functions; access redirection and kernel context propagation are performed among K-functions and IK-functions.

## 6 EXPERIMENTAL EVALUATION

In order to evaluate *hiCUDA*, we implemented a prototype compiler to translate an input C program with *hiCUDA* directives to an equivalent CUDA program. This allows the use of NVIDIA CUDA compiler tool chain [6] to generate binaries. Fig. 11 shows the entire compilation flow. The *hiCUDA* compiler is built around Open64 (version 4.1) [14]. It consists of three components:

1. A C front-end that supports *hiCUDA* directive syntax, which is extended from the GNU 3 front-end in Open64.
2. A compiler pass that lowers *hiCUDA* directives to CUDA code, based on the algorithms described in Section 5. It uses several existing modules in

Fig. 11. Compilation flow of a *hiCUDA* program.TABLE 1  
CUDA Benchmarks for Evaluating the *hiCUDA* Compiler

Application	Description (after [2], [15])
Black-Scholes Option Pricing (BSOP)	Black-Scholes model for European options.
Matrix Multiply (MM)	Computes the product of two matrices.
N-body Simulation (NBODY)	Approximates the evolution of a system of bodies that continuously interact with each other.
Coulombic Potential (CP)	Computes the coulombic potential at each grid point over on plane in a 3D grid.
Sum of Absolute Differences (SAD)	Sum of absolute differences kernel, used in MPEG video encoders.
Two Point Angular Correlation Function (TPACF)	TPACF is an equation used here as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body.
Rys Polynomial Equation Solver (RPES)	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.
Magnetic Resonance Imaging Q (MRI-Q)	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
Magnetic Resonance Imaging FHD (MRI-FHD)	Computation of an image-specific matrix FHD, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.

Open64, such as data flow analysis, array section analysis, and interprocedural analysis framework.

3. A CUDA code generator, extended from the C code generator in Open64.

We evaluate *hiCUDA* using this compiler. In particular, we use nine CUDA benchmarks to show that the compiler-generated code from a *hiCUDA* program performs as well as the handwritten CUDA code implementing the same algorithm. This will be detailed in the next section. Further, we present a case study of accelerating a real-world medical application—Monte Carlo simulation for Multi-Layered media, which will be discussed in Section 6.2.

## 6.1 Performance

To compare the performance of *hiCUDA* programs against handwritten CUDA versions, we use nine benchmarks listed in Table 1. For each benchmark, we start from a sequential version and insert *hiCUDA* directives to achieve the transformations that result in the corresponding CUDA version. The first three benchmarks (BSOP, MM, and NBODY) are obtained from the CUDA SDK [2]. For each of them, we wrote our own sequential version based on the reference CPU routines in the CUDA code. For each of the remaining benchmarks, we obtained two pairs of sequential versus CUDA versions from the *base*, *cuda\_base*, *cpu*, and *cuda* versions of the Parboil benchmark suite, respectively [15]. The *cuda\_base* and *cuda* versions are CUDA programs ported from the sequential *base* and *cpu* versions, respectively. Compared to the *base* version, the *cpu* version includes optimizations for achieving better performance in the *cuda* version (compared to *cuda\_base*).

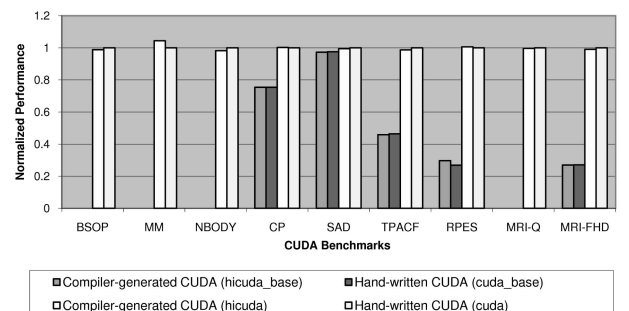
We conducted the experiments using a GeForce 8800GT card with CUDA v1.1 driver and compiler tool chain, on an Intel Core-2-Quad (Q6600) machine running Ubuntu 7.10.

Fig. 12 shows the performance of each benchmark (i.e., the inverse of wall clock execution time), normalized with

respect to the *cuda* version. The *hiCUDA* versions built upon the *base* and *cpu* versions are labeled *hicuda\_base* and *hicuda*, respectively. The figure shows that the performance of our compiler-generated CUDA code is comparable or slightly exceeds that of the corresponding handwritten CUDA versions within  $\pm 1.3$  percent (except noticeable performance improvement in matrix multiply).

It is important to note that, in order to achieve the algorithm implemented in the handwritten CUDA versions, we did make some modifications to the sequential code *before* inserting *hiCUDA* directives and to the CUDA code generated by the *hiCUDA* compiler. The modifications made to the sequential code are mostly unrelated to the CUDA-porting process, and include:

- Standard loop transformations, for example, strip-mining, permutation, fusion/fission, collapsing, and unrolling.
- Data padding and repacking, for optimized memory performance.
- General math operation replacement.

Fig. 12. Performance comparison between *hiCUDA* and CUDA programs.

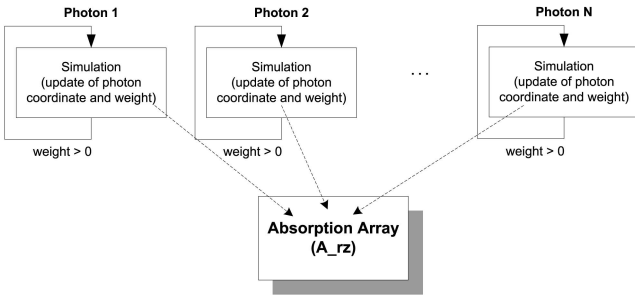


Fig. 13. High-level structure of the MCML program.

Modifications made to the generated CUDA code are mostly manual CUDA optimizations, and include:

- Programming idioms, e.g., reduction and histogram.
- CUDA-specific math operation acceleration.
- Advanced use of shared memory.

We believe that standard loop transformations and programming idioms could be easily automated and incorporated into the *hiCUDA* language.

## 6.2 A Case Study: MCML

We provided an earlier version of the prototype compiler to a medical research group at University of Toronto to accelerate a real-world application called Monte Carlo simulation for Multi-Layered media [16]. It is a gold standard code for using Monte Carlo methods to compute light-dose distribution for photodynamic therapy (PDT) treatment planning. The high-level execution flow of this program is represented in Fig. 13. The program is massively parallel as the millions of photons are simulated independently (except when making updates to the shared copy of the absorption array).

We assisted the research group to port MCML to CUDA and then to *hiCUDA*. To make this program fully parallelizable, two changes had to be made to the sequential code: 1) replacing the existing random number generator with a parallelizable one, and 2) privatizing the absorption array (*A\_rz*) so that each GPU thread has its own copy to work with. Also, double-precision floating-point operations were turned into single-precision because they are not supported by GeForce 8800. The actual process of converting to CUDA code included three steps:

1. Creating two kernels: one for photon simulation and the other for summing private copies of *A\_rz*.
2. Distributing the photon loop among the threads.
3. Setting up data in global and constant memories.

Note that the photon simulation code contains quite a few chains of function calls, and the absorption array and the data structure that describe the media are dynamically allocated.

To perform the porting tasks in *hiCUDA*, 17 directives were inserted to the sequential code (that contains the modification described above), including two kernels, three `loop_partitions`, two barriers, one `shape`, three `globals`, three `constants`, and three `shareds`). Both the CUDA and *hiCUDA* versions achieved a 18× speedup on a GeForce 8800GTX card over single-thread performance on a 3-GHz Intel Xeon 5160 processor. The feedback provided by

the research group is that *hiCUDA* is simpler to learn and use compared to CUDA. Also, it quickly became evident that using *hiCUDA* directives to partition kernel computation and set up host-GPU memory transfer helps novice programmers to avoid constructing incorrect expressions involving thread index variables and thoroughly reading through the CUDA runtime API.

## 7 RELATED WORK

There have been a large body of work in enhancing the software support for GPGPU programming. They can be divided into three categories.

The first category extends CUDA support to other programming languages, such as PyCUDA [17] for Python, jCUDA [18] for Java, and CUDA Fortran [19] (to be jointly developed by PGI and NVIDIA). This group of work is different from ours in that they still require the programmer to write kernel code in a separate function and use thread index variables to partition the computation. In contrast, we aim to simplify this process through compiler directives.

The second category of related work provides high-level abstraction of CUDA programming in terms of compiler directives, and therefore, is closely related to ours. Lee et al. [20] propose a compiler framework for translating an OpenMP [21] program to a CUDA program. The main contributions of this work include an interpretation of OpenMP semantics under the CUDA model and a set of transformations that optimize global memory accesses. PGI has recently released a directive-based Accelerator Programming Model [22] for CPU+Accelerator systems, and the latest PGI Fortran and C compiler supports this model on CUDA-enabled NVIDIA GPUs. Compared to *hiCUDA*, OpenMP is a standard API that many programmers are already familiar with and many existing applications are programmed in OpenMP. However, both the OpenMP and the Accelerator model are not specific to the CUDA architecture, and therefore, lack the support of important concepts like shared memory and thread block. Creating an abstraction that closely matches the CUDA model is exactly the reason we design a new and simpler set of directives. Further, it is unclear to us how both work provide interprocedural support for real applications.

The third category of work that is related to ours focuses on helping the programmer optimize a CUDA application. Ryoo et al. [5], [7] find that the optimization space of a CUDA program is discontinuous. Instead of sweeping through the optimization space in brute force, they develop two metrics to effectively prune this space. CUDA-lite [23], developed by Ueng et al., uses compiler techniques to automate some CUDA optimization patterns, with the help of programmer hints. Similarly, Luk et al. [24] examine optimizations for CPU/GPU systems. These works are orthogonal and complementary to ours, in that they assume that the programmer has already written the entire CUDA application while our work eases the transition from a sequential program to CUDA.

## 8 CONCLUSION AND FUTURE WORK

We have designed a high-level abstraction of CUDA, in terms of simple compiler directives. We believe that it can greatly

ease CUDA programming. Our experiments show that *hiCUDA* does not sacrifice performance for ease-of-use: the CUDA programs generated by our *hiCUDA* compiler perform as well as the handwritten versions. Further, the use of *hiCUDA* for porting a real-world application to CUDA suggests that it can potentially save much of the development time, but necessitate that the compiler be able to provide interprocedural support. This encourages us to share this system with the CUDA programming community, by releasing the *hiCUDA* compiler at <http://www.hicuda.org>.

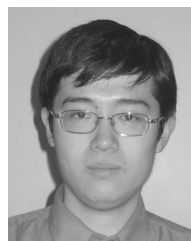
Currently, we have finished the core design of the *hiCUDA* language, which simplifies the most common tasks almost every CUDA programmer has to do. This work opens up many directions for ongoing research. First, we have observed that, for many applications, standard loop transformations and high-level idioms are required to be applied before and after inserting *hiCUDA* directives, respectively. Since they often involve nontrivial code changes, it is highly beneficial to automate these transformations by incorporating them into *hiCUDA*. Second, we would like to enhance the capability of the *hiCUDA* compiler so that it would guide the programmer to write a correct and optimized program. For example, the compiler can help programmers validate a partitioning scheme of kernel computation by doing data dependence analyses, and detect nonoptimized memory access patterns. Third, we would like to further simplify or even eliminate some *hiCUDA* directives so that the burden on the programmer is further reduced. For example, we could delegate the work of inserting *hiCUDA* data directives to the compiler, which can determine an optimal data placement strategy using various data analyses [25]. This direction would ultimately lead to a parallelizing compiler for GPUs, which requires no intervention from the programmer. Finally, since OpenCL supports a data parallel programming model that is very close to CUDA, we do not see any difficulty in porting the *hiCUDA* language and compiler support to OpenCL.

## ACKNOWLEDGMENTS

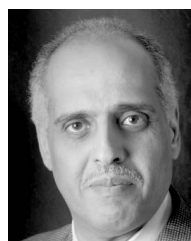
The authors are grateful to William Lo and Lothar Lilge for providing the Monte Carlo simulation application for their use of a prerelease version of the *hiCUDA* compiler to port the MCML application to a GPU. The feedback they provided was invaluable in improving both the syntax and semantics of *hiCUDA*. This work is supported by the NSERC research grants.

## REFERENCES

- [1] NVIDIA, "NVIDIA GeForce 8800 GPU Architecture Overview," [http://www.nvidia.com/object/IO\\_37100.html](http://www.nvidia.com/object/IO_37100.html), Nov. 2006.
- [2] NVIDIA, "NVIDIA CUDA Programming Guide v1.1," [http://developer.download.nvidia.com/compute/cuda/1.1/NVIDIA\\_CUDA\\_Programming\\_Guide\\_1.1.pdf](http://developer.download.nvidia.com/compute/cuda/1.1/NVIDIA_CUDA_Programming_Guide_1.1.pdf), Nov. 2007.
- [3] I. Buck et al., "Brook for GPUs: Stream Computing on Graphics Hardware," *Proc. ACM SIGGRAPH*, pp. 777-786, 2004.
- [4] "Open Computing Language (OpenCL)," <http://www.khronos.org/opensl/>, 2010.
- [5] S. Ryoo et al., "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," *Proc. Symp. Principles and Practice of Parallel Programming*, pp. 73-82, 2008.
- [6] NVIDIA, "The CUDA Compiler Driver NVCC v1.1," [http://www.nvidia.com/object/cuda\\_programming\\_tools.html](http://www.nvidia.com/object/cuda_programming_tools.html), 2007.
- [7] S. Ryoo et al., "Program Optimization Space Pruning for a Multithreaded GPU," *Proc. Int'l Symp. Code Generation and Optimization*, pp. 195-204, 2008.
- [8] ISO( )14882:2003, "Information Technology—Programming Languages—C++," ISO, 2003.
- [9] T.D. Han, "Directive-Based General-Purpose GPU Programming," master's thesis, Univ. of Toronto, Sept. 2009.
- [10] C. Liao et al., "Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing*, Oct. 2009.
- [11] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [12] J. Fabri, "Automatic Storage Optimization," *Proc. Symp. Compiler Construction*, pp. 83-91, 1979.
- [13] A. Clementson and C. Elphick, "Approximate Coloring Algorithms for Composite Graphs," *J. Operational Research Soc.*, vol. 34, no. 6, pp. 503-509, 1983.
- [14] "Open64 Research Compiler," <http://open64.sourceforge.net>, 2010.
- [15] IMPACT Research Group, "The Parboil Benchmark Suite," <http://www.crhc.uiuc.edu/IMPACT/parboil.php>, 2007.
- [16] L. Wang, S. Jacques, and L. Zheng, "MCML—Monte Carlo Modeling of Light Transport in Multi-Layered Tissues," *Computer Methods and Programs in Biomedicine*, vol. 47, no. 2, pp. 131-146, 1995.
- [17] A. Klockner, "Pycuda v0.94beta Documentation," <http://documen.tician.de/pycuda/>, 2010.
- [18] GASS, "JCUDA: Java for CUDA," <http://www.gass-ltd.co.il/en/products/jcuda/>, 2010.
- [19] The Portland Group, "CUDA Fortran Programming Guide and Reference v0.9," [http://www.pggroup.com/lit/whitepapers/pgi\\_spec\\_cuda\\_fortran\\_0.9.pdf](http://www.pggroup.com/lit/whitepapers/pgi_spec_cuda_fortran_0.9.pdf), June 2009.
- [20] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *Proc. Symp. Principles and Practice of Parallel Programming*, pp. 101-110, 2009.
- [21] OpenMP ARB, "OpenMP Specification v3.0," <http://openmp.org/wp/openmp-specifications/>, May 2008.
- [22] The Portland Group, "PGI Fortran and C Accelerator Programming Model," [http://www.pggroup.com/lit/whitepapers/pgi\\_accel\\_prog\\_model\\_1.0.pdf](http://www.pggroup.com/lit/whitepapers/pgi_accel_prog_model_1.0.pdf), June 2009.
- [23] S.-Z. Ueng et al., "CUDA-lite: Reducing GPU Programming Complexity," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing*, pp. 1-15, 2008.
- [24] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," *Proc. Int'l Symp. Microarchitecture*, pp. 45-55, 2009.
- [25] M.M. Baskaran et al., "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs," *Proc. Int'l Conf. Supercomputing*, pp. 225-234, 2008.



**Tianyi David Han** received the MSc degree in computer engineering from the University of Toronto in 2007. He is currently working toward the PhD degree in the Edward S. Rogers Sr. Department of Electrical and Computer Engineering at the University of Toronto, Ontario, Canada. His research interests are in the areas of parallel computing and general purpose GPU computing. He is a student member of the IEEE and the IEEE Computer Society.



**Tarek S. Abdelrahman** received the PhD degree in computer science and engineering from the University of Michigan at Ann Arbor in 1989. He is currently a professor of electrical and computer engineering and of computer science at the University of Toronto, Ontario, Canada. His research interests are in the areas of parallel systems, parallelizing and optimizing compilers, parallel programming models, and runtime support. He is a senior member of the IEEE, the IEEE Computer Society, and the ACM, and a member of the USENIX.