

*hi*CUDA: A High-level Directive-based Language for GPU Programming

Tianyi David Han

The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
han@eecg.toronto.edu

Tarek S. Abdelrahman

The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
tsa@eecg.toronto.edu

ABSTRACT

The Compute Unified Device Architecture (CUDA) has become a de facto standard for programming NVIDIA GPUs. However, 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 components of the GPU memory, and of manually optimizing the utilization of the GPU memory. Practical experience shows that the programmer needs to make significant code changes, which are often tedious and error-prone, before getting an optimized program. We have designed *hi*CUDA, 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. Nonetheless, it supports the same programming paradigm already familiar to CUDA programmers. We have prototyped a source-to-source compiler that translates a *hi*CUDA program to a CUDA program. Experiments using five standard CUDA benchmarks show that the simplicity and flexibility *hi*CUDA provides come at no expense to performance.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—*concurrent, distributed, and parallel languages, specialized application languages, very high-level languages*

General Terms

Languages, Design, Documentation, Experimentation, Measurement

Keywords

CUDA, GPGPU, Data parallel programming

1. INTRODUCTION

The Compute Unified Device Architecture (CUDA) has become a de facto standard for programming NVIDIA GPUs.

Although it is a simple extension to C, 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 performed by a compiler.

Therefore, we have defined a directive-based language called *hi*CUDA (for *high-level* CUDA) for programming NVIDIA GPUs. The language 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 *hi*CUDA directives makes it easier to experiment with different ways of identifying and extracting GPU computation, and of managing the GPU memory. We have implemented a prototype compiler that translates a *hi*CUDA program to an equivalent CUDA program. Our experiments with five standard CUDA benchmarks show that the simplicity and flexibility *hi*CUDA provides come at no expense to performance. For each benchmark, the execution time of the *hi*CUDA-compiler-generated code is within 2% of that of the hand-written CUDA version.

The remainder of this paper is organized as follows. Section 2 provides some background on CUDA programming. Section 3 introduces our *hi*CUDA directives using a simple example. Section 4 specifies the *hi*CUDA language in details. Section 5 describes our compiler implementation of *hi*CUDA and gives an experimental evaluation of the compiler-generated code. Section 6 reviews related work. Finally, Section 7 presents concluding remarks and directions for future work.

2. CUDA PROGRAMMING

CUDA provides a programming model that is ANSI C, extended with several keywords and constructs. The programmer writes a single source program that contains both the host (CPU) code and the device (GPU) code. These two parts will be automatically separated and compiled by the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPGPU '09 March 8, 2009, Washington, D.C. USA
Copyright 2009 ACM 978-1-60558-517-8 ...\$5.00.

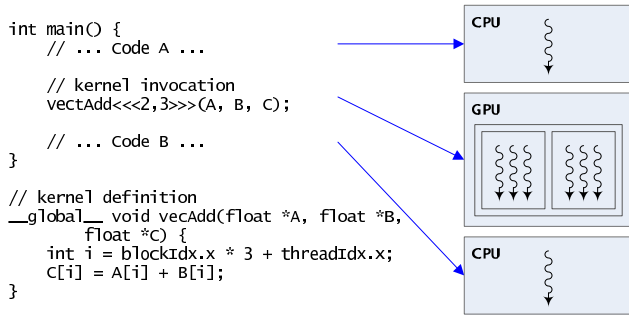


Figure 1: Kernel definition and invocation in CUDA.

CUDA compiler tool chain [8].

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. Figure 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 two-dimensional space (or one-dimensional if the size of the second dimension is 1). Each block contains multiple threads, organized in a three-dimensional space. In the example (Figure 1), the grid contains 2 blocks, each containing 3 threads, so the kernel is executed by 6 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* [7].

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 block has its private *shared memory*. All threads in the block have read and write access to this shared 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*. These three memories have the same access latency as the local memory.

Local variables in a kernel function are automatically allocated in registers (or local memory). Variables in other GPU memories must be created and managed explicitly, through the CUDA runtime API. The global, constant and texture memory are also accessible from the host. The data needed by a kernel must be transferred into these memories before it is launched. Note that these data are persistent across kernel launches. The shared memory is essentially a cache for the global memory, and it requires explicit management

in the kernel. In contrast, the constant and texture memory have caches that are managed by the hardware.

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 it, and partition the kernel 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. This includes redirecting variable accesses in the kernel to the corresponding copies allocated in the GPU memories.
4. Perform memory optimizations in the kernel, such as utilizing the shared memory and coalescing accesses to the global memory [7, 13].
5. Perform other optimizations in the kernel in order to achieve an optimal balance between single-thread performance and the level of parallelism [12].

Note that a CUDA program may contain multiple kernels, in which case the procedure above needs to be applied to each of them.

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 [12]. This not only increases development time, but also makes the program difficult to understand and to maintain. For example, it is non-intuitive 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 can easily go wrong, prolonging program development and debugging.

3. *hi*CUDA THROUGH AN EXAMPLE

We illustrate the use of *hi*CUDA directives with the popular matrix multiplication code, shown in Figure 2. The code computes the product 64×32 matrix *C* of two matrices *A* and *B* of dimensions 64×128 and 128×32 respectively. We map the massive parallelism, available in the triply nested loops (*i,j,k*) (lines 10-18), onto the GPU. The resulting *hi*CUDA program is shown in Figure 3.

The loop nest (*i,j,k*) will be executed on the GPU. Thus, it is surrounded by the `kernel` directive (lines 13 and 33 of Figure 3). 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 4×2 thread blocks, each of which contains 16×16 threads.

We exploit parallelism in the matrix multiply code by dividing the iterations of the *i* and *j* loops among the threads. The `loop_partition` directives (lines 15 and 17 of Figure 3) 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 4×2 thread blocks) respectively. Thus, each thread

```

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);

```

Figure 2: The original matrix multiply program.

block executes a $64/4 \times 32/2$ or a 16×16 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 16×16 threads) respectively. Thus, each thread executes a $16/16 \times 16/16$ tile or a single iteration in the 16×16 tile assigned to the thread block. This partitioning scheme is shown in Figure 4a.

The arrays A , B and C must be allocated in the global memory of the GPU device. Further, the values of A and B , initialized on the host, must be copied to their corresponding global memory 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 Figure 3). 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 must be improved by utilizing the shared memory on the GPU [7]. 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 Figure 4b. 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 Figure 3) 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 directives (i.e. a 1×32 tile of A and a 32×1 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 16×32 tile of A and

```

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 hicuda global alloc A[*][*] copyin
10 #pragma hicuda global alloc B[*][*] copyin
11 #pragma hicuda global alloc C[*][*]
12
13 #pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
14 // C = A * B
15 #pragma hicuda loop_partition over_tblock over_thread
16 for (i = 0; i < 64; ++i) {
17     #pragma hicuda 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 hicuda shared alloc A[i][kk:kk+31] copyin
22             #pragma hicuda shared alloc B[kk:kk+31][j] copyin
23             #pragma hicuda barrier
24             for (k = 0; k < 32; ++k) {
25                 sum += A[i][kk+k] * B[kk+k][j];
26             }
27             #pragma hicuda barrier
28             #pragma hicuda shared remove A B
29             C[i][j] = sum;
30         }
31     }
32 }
33 #pragma hicuda kernel_end
34
35 #pragma hicuda global copyout C[*][*]
36 #pragma hicuda global free A B C
37
38 printMatrix((float*)C, 64, 32);
39

```

Figure 3 includes several annotations with arrows pointing to specific code sections:

- Data allocation and initialization (host to GPU):** Points to lines 9-11, which use `#pragma hicuda global alloc` with `copyin` for arrays A and B.
- Kernel:** Points to line 13, which starts the `#pragma hicuda kernel` block.
- Strip-mining:** Points to the inner loop `for (kk = 0; kk < 128; kk += 32) {` on line 20.
- Preloading data to the shared memory:** Points to lines 21-22, which use `#pragma hicuda shared alloc` to load data into shared memory.
- Data write-back (GPU to host) and deallocation:** Points to lines 35-37, which use `#pragma hicuda global copyout` for C and `#pragma hicuda global free` for A and B.

Figure 3: The *hiCUDA* matrix multiply program.

another for holding a 32×16 tile of B . The details of the `shared` directive appear in Section 4.2.

For comparison purpose, the hand-written CUDA version for matrix multiply is shown in Figure 5. It is clear that the *hiCUDA* code is simpler to write, to understand and to maintain. The programmer does not need to separate the kernel code from the host code nor to use explicit thread indices to partition computations. Nonetheless, *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 write 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 de-allocate 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. 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.

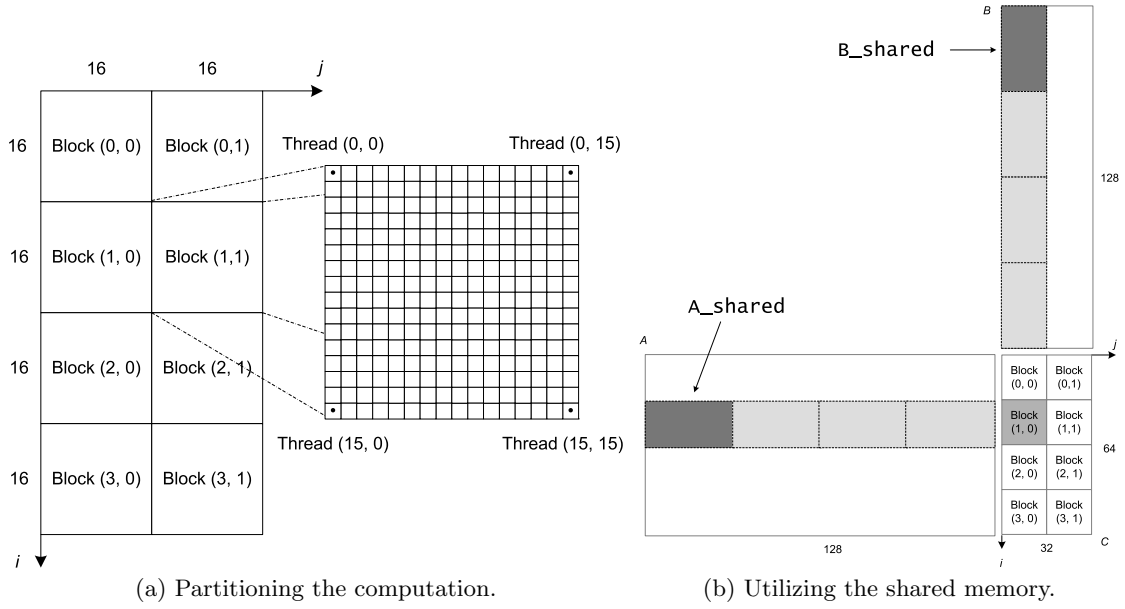


Figure 4: The scheme of accelerating the matrix multiply code on GPU.

```
// Randomly init A and B.
randomInitArr((float*)A, 64*128);
randomInitArr((float*)B, 128*32);

size = 64 * 128 * sizeof(float);
cudaMalloc((void**)&d_A, size);
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
size = 128 * 32 * sizeof(float);
cudaMalloc((void**)&d_B, size);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

size = 64 * 32 * sizeof(float);
cudaMalloc((void**)&d_C, size);

dim3 dimBlock(16, 16);
dim3 dimGrid(32/dimBlock.x, 64/dimBlock.y);

matrixMul<<<dimGrid, dimBlock>>>>(
    d_A, d_B, d_C, 128, 32);

cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Data allocation and initialization (host to GPU)

Kernel launch

Data write-back (GPU to host) and deallocation

```
***** matrixMul kernel *****
__global__ void matrixMul(float *A, float *B, float *C,
    int wA, int wB) // width of A and B
{
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;

    int aBegin = wA * 16 * by + wA * ty + tx;
    int aEnd = aBegin + wA;
    int aStep = 32;
    int bBegin = 16 * bx + wB * ty + tx;
    int bStep = 32 * wB;

    __shared__ float As[16][32];
    __shared__ float Bs[32][16];

    float Csub = 0;

    for (int a = aBegin, b = bBegin; a < aEnd;
        a += aStep, b += bStep) {
        As[ty][tx] = A[a]; As[ty][tx+16] = A[a + 16];
        Bs[ty][tx] = B[b]; Bs[ty+16][tx] = B[b + 16*wB];

        __syncthreads();

        for (int k = 0; k < 32; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        __syncthreads();
    }

    C[wB*16*by + 16*bx + wB*ty + tx] = Csub;
}
```

Iterator loop of k (after strip-mining)

Data transfer to the shared memory

Inner loop of k (after strip-mining)

Figure 5: The CUDA matrix multiply program.

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. *thread-block-clause* and *thread-clause* specify a virtual grid of GPU threads. They have the following format:

```
tbblock( 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 space¹.

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. The *hiCUDA* compiler automatically maps the virtual spaces to the 2D thread-block space and 3D thread space supported by CUDA. By default, the host thread waits for the kernel to finish execution. If the `nowait` clause is present, the host thread proceeds asynchronously after launching the kernel.

Local variables in *sequential-code* are automatically allocated in registers. All other variables needed or produced by *sequential-code* must be allocated in the global, constant or texture memory before the kernel invocation, which can be done through the data directives discussed in the next section. The only exception is scalar variables that are read-only in *sequential-code*; if they are not explicitly allocated in the GPU memory, they are passed into the kernel function as parameters.

With the `kernel` directive, the kernel in its entirety will be 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
```

¹In 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.

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 a sub-clause, and the distribution strategy is always CYCLIC. The rationale behind this restriction is explained later in the section.

Figure 6 shows an example of how the `loop_partition` directive works. The first thing to note is that the directive can be arbitrarily nested within each other. If the directive contains 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 `loop_partition` directives that also contain the `over_tblock` clause. In the example, the directives for loop *i*, *j* and *k* have *LB* defined, which are 1, 2 and 2 respectively. Similarly, if a `loop_partition` directive contains the `over_thread` clause, its *thread nesting level* (or *LT*) is defined to be the nesting level (starting from 1) with respect to enclosing `loop_partition` directives that also contain the `over_thread` clause.

If a `loop_partition` directive only has the `over_tblock` clause, 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, where B_i is defined previously. Each group has a distinct index in the LB^{th} dimension of the virtual thread-block space. The subset of loop iterations assigned to each group is determined by the distribution strategy specified in the clause. These iterations are executed by every thread in every thread block in the group. In the example, the `loop_partition` directive for loop *i* distributes the iterations over thread blocks. Since its *LB* is 1, the iteration space is distributed over the first dimension of the thread-block space, whose size is 2. Based on the blocking distribution strategy, the thread blocks with the first-dimension index 0 execute the first half of the iteration space, and those with the first-dimension index 1 execute the second half. The `loop_partition` directive for loop *k* works similarly, except that the iterations are distributed over the second dimension of the thread-block space and the distribution strategy is cyclic.

If the directive only has the `over_thread` clause, each thread block executes all iterations of *for loop*. Within the thread block, the iterations are distributed over the LT^{th} dimension of the virtual thread space, specified in the enclosing `kernel` directive.

If both the `over_tblock` and `over_thread` clauses are specified, the set of loop iterations assigned to each thread block is further distributed among its threads. In the example, the directive for loop *j* specifies that each thread block is responsible for executing 12/3 or 4 iterations, which are distributed over 2 threads. Thus, each thread executes 2 iterations of loop *j*. If *distr-type* in the `over_tblock` clause is CYCLIC, the actual distribution strategy is blocking-cyclic with block size T_{LT} , where T_i and LT are defined previously. Within a cycle, the T_{LT} iterations assigned to each thread block is distributed so that each thread gets one iteration.

It is worth noting that the `loop_partition` directive supports non-perfect distribution of iterations over the threads. The *hiCUDA* compiler automatically generates “guard” code to ensure the exact number of iterations being executed.

In designing the `loop_partition` directive, we restrict the

```

#pragma hicuda kernel demo block(2,3) thread(2)

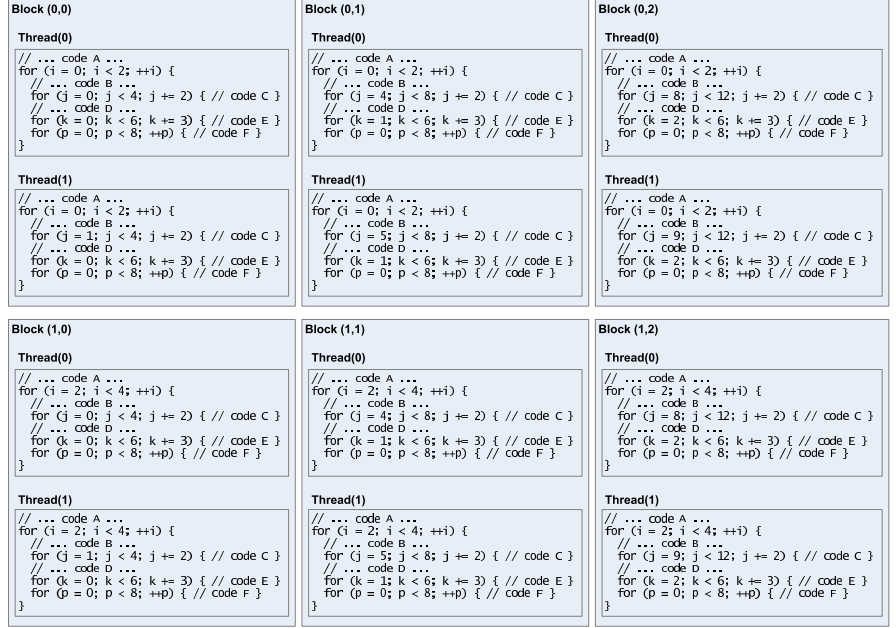
// ... code A ...

#pragma hicuda loop_partition over_tblock(BLOCK)
for (i = 0; i < 4; ++i) {
    // ... code B ...
    #pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 12; ++j) { // ... code C ... }
    // ... code D ...
    #pragma hicuda loop_partition over_tblock(CYCLIC)
    for (k = 0; k < 6; ++k) { // ... code E ... }
    for (p = 0; p < 8; ++p) { // ... code F ... }
}

#pragma hicuda kernel_end

```

(a) A kernel in a *hiCUDA* program.



(b) Code executed by each GPU thread.

Figure 6: Example use of the `loop_partition` directives.

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 Figure 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* can not be partitioned, i.e. it can not contain any `loop_partition` directives. If loop `p` in Figure 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, i.e. for initialization or “summary” code. They usually have to be executed once in *each* thread block, because the thread blocks are independent.

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 three main directives in its data model: `global`, `constant` and `shared`. Each directive manages the lifecycle of variables in the corresponding GPU memory. Currently, *hiCUDA* does not support the use of texture memory.

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

To support the management of dynamic arrays, *hiCUDA* also provides an auxiliary `shape` directive, allowing the user to specify the dimension sizes of these arrays. It has the following syntax:

```

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

```

where *ptr-var-sym* 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 has the following three forms:

```

#pragma hicuda global \
    alloc variable \
    [ { copyin [variable] } | clear ]

```

```
#pragma hicuda global \
    copyout variable \
    [ to 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 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 *variable*'s 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* (or the *variable* in the **to** clause if present). Note that the two *variable*'s do not have to refer to the same *var-sym* but must have the same type and shape (if they are arrays). 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 in its first form. 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.

The implicit global memory variable created by a **global** directive (the first form) is considered visible to the kernels between this directive and the corresponding **global** directive in its third form (i.e. when the global memory variable is deallocated). Within each kernel, accesses to host variables are redirected to their corresponding visible global memory variables, if present.

```

1  float A[1026][1026];
2  float B[1026][1026];
3
4  // short form of section specification
5  // for the entire array
6  #pragma hicuda global alloc A[*][*] copyin
7  #pragma hicuda global alloc B[1:1024][1:1024]
8
9  for (iter = 0; iter < 10; ++iter)
10 {
11     for (i = 1; i < 1025; ++i)
12         for (j = 1; j < 1025; ++j)
13             B[i][j] = 0.25 * (
14                 A[i-1][j] + A[i+1][j] +
15                 A[i][j-1] + A[i][j+1]
16             );
17
18     for (i = 1; i < 1025; ++i)
19         for (j = 1; j < 1025; ++j)
20             A[i][j] = B[i][j];
21 }
22
23 #pragma hicuda global copyout A[1:1024][1:1024]

```

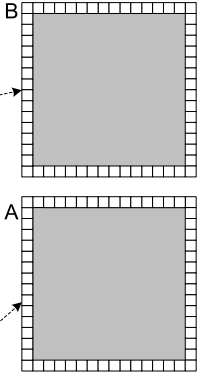


Figure 7: Array section specification in the **global** directives in the *jacobi* program.

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. Figure 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 (line 7 and 23 of Figure 7).

The **constant** directive is similar to the **global** directive. It has the following two forms:

```
#pragma hicuda constant \
    copyin variable

#pragma hicuda constant \
    remove {var-sym}+

```

The first form specifies that a copy of *variable* is to be statically allocated in the constant memory and initialized with the content of *variable* in the host memory. The directive does not need a **copyout** clause because the constant memory holds read-only data. Its **remove** clause is the same as the **free** clause in the **global** directive.

The **shared** directive looks similar to the **global** directive, with the following three forms:

```
#pragma hicuda shared \
    alloc variable \
    [ copyin[(nobndcheck)] [variable] ]

#pragma hicuda shared \
    copyout[(nobndcheck)] variable \
    [ to variable ]

#pragma hicuda shared \
    remove {var-sym}+

```

The semantics of the **shared** directive is different from that of the **global** directive in two aspects. First, the latter allocates data in the global memory and deals with data transfer between it and the host memory, while the former allocates data in the shared memory and deals with data transfer between it and the global memory. In addition to supporting the use of the shared memory as a cache for the global memory, the **shared** directive also allows variables local to the kernel region to be allocated in the shared memory. Since these variables do not have corresponding global memory variables, the **copyin** and **copyout** clause cannot be used. In this case, the only action associated with the directive is to redirect all accesses to *variable* (in the code bounded by this directive and a matching **shared** directive in its third form) to the corresponding shared memory variable.

The second difference between the **shared** and the **global** directive is 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 *variable*'s for *all concurrently executed iterations*. Consider the example shown in Figure 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. As shown in Figure 8b, this analysis determines the size of the shared memory variable to be created (i.e. a five-element array), and the code that loads data from the global memory to this variable. Since the shared memory variable is accessible by all threads in a thread block, the data-loading process is done *cooperatively* by these threads. In this case, each thread just needs to load at most two elements of `A`. Similarly, in the matrix multiply example (Figure 3), the 16×16 threads in a thread block concurrently execute 16×16 iterations of loop nest `(i,j)`. The *hiCUDA* compiler merges `A[i][kk:kk+31]` and `B[kk:kk+31][j]` with respect to a 16-value range of `i` and a 16-value range of `j`, and obtains a 16×32 tile of `A` and a 32×16 tile of `B`. Note that the starting indices of the 16-value ranges depend on the ID of the thread block, and the compiler uses them to generate code that loads appropriate sections of `A` and `B` for each thread block.

```

1 // Cyclically distribute it over 3 threads.
2 #pragma loop_partition over_thread
3 for (i = 0; i < 6; ++i) {
4     // A is a 6-element integer array.
5     #pragma shared alloc A[i-1:i+1] copyin
6     // ... Computation ...
7 }

```

(a) A kernel loop containing a `shared` directive.

```

1 __shared__ As[5];
2
3 for (ii = 0; ii < 6; ii += 3) {
4     // Load A[ii-1:ii+3] into As cooperatively.
5
6     // ... Computation for iteration
7     //   i = ii + threadIdx.x ...
8 }

```

(b) Code executed by each GPU thread.

Figure 8: Semantics of the `shared` directive.

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 Figure 8a, the `shared` directive attempts to load `A[-1]` (when `i = 0`) and `A[6]` (when `i = 5`). By de-

fault, the *hiCUDA* compiler automatically generates code that guards against invalid accesses if necessary. Since the decision made by the compiler could 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.

5. EXPERIMENTAL EVALUATION

We implemented a prototype compiler to translate input C programs with *hiCUDA* directives to equivalent CUDA programs. This allows the use of the existing CUDA compiler tool chain from NVIDIA to generate binaries [8]. Figure 9 shows the compilation flow. The *hiCUDA* compiler is built around Open64 (version 4.1) [9]. It consists of three components:

1. The GNU 3 front-end, which is extended from the one in Open64.
2. A compiler pass that lowers *hiCUDA* directives, which uses several existing modules in Open64, such as data flow analysis, interprocedural analysis and array section analysis.
3. The CUDA code generator, which is extended from the C code generator in Open64.

We compare the performance of our *hiCUDA* code against hand-written versions of standard CUDA benchmarks, listed in Table 1, on a Geforce 8800GT card. For each benchmark, we start from a sequential version and insert *hiCUDA* directives to achieve the transformations that result in the corresponding CUDA version. For the matrix multiply benchmark, we wrote our own sequential version (Figure 2), and obtained the CUDA code from NVIDIA CUDA SDK [7]. For each of the remaining benchmarks, we obtained the sequential version and two CUDA versions from the *base*, *cuda_base* and *cuda* versions of the Parboil benchmark suite respectively [5]. The *cuda* version is more optimized than the *cuda_base* version, and we use *hiCUDA* directives to achieve transformations done in the *cuda* version.

Figure 10 shows the performance of each benchmark (i.e. the inverse of wall-clock execution time), normalized with respect to the *cuda* version. The figure shows that the performance of our compiler-generated CUDA code is comparable or slightly exceeds that of the hand-written *cuda* versions. The noticeable improvement in the matrix multiply benchmark is purely caused by different ways of writing the same kernel function, and therefore depend on the optimizations performed in the existing CUDA compiler.

The reader should note that, although we are able to achieve all CUDA optimizations in *hiCUDA* for these benchmarks, we are sometimes required to make structural changes to the sequential code before applying *hiCUDA* directives. Such changes include array privatization, loop interchange and unrolling, and the use of fast math libraries. We believe that these standard transformations could be easily automated and incorporated into our language in the future.

Apart from the kernel benchmarks listed above, *hiCUDA* is also used by a medical research group at University of Toronto to accelerate a real-world application: Monte Carlo simulation for multi-media tissue [15]. A manually written CUDA version was developed in 3 months, achieving a 27X

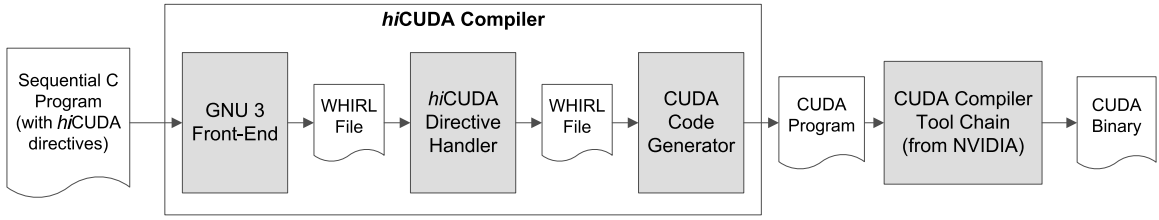


Figure 9: Compilation flow of a *hiCUDA* program.

Table 1: CUDA benchmarks for evaluating the *hiCUDA* compiler.

Application		Description (after [7, 5])
MM	Matrix Multiply	Computes the product of two matrices.
CP	Coulombic Potential	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. Adapted from 'cionize' benchmark in VMD.
SAD	Sum of Absolute Differences	Sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
TPACF	Two Point Angular Correlation Function	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.
RPES	Rys Polynomial Equation Solver	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.

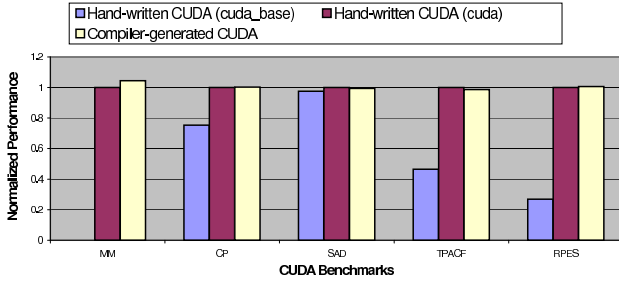


Figure 10: Performance comparison between *hiCUDA* and CUDA programs

speedup on a Geforce 8800GTX card over single-thread performance on Intel Xeon (3.0GHz). The same code transformations and optimizations were then applied to the original sequential version using *hiCUDA* directives, achieving the same speedup within 4 weeks.

6. RELATED WORK

There have been several streaming languages for General-Purpose GPU (GPGPU) programming, including Sh [6] and Brook [2]. Sh is a C++ library that allows programmers to embed data parallel kernel code in regular C++ code, and execute them on GPUs. It has been commercialized by RapidMind [11]. Brook is a full-fledged streaming language, that introduces stream data type, kernel definition and invocation, and reduction support. It has been implemented to target modern GPUs by the Stanford University Graphics group, resulting in BrookGPU [3]. This compiler and runtime system can use AMD's Close-to-Metal (CTM) [4] as its computational back-end. Recently, AMD decided to support OpenCL [10], a new language for programming heterogeneous data and task parallel computing across GPUs

and CPUs. This C99-based language was created by Apple in cooperation with others.

To our best knowledge, *hiCUDA* is the first directive-based language for GPGPU programming. The syntax of *hiCUDA* directives bears some similarity to that of OpenMP directives [1]. However, the two languages significantly differ in three aspects: 1) the threading hierarchy supported by *hiCUDA* is more complicated than the flat threading model OpenMP supports, 2) the GPU memory hierarchy supported by *hiCUDA* is more complicated than the shared memory model OpenMP supports, and 3) OpenMP supports a more general execution model than the SIMD model supported by *hiCUDA*, which means many OpenMP directives are unnecessary in *hiCUDA*.

CUDA-lite [14], developed by the Impact Research Group at UIUC, is a memory optimization tool for CUDA programs. It is similar to *hiCUDA* in that it also leverages the use of annotations. Currently, this tool takes a CUDA program that uses only the global memory, and optimizes its memory performance via access coalescing. Our work on *hiCUDA* is orthogonal to CUDA-lite. While CUDA-lite aims to automate memory optimization, it still requires the programmer to write a full-fledged CUDA program. In contrast, *hiCUDA* aims to simplify the means by which programmers express their CUDA code.

7. CONCLUDING REMARKS

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 hand-written versions. Further, the use of *hiCUDA* for a real-world application suggests that it can significantly cut development time. This encourages us to share this system with the CUDA programming commu-

nity. We hope that the feedback we receive will allow us to more comprehensively assess the ease of use of *hiCUDA* and to accordingly improve both the abstraction *hiCUDA* provides as well as the prototype compiler implementation.

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 are required to be applied before inserting *hiCUDA* directives. Since they often involve non-trivial 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 non-optimized memory access patterns. Last, 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. This direction would ultimately lead to a parallelizing compiler for GPUs, which requires no intervention from the programmer.

8. ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their constructive and helpful comments. The authors are also grateful to William Lo and Lothar Lilge for providing the Monte Carlo simulation application for their use of a pre-release version of the *hiCUDA* compiler to port the application to a GPU. The feedback they provided was invaluable in improving both the syntax and semantics of *hiCUDA*. Finally, this work is supported by NSERC research grants.

9. REFERENCES

- [1] OpenMP Architecture Review Board. OpenMP specification v3.0. <http://openmp.org/wp/openmp-specifications/>, May 2008.
- [2] I. Buck. Brook specification v0.2. <http://merrimac.stanford.edu/brook/>, October 2003.
- [3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proc. of SIGGRAPH 2004*, pages 777–786, New York, NY, USA, 2004.
- [4] AMD Stream SDK. <http://ati.amd.com/technology/streamcomputing/>.
- [5] IMPACT Research Group. The Parboil benchmark suite. <http://www.crhc.uiuc.edu/IMPACT/parboil.php>, 2007.
- [6] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proc. of ACM SIGGRAPH/EUROGRAPHICS*, pages 57–68, 2002.
- [7] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide v1.1. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf, November 2007.
- [8] NVIDIA. The CUDA Compiler Driver NVCC v1.1. http://www.nvidia.com/object/cuda_programming_tools.html, November 2007.
- [9] Open64 research compiler. <http://open64.sourceforge.net>.
- [10] Open Computing Language (OpenCL). <http://www.khronos.org/opencl/>.
- [11] RapidMind Inc. <http://www.rapidmind.net>.
- [12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S-Z Ueng, J. A. Stratton, and W-M W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, pages 195–204, 2008.
- [13] S. Ryoo, S. S. Rodrigues, C. I. and Baghsorkhi, S. S. Stone, D. B. Kirk, and W-M W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of Symp. on Principles and Practice of Parallel Programming*, pages 73–82, 2008.
- [14] S-Z Ueng, S. Baghsorkhi, M. Lathara, and W-M Hwu. CUDA-lite: Reducing GPU programming complexity. In *LCPC 2008: LNCS 5335*, pages 1–15, 2008.
- [15] L. Wang, S. L. Jacques, and L. Zheng. MCML—Monte Carlo modeling of light transport in multi-layered tissues. *Computer Methods and Programs in Biomedicine*, 47(2):131–146, 1995.