Directive-based General-Purpose GPU Programming

by

Tian Yi David Han

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2009 by Tian Yi David Han
Abstract

Directive-based General-Purpose GPU Programming

Tian Yi David Han

Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

Graphics Processing Units (GPUs) have become a competitive accelerator for non-graphics applications, 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. 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. We have also prototyped a compiler that translates a hiCUDA program to a CUDA program and can handle real-world applications. Experiments using seven standard CUDA benchmarks show that the simplicity hiCUDA provides comes at no expense to performance.
Acknowledgements

First, I would like to thank my supervisor, Tarek Abdelrahman, for his insightful guidance throughout the research. His vision and experience helped me stay on the right track and focus on the most important problems at each stage. I also appreciate his patient help on the presentation and documentation of this work.

I would also like to thank William Lo for his feedback on this work at an early stage, which helped me shape the direction for improvements. Thanks also go to my fellow graduate students: Reza Azimi, Utku Aydonat, Ivan Matosevic, Davor Capalija and Cedomir Segulja, for their constructive comments on the presentation of this work.

Special thanks go to my parents for their spiritual support and encouragement, without which I would not have made this far. Finally, I would like to acknowledge the financial support provided by the National Sciences and Engineering Research Council of Canada.
4 The hiCUDA Compiler

4.1 Overview .................................................. 28

4.2 Kernel Region Identification ................................ 32

4.3 GPU Data Management ..................................... 35
  4.3.1 Allocation and Deallocation ......................... 35
  4.3.2 Data Transfer ........................................ 36

4.4 Reaching Directives Analysis .............................. 40

4.5 Data Access Analysis ...................................... 44

4.6 Kernel Access Redirection ................................ 46

4.7 Kernel Loop Partitioning .................................. 52
  4.7.1 Kernel Context Propagation ......................... 52
  4.7.2 Handling loop partition Directives .................. 53

4.8 Utilizing the Shared Memory ............................. 58
  4.8.1 Allocation and Data Transfer ....................... 59
  4.8.2 Kernel Loop Context Propagation ................. 66

4.9 Kernel Outlining .......................................... 68
  4.9.1 Mapping the Virtual Thread Block and Thread Spaces 70

4.10 Supporting Phases ......................................... 72
  4.10.1 Support of Dynamic Arrays ....................... 73
  4.10.2 Optimization of GPU Memory Allocation .......... 75

4.11 Limitations .................................................. 79

5 Experimental Evaluation .................................... 81

5.1 Performance ................................................. 82

5.2 A Case Study: MCML ...................................... 87

6 Related Work .................................................. 90

7 Conclusion and Future Work ............................... 93
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Kernel access redirection</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>CUDA benchmarks for evaluating the hiCUDA compiler</td>
<td>83</td>
</tr>
<tr>
<td>A.1</td>
<td>Phase ordering in <em>ipa_link</em></td>
<td>97</td>
</tr>
<tr>
<td>B.1</td>
<td>Regression test suite of the hiCUDA compiler</td>
<td>98</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Kernel definition and invocation in CUDA ........................................... 6
2.2 Architecture of a CUDA-enabled GPU ............................................. 7
2.3 Matrix multiply in CUDA ............................................................ 10
3.1 Matrix multiplication example ......................................................... 14
3.2 The scheme of accelerating the matrix multiply code on GPU .................. 15
3.3 Example use of the loop partition directives ....................................... 20
3.4 Array section specification in the global directives in the jacobi program .. 25
3.5 Semantics of the shared directive ................................................... 27
4.1 Cases of invalid kernel regions in an inter-procedural context ................. 33
4.2 Call flow directions among the four categories of procedures ................. 34
4.3 Translation of array section transfer ............................................... 38
4.4 Translation of global and constant directives ..................................... 39
4.5 Motivation of inter-procedural reaching directives analysis .................... 40
4.6 Core algorithm of reaching directive propagation .................................. 42
4.7 Examples of reaching directives analysis .......................................... 43
4.8 Example of inter-procedural data access analysis .................................. 45
4.9 Example of kernel access redirection ................................................ 48
4.10 CUDA code generated from the example program (Figure 4.9) ............... 50
4.11 Example of kernel loop partitioning ................................................. 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>CUDA code templates for a <code>loop.partition</code> directive.</td>
<td>56</td>
</tr>
<tr>
<td>4.13</td>
<td>CUDA code templates for calculating a loop’s trip-count.</td>
<td>57</td>
</tr>
<tr>
<td>4.14</td>
<td>An example of <code>shared</code> directive handling</td>
<td>61</td>
</tr>
<tr>
<td>4.15</td>
<td>Code template for data transfer between the shared and the global memory</td>
<td>64</td>
</tr>
<tr>
<td>4.16</td>
<td>Examples of worst-case partitioning of a contiguous section into segments</td>
<td>65</td>
</tr>
<tr>
<td>4.17</td>
<td>An example of kernel loop context propagation</td>
<td>67</td>
</tr>
<tr>
<td>4.18</td>
<td>Example of kernel outlining</td>
<td>69</td>
</tr>
<tr>
<td>4.19</td>
<td>Example of shape propagation and pointer promotion</td>
<td>74</td>
</tr>
<tr>
<td>4.20</td>
<td>An example of efficient GPU memory allocation</td>
<td>77</td>
</tr>
<tr>
<td>4.21</td>
<td>Algorithm for finding the optimal allocation scheme</td>
<td>78</td>
</tr>
<tr>
<td>4.22</td>
<td>Limitations of the current compiler</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Compilation flow of a <code>hi</code>CUDA program</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance comparison between <code>hi</code>CUDA and CUDA programs.</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>High-level structure of the MCML program</td>
<td>87</td>
</tr>
<tr>
<td>A.1</td>
<td>Major components in the <code>hi</code>CUDA compiler</td>
<td>96</td>
</tr>
</tbody>
</table>
Chapter 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 GPGPU (for General-Purpose computing on the GPU), resulted because of a combination of several factors. First, the computing capability of a GPU is an order of magnitude greater than that of a CPU, and more importantly, is increasing at a faster rate [9]. This is because the GPU architecture is optimized for data-parallel (or throughput) computations, and much of the transistors dedicated to extracting instruction-level parallelism in the CPU architecture are instead used for arithmetic units. Second, these powerful GPUs are widely available at a much lower cost than traditional supercomputers. The most important factor that brings GPUs to the field of high performance computing is the great improvements in their programmability. The traditional fixed-function graphics pipeline has evolved into a much more flexible and unified many-core architecture, and non-graphics data-parallel languages have been introduced to program these cores. The Compute Unified Device Architecture (CUDA) is such a programming language specifically designed for NVIDIA GPUs, and is the focus of this thesis.

As a simple extension to C, CUDA has quickly become popular and attracted more
and more non-graphics programmers to port existing applications to CUDA. However, experience shows that this porting process remains challenging. 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 [34]. Nonetheless, many of the tasks involved are mechanical, and we believe can be automated by a compiler. This work is an attempt to achieve such automation.

1.1 Thesis Overview

We have defined a directive-based language called hiCUDA (for high-level CUDA) 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. Furthermore, we extended this prototype to provide full inter-procedural support needed for real applications. Our experiments with seven standard 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% of that of the hand-written CUDA
version. Thus, the contributions of our work are:

1. The definition of the hiCUDA language.

2. The design and implementation of the hiCUDA compiler, with inter-procedural support for real-world applications.

3. Experimental evaluation on the performance of hiCUDA programs using seven benchmarks and a case study.

4. The release of the hiCUDA compiler prototype in the public domain.

1.2 Thesis Organization

The remainder of this document is organized as follows. Chapter 2 provides background on the CUDA architecture and programming model as well as other GPGPU programming languages. Chapter 3 introduces the hiCUDA directives using a simple example and then specifies the hiCUDA language in details. Chapter 4 describes the design and implementation of the hiCUDA compiler. Chapter 5 gives our experimental evaluation of hiCUDA based on the prototype compiler. Chapter 6 reviews related work, and finally, Chapter 7 presents concluding remarks and directions for future work.
Chapter 2

Background

GPU programmability is the main driving force behind the growth of the GPGPU community. Early GPUs had fixed-function pipelines and therefore were only used for rendering images. About ten years ago, this rigid pipeline became transformed into a more flexible one, in which two main stages (vertex and fragment processing) were made programmable. Although the original intention of graphics vendors was to increase the visual realism of rendered images (by supporting user-defined operations on the primitives), this movement opened up the door for GPGPU. Since then, each new generation of GPUs added more generality to the two programmable stages, e.g. by increasing the limits on the number of inputs and outputs and providing limited support of flow control. Moreover, the advent of high-level shading languages like Cg [21], HLSL [23] and GLSL [17] allowed the programmer to write GPU programs in a C-like language, leaving the generation of assembly-level vertex and fragment programs to a compiler. Despite these improvements, GPGPU programming remained a big challenge, mainly because shading languages are still graphics-oriented and the programmer was forced to craft a mapping of the application algorithm to the graphics domain (i.e. in terms of geometric primitives, fragments and textures). Even worse, specialized vertex and fragment processors still lacked support of general memory access patterns, making the search for
these mappings more difficult. A detailed review of GPGPU work done in this early era can be found in [31]. In response to these difficulties, GPUs have been through another evolution in the past few years. Specialized vertex and fragment processors have been unified to become generic homogeneous cores with the ability to random-address device memory through a wider bandwidth. This makes the GPU a generic data-parallel co-processor, and therefore allows more general programming models to be developed on top of it. A representative example is NVIDIA GPUs (starting from GeForce 8 series) and its CUDA programming model, which this work is based on. The rest of the chapter briefly introduces the CUDA programming model and the architecture of CUDA-enabled NVIDIA GPUs, and then describes GPGPU programming languages developed by other parties.

2.1 CUDA Programming Model

The Compute Unified Device Architecture (CUDA) provides a programming model that is ANSI C, extended with several keywords and constructs [26]. The programmer writes a 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 [28].

CUDA allows the programmer to write device code in C functions called kernels [26]. 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 2.1 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 thread block contains
multiple threads, organized in a three-dimensional space. In the example (Figure 2.1), the grid contains 2 thread blocks, each containing 3 threads, so the kernel is executed by 6 threads in total. Each GPU thread is given a unique \textit{thread ID} that is accessible within the kernel, through the built-in variables \texttt{blockIdx} and \texttt{threadIdx}. They are vectors that specify an index into the thread block space (that forms the grid) and the thread space (that forms a thread block) respectively. In the example, each thread uses its ID to select a distinct vector element for addition. Note that the threads within a thread block can barrier-synchronize by invoking the \_\_\texttt{syncthreads} primitive. However, no synchronization is supported across different thread blocks.

![Figure 2.1: Kernel definition and invocation in CUDA.](image)

During kernel execution, the GPU threads have access to multiple GPU memories (but not the host memory). Each thread can read and/or write its private \textit{registers} and \textit{local memory} (for spilled registers). Each thread block has its private \textit{shared memory}, which can be read or written by the threads within the block. Globally, all threads have read and write access to the \textit{global memory}, and read-only access to the \textit{constant memory} and the \textit{texture 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.

2.2 CUDA-enabled GPU Architecture

Figure 2.2 shows the general architecture of an NVIDIA GPU that supports the CUDA programming model. The following description applies to a GeForce 8800GT, which belongs to the GeForce 8 series, the earliest generation of CUDA-enabled GPUs.

A GeForce 8800GT consists of 14 streaming multiprocessors (SMs). Each SM contains 8 streaming processors (SPs), which execute instructions issued by the instruction unit (one per SM) in an SIMD (single-instruction multiple-data) fashion. Each of the 112 SPs runs at 1.5GHz, and has a multiply-add arithmetic unit that can perform 32-bit single-precision integer/floating-point operations. Omitted in the figure, each SM also contains 2 special functional units (SFUs) that can execute more complex floating-point operations like inverse square root and trigonometric functions. A grid of GPU threads
is executed on the device by scheduling thread blocks onto the SMs. Each SM processes batches of thread blocks one after another. Once a thread block is scheduled on an SM, its entirety must be executed by this SM. Each thread block in a batch is split into groups of 32 threads called warps. The SM executes a warp in an SIMD fashion, by scheduling each thread in the warp to an SP. Note that it takes multiple cycles to issue an instruction to all 32 threads in a warp as there are only 8 SPs per SM. The way a thread block is split into warps is static, i.e. the first 32 threads form the first warp, so on and so forth.

The GPU has a complex memory hierarchy. Each SM has 8192 registers with single-cycle access latency and 16KB on-chip shared memory at nearly the same speed as registers. The shared memory has 16 banks, and performs the best when simultaneous requests are made to elements in different banks. The global, constant, texture and local memory (in the CUDA model) all reside on the off-chip device memory, accessible by all SPs as well as the host (that contains the CPU). The device memory is 512MB (for a typical GeForce 8800GT) and has 200-300-cycle access latency. Its bandwidth is best utilized when simultaneous accesses to the global memory are coalesced, i.e. form a contiguous segment with proper alignment [26]. Additionally, each SM has an 8KB single-ported cache for the constant memory, and every pair of SMs shares a 16KB cache for the texture memory. In contrast to the shared memory, these caches are managed by the hardware. The former can be accessed at nearly the register speed, while the latter requires more than 100 cycles of access latency.

The GPU architecture poses various constraints on the number of threads and thread blocks that can be simultaneously scheduled on an SM. First, the maximum number of threads, thread blocks and warps on an SM is 768, 8 and 24 respectively. Second, a thread block can contain at most 512 threads. Last, registers and shared memory for each SM are dynamically partitioned among the batch of thread blocks scheduled on the SM, so their sizes pose a limit on the number of threads and thread blocks in a batch. For example, if each thread needs 32 registers and each thread block needs 3KB of shared
memory, then an SM can process at most \(\lfloor \frac{16}{3} \rfloor = 5\) thread blocks, containing at most \(8192/32 = 256\) threads in total (or \(\lfloor \frac{256}{5} \rfloor = 51\) threads per thread block). Previous work \[34\] has shown that these hardware constraints are the main factor that makes the optimization of a CUDA program time-consuming because they result in a discontinuous optimization space.

### 2.3 CUDA Programming

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 that appear in the kernel code 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 \[26, 34\].

5. Perform other optimizations in the kernel in order to achieve an optimal balance between single-thread performance and the level of parallelism \[35\].

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 [35]. This not only increases development time, but also makes the program difficult to understand and to maintain. Consider the standard matrix multiply code in CUDA, shown in Figure 2.3. 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 increase the likelihood of coding mistakes, prolonging program development and debugging.

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

size = 64 * 128 * sizeof(float);
cudaMalloc((void**)&LA, size);
cudaMemcpy(LA, A, size, cudaMemcpyHostToDevice);
Size = 128 * 32 * sizeof(float);
cudaMalloc((void**)&LB, size);
cudaMemcpy(LB, B, size, cudaMemcpyHostToDevice);

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

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

matrixMul<<<dimGrid, dimBlock>>>(
    LA, LB, LC, 128, 32);

cudaMemcpy(C, LC, size, cudaMemcpyDeviceToHost);

data write-back (GPU to host) and deallocation

cudaFree(GLA);
cudaFree(GLB);
cudaFree(GLC);

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

    int aBegin = wa * 16 + tx;
    int aEnd = aBegin + wx;
    int aStep = 32;
    int bBegin = wb * 16 + tx;
    int bEnd = bBegin + wy;

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

Figure 2.3: Matrix multiply in CUDA.

2.4 Other GPGPU Programming Approaches

Apart from CUDA, there have been quite a few other interfaces for GPGPU programming developed in both industry and academia.
As a competitor to NVIDIA, AMD has released GPUs with unified programmable cores since late 2007. Their latest software stack (AMD Stream SDK v1.4beta) supports GPGPU programming in two levels of abstraction: Compute Abstraction Layer (CAL) at the lower level and Brook+ at the higher level [8]. CAL allows the programmer to write kernels in the assembly-level AMD Intermediate Language (IL), and is at a similar level of abstraction to Parallel Thread Extension (PTX) [27] in CUDA. Built on top of CAL, Brook+ provides a C-like stream programming environment, extended from the streaming language Brook [3] and its GPU implementation BrookGPU [4] developed at Stanford University. Although at a similar level of abstraction, Brook+ is not as popular as CUDA due to several reasons. First, its streaming language origin limits the flexibility of the kernel computation supported, even though most of the extensions added to Brook+ aim to address this issue. Second, Brook+ does not expose enough architectural features of the latest AMD GPUs (like local data store – the equivalent of shared memory in CUDA) to allow aggressive optimizations. This often forces the programmer to optimize kernels in CAL, which is much more tedious. Finally, the immaturity of the compiler support and the lack of detailed documentation often make the porting of an application to Brook+ a painful experience.

Apart from simplicity and performance, another important aspect in the design of GPGPU languages is portability. Apple Computer recently developed OpenCL [30], a standard API for programming both GPGPUs and multi-core CPUs. A large number of vendors have promised to support OpenCL, including NVIDIA, AMD and Intel. Overall, the concepts in this language can be closely mapped to those in CUDA. The basic unit of computation is a kernel, associated with an N-dimensional execution domain (the equivalent of a CUDA grid) to which the kernel is applied to. This exploits data-level parallelism. The execution of such kernels can proceed either in-order or out-of-order depending on the parameters passed to the system when queuing up the kernel for execution, thus exploiting task-level parallelism. During kernel execution, the elements in
an execution domain, called \textit{work-items}, are grouped into \textit{work-groups} for communication and synchronization purposes, which is equivalent to the concept of CUDA thread block. OpenCL supports a four-level memory hierarchy: private, local, constant and global memory from top to bottom. They directly correspond to registers, shared memory, constant memory and global memory in CUDA respectively. An important feature of OpenCL is the ability to probe the available hardware resources and adjust execution based on the current environment at run-time. As a result of this, the programmer needs to write more “setup” code in OpenCL than in CUDA.

Apart from AMD Brook+ and OpenCL, other GPGPU languages include Scout \cite{22} for scientific visualization and Intel Ct \cite{12} for throughput architectures. Both of them, along with Brook, are based on the \textit{shape} concept introduced by the C* programming language \cite{33} for the Connection Machine back in the early 1990s. An alternative to designing new languages for GPGPU programming is to provide a run-time library to existing languages like C/C++. Platforms that fall in this category include Accelerator \cite{36} from Microsoft Research and RapidMind \cite{24}. 
Chapter 3

The hiCUDA Language

3.1 Overview

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 \[1\]. 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.

The use of hiCUDA directives is illustrated with the popular matrix multiply code shown in Figure \[3.1a\]. 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 Figure \[3.1b\].

The loop nest ($i,j,k$) is targeted for execution on the GPU. Thus, it is surrounded
Chapter 3. The hiCUDA Language

14

(a) Original program.

```c
float A[64][128];
float B[128][32];
float C[64][32];

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

// C = A * B.
for (i = 0; i < 64; ++i) {
    for (j = 0; j < 32; ++j) {
        float sum = 0;
        for (k = 0; k < 128; ++k) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
printMatrix((float*)C, 64, 32);
```

(b) hiCUDA program.

```c
#pragma hiCUDA global alloc A[2][2]
#pragma hiCUDA global alloc B[2][2]
#pragma hiCUDA global alloc C[2][2]

// C = A * B.
#pragma hiCUDA kernel matrixMul tblock(16,16)
#pragma hiCUDA loop_partition over_tblock over_thread
for (i = 0; i < 64; ++i) {
    #pragma hiCUDA loop_partition over_tblock over_thread
    for (j = 0; j < 32; ++j) {
        float sum = 0;
        for (k = 0; k < 128; ++k) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
#pragma hiCUDA kernelEnd
```

Figure 3.1: Matrix multiplication example.

by the kernel directive (lines 13 and 33 of Figure 3.1b). 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.

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 Figure 3.1b) 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 block executes a 64/4 × 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 × 16/16 tile or a single iteration in the 16 × 16 tile assigned to the thread block. This partitioning scheme is shown in Figure 3.2a.

![Figure 3.2: The scheme of accelerating the matrix multiply code on GPU.](image)

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 3.1b). 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 [26]. 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 3.2b. 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 3.1b) 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 $1 \times 32$ 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 $(i, j, k)$. 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 3.3.

When compared with the hand-written CUDA version for matrix multiply (Figure 2.3), 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.

The remainder of this chapter describes the hiCUDA language in details, including the computation and the data models.
3.2 Computation Model

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

3.2.1 kernel Directive

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

```plaintext
#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.

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 \( B_1 \times B_2 \times \ldots \times B_n \) thread blocks, where \( B_i \) is the \( i^{th} \) `dim-sz` specified in `thread-block-clause`. Similarly, each thread block contains \( T_1 \times T_2 \times \ldots \times T_m \) threads, where \( T_i \) is the \( i^{th} \) `dim-sz` specified in `thread-clause`.

---

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.
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 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.

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 described 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.

### 3.2.2 loop_partition Directive

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:

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

Figure 3.3 shows an example of how the \texttt{loop\_partition} directive works. The first thing to note is that the associated loops of these directives can be arbitrarily nested within each other. If the directive contains the \texttt{over\_tblock} clause, its \textit{thread-block nesting level} (or \textit{LB}) is defined to be the nesting level (starting from 1) with respect to enclosing \texttt{loop\_partition} directives that also contain the \texttt{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 \texttt{loop\_partition} directive contains the \texttt{over\_thread} clause, its \textit{thread nesting level} (or \textit{LT}) is defined to be the nesting level (starting from 1) with respect to enclosing \texttt{loop\_partition} directives that also contain the \texttt{over\_thread} clause.

If a \texttt{loop\_partition} directive only has the \texttt{over\_tblock} clause, the iteration space of \textit{for loop} is distributed over the \textit{LB}\textsuperscript{th} dimension of the virtual thread-block space, specified in the enclosing \texttt{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 \(LB\textsuperscript{th}\) 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 \texttt{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 \texttt{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 \texttt{over\_thread} clause, each thread block executes all iterations of \textit{for loop}. Within the thread block, the iterations are distributed over the \(LT\textsuperscript{th}\)
Chapter 3. The hiCUDA Language

(a) A kernel in a hiCUDA program.

(b) Code executed by each GPU thread.

Figure 3.3: Example use of the loop_partition directives.
dimension of the virtual thread space, specified in the enclosing \texttt{kernel} directive.

If both the \texttt{over_tblock} and \texttt{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 \texttt{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 \texttt{j}. If \texttt{distr-type} in the \texttt{over_tblock} clause is \texttt{CYCLIC}, the actual distribution strategy is blocking-cyclic with block size \(T_{LT}\), where \(T_t\) 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 \texttt{loop_partition} directive supports uneven distribution of iterations over the threads. The \texttt{hiCUDA} compiler automatically generates “guard” code to ensure the exact number of iterations being executed.

In designing the \texttt{loop_partition} directive, we restrict the distribution strategy for the \texttt{over_thread} clause to be cyclic. This ensures that \textit{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.

### 3.2.3 singular and barrier Directives

By default, any code that is not partitioned among the threads is executed by \textit{every} thread. An example would be loop \texttt{p} in Figure 3.3a. Sometimes, however, this redundant execution could cause incorrect result or degraded performance. \texttt{hiCUDA} allows the programmer to identify kernel code to be executed \textit{only once} in a thread block, by enclosing it with two \texttt{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 3.3a 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 and do not share data.

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

### 3.3 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 (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 four 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 a **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 data main data directives is the same as that for a local C variable with respect to statements that refer to this variable.

### 3.3.1 **Global Directive**

The **global** directive has 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 ]`
Chapter 3. The hiCUDA Language

• [•], 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 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. 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.

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.

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 3.4 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
Figure 3.4: Array section specification in the global directives in the jacobi program.

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 3.4).

3.3.2 constant and texture Directive

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

```c
#pragma hicuda constant copyin \{variable\}+
#pragma hicuda constant remove \{var-sym\}+
```

The texture directive has two forms identical to those of the constant directive:

```c
#pragma hicuda texture copyin \{variable\}+
#pragma hicuda texture remove \{var-sym\}+
```

The first form specifies that, for each variable, a copy of it is to be statically allocated in the constant (or texture) memory and initialized with the content of variable in the host memory. Neither directive needs a copyout clause because the constant (and texture)
memory holds read-only data. The remove clause is the same as the free clause in the
global directive.

3.3.3 shared 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
#pragma hicuda 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 variable’s for all concurrently executed iterations. Consider the example shown in Figure 3.5a 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 3.5b 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 3.1b), the $16 \times 16$ threads in a thread block concurrently execute $16 \times 16$ 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 $16 \times 32$ tile of $A$ and a $32 \times 16$ 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.

(a) A kernel loop containing a shared directive.  
(b) Code executed by each GPU thread.

![Figure 3.5: Semantics of the shared directive.](image)

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 3.5a, 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.
Chapter 4

The hiCUDA Compiler

4.1 Overview

The hiCUDA compiler takes an input a set of files containing C code with hiCUDA directives and produces an equivalent CUDA program in 3 files: one for the GPU code (kernels), one for the CPU (host) code and a common header file. The compiler performs a number of key steps in this process. They are:

- **Kernel region identification.** The compiler identifies regions of code that are surrounded by the `kernel` and `kernel_end` directives. For each kernel region, the compiler verifies that it has a single entry and a single exit. In addition, the compiler ensures that no kernel region is nested within another, emitting an error in such cases. Since kernel regions may appear in different functions in the code, an inter-procedural framework is necessary to do this error checking.

- **Management of GPU data.** The compiler is responsible for generating code that manages the life-cycle of data residing on the device memory, as specified by the `global` and `constant` directives. For each `global` directive with an `alloc` clause, the compiler creates a new global memory variable and replaces the directive with an appropriate call to the CUDA run-time library that allocates global memory for
this variable. The size of the allocated memory is based on the section specified in the alloc clause. Similarly, calls to the CUDA run-time library are inserted to deallocate the global memory variables corresponding to the host variables specified by the free clause of a global directive. In the case of constant directives, the declaration of a constant memory variable is static, so its allocation and deallocation do not involve any code generation. The directives simply mark the scope of the constant memory variable.

For each global (or constant) directive with a copyin or copyout clause, the compiler inserts calls to the CUDA run-time library to transfer data between the host memory and the device memory. The clause specifies the data section of the host variable to be transferred. Since the CUDA run-time call can only transfer a contiguous chunk of data, the compiler must generate code that invokes the call multiple times (i.e. in a loop nest) for a non-contiguous data section.

- **Reaching directives analysis.** For each kernel region, the compiler must determine the global and constant directives that reach it, i.e. those directives whose lexical scopes enclose the kernel region. These directives may be defined in functions other than where the kernel itself is defined. Thus, an analysis similar to inter-procedural reaching definition analysis [25] is employed.

- **Data access analysis.** The compiler must conservatively determine data accesses made by each kernel region. This includes scalars as well as sections of arrays. These data accesses are used to ensure that the data copied into the GPU memory by reaching global and constant directives cover the data needed by the kernel region (an error is emitted if otherwise). Note that the compiler can tolerate cases where a scalar read (not modified) by a kernel region is not explicitly copied into the GPU memory – this variable will be passed into the kernel as a parameter (which resides in the shared memory). Since the kernel region may contain function
calls, the data access analysis must also be preformed inter-procedurally. The results of this analysis and the previous reaching directive analysis are combined to determine what GPU memory variables are needed by each kernel region. 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.

- **Outlining of kernel region.** After data access analysis, the compiler extracts each kernel region into a separate kernel function. Its parameters are the GPU memory variables determined previously, which also include those read-only scalar variables in the kernel region. Then the compiler replaces the original code region by an invocation to the kernel function with an execution configuration specified by the tblock and thread clauses of the kernel directive. Since the thread block space and the thread space supported by CUDA is limited to be 2-D and 3-D respectively, the virtual spaces specified in the two clauses are first mapped to 2-D and 3-D. For example, tblock(2,3,4,5) is mapped to a 3-D thread block space blockDim.(x,y,z) = (4x5,3,2).

- **Kernel access redirection.** Once a kernel body is outlined into a separate function, accesses to scalars and arrays inside must be redirected to their corresponding GPU memory variables. Handling scalars is relatively straightforward as it only involves variable replacement. Handling an array access is a bit more involved since it is possible for only a section of the array to be allocated in the GPU memory. In such cases, the offset of each dimension of the access needs to be adjusted based on the allocated section’s starting offset (for this dimension). Furthermore, access redirection must be done inter-procedurally since the scalars and arrays may be passed as arguments to functions called within the kernel body.

- **Kernel loop partitioning.** The compiler translates each loop partition dire-
tive 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. For example, distributing a simple loop
\[
\text{for (i = 0; i < 5; ++i)}
\]
among 3 threads (cyclically) results in loop
\[
\text{for (i = threadIdx.x; i < 5; i += 3)}
\]
in the kernel function. The code generation is straightforward in most cases, except when the loop iterations are distributed among thread blocks in a \text{BLOCK} fashion. In such a case, if the compiler is not certain that the distribution is even, it must insert code to prevent certain thread blocks to execute extra iterations. As an optimization, if the compiler is certain that each thread executes at most one iteration of the loop, it eliminates the loop, and optionally replaces it with a conditional branch.

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 \text{loop.partition} directive. Since \text{loop.partition} directives may be placed in functions called within a kernel region, an inter-procedural propagation of kernel contextual information is required.

- **Utilizing the shared memory.** The translation of \text{shared} directives is more complicated than that of \text{global} and \text{constant} directives. First, when declaring a shared memory variable, the compiler can not simply determine the variable shape based on the array section specified in the directive. It must merge this section across all iterations of the enclosing loops that are concurrently executed by a thread block. This ensures that the shared memory variable can host data needed by all concurrent threads. Since the \text{shared} directives and the enclosing loops may not be in the same function, an inter-procedural propagation of loop contextual information is required to do section merging. Second, data transfer between the shared memory and the global memory is done cooperatively by all threads in a thread block. The compiler determines an assignment of array elements to each thread so that both the global and the shared memory are utilized optimally during
the transfer.

- **Optimization of GPU memory allocation.** The creation of a GPU memory variable (due to a `global`, `constant` or `shared` directive) normally involves allocating a *new* piece of GPU memory. This is expensive for the shared and constant memory which only support static allocation. Therefore, in such cases, the compiler reuses memory for variables whose live ranges do not overlap. This is a standard *compile-time memory allocation* problem that can be solved by using an approach similar to graph coloring for register allocation [6].

- **Support of dynamic arrays.** To support the use of dynamically allocated arrays, *hiCUDA* provides a `shape` directive that allows the programmer to specify the shape of such arrays. The compiler first propagates this information inter-procedurally and then promotes all array references made through pointers (e.g., `*(ptr + offset)`) into regular array references (i.e., `ptr[offset]`). This allows the above data analyses to be used with minimal modification. The process of pointer promotion involves factoring 1-D access offsets into multi-dimensional offsets, which must be verified to be within the corresponding array bounds using a linear programming solver.

The following sections describe in details how each *hiCUDA* directive is translated into CUDA code and how the translation is done in an inter-procedural context.

### 4.2 Kernel Region Identification

A kernel region is surrounded by the `kernel` and `kernel_end` directives. A valid kernel region must meet the following criteria:

1. It has a single entry at the beginning of the region and a single fall-through normal exit.
2. It is not nested within another kernel region.

3. It does not contain any global or constant directives.

The first criterion can be verified intra-procedurally, while the latter two must be done inter-procedurally because a kernel region may contain function calls. For example, the compiler must be able to screen out cases shown in Figure 4.1. In Figure 4.1a, the kernel region k_bar is nested within kernel region k_foo, making the latter invalid. In Figure 4.1b, the global directives in procedure bar are actually inside kernel region k_foo, making it invalid.

(a) Nesting of kernels. (b) global or constant directives inside kernels.

Figure 4.1: Cases of invalid kernel regions in an inter-procedural context.

To validate kernel regions, the compiler first classifies each procedure into one of the following four categories:

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

Ensuring that the four categories are disjoint essentially guards against kernel nesting (the second criterion). This is because, if kernel region A is nested within kernel region B (which could be A itself), the procedure containing B is both a K-procedure and an IK-procedure.

Figure 4.2 shows the call graph partitioned into the four categories and illustrates the call flow directions among them. MK-/K-procedures can call IK-/N-procedures but not the other way because otherwise these IK-/N-procedures would have been MK-procedures. MK- and K-procedures can call each other. Note that a call from a K-procedure to an MK-procedure must happen outside kernel regions, because otherwise potential kernel nesting occurs. N-procedures can call IK-procedures but not the other way because otherwise these N-procedures would have been IK-procedures.

Figure 4.2: Call flow directions among the four categories of procedures.

The classification process consists of four steps. First, K-procedures are identified. Second, a flag $IK$ that indicates whether or not the current procedure may be inside a kernel region is propagated inter-procedurally in a forward direction. The initial group of procedures that have this flag set is those that are directly called within kernel regions. Third, a flag $MK$ that indicates whether or not the current procedure may lead

---

1Here, the forward propagation direction is from caller to callee; the backward direction is from callee to caller.
to a kernel region is propagated inter-procedurally in a backward direction. The initial

group of procedures that have this flag set is the K-procedures. Last, each procedure is
classified as an MK-procedure if it is not a K-procedure but has the MK flag set, and
as an IK-procedure if it has the IK flag set. Potential kernel nesting occurs when a
procedure has both flags set, and this error is reported.

Once the classification is done, the compiler can easily verify the third criterion by
ensuring that 1) there is no global or constant directives within kernel regions in a
K-procedure, and 2) there is no global or constant directives in an IK-procedure. The
compiler ignores data directives in N-procedures but emits a warning.

Not only is the classification scheme used to validate kernel regions, it is used through-
out the analyses described in the following sections. For example, reaching directives
analysis (Section 4.4) is done among K-functions and MK-functions; access redirection
is done among K-functions and IK-functions.

4.3 GPU Data Management

The global and constant directives specify the life-cycle of data residing on the device
memory. They are directly translated into CUDA code in place.

4.3.1 Allocation and Deallocation

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
locally and its type is a pointer to the type of H if H is a scalar or a struct, or a pointer to
the element type of H if H is an array. To allocate global memory for this new variable,
the compiler inserts a call to the CUDA runtime library:

    cudaMalloc((void**)&g_H, <size in bytes>);
The size of the allocated memory is the size of \( H \)'s type if \( H \) is a scalar or a struct, or the size of the array section specified in the `alloc` clause if \( H \) is an array. Deallocation of \( g.H \) is specified by a `global` directive with a `free` clause that contains the host variable \( H \). The compiler inserts another call to the CUDA runtime library to deallocate global memory:

\[
\text{cudaFree}(g.H);
\]

The handling of `constant` directives is slightly different because a constant memory variable must be declared statically. For each `constant` directive with a `copyin` clause, the compiler simply declares a corresponding constant memory variable globally and ensures that its size is constant. The deallocation of this variable, specified by a `constant` directive with a `remove` clause, does not have any corresponding CUDA code. This directive simply marks the end of the variable's scope.

Since the constant memory allocated for a `constant` directive persists outside the directive scope, it can be reused for other `constant` directives with non-overlapping 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 in the program. In this scheme, a single constant memory variable (\( \text{cmem} \)) is declared globally. The new variable (named \( c.H \)) created for each `constant` directive with a `copyin` clause is a `local` variable initialized to point to \( \text{cmem} \) at an appropriate offset determined by the algorithm. Section 4.10.2 gives details of the algorithm.

### 4.3.2 Data Transfer

In a `global` directive, the `copyin` clause specifies the transfer of data from the host memory to the global memory, which is translated into a call to the CUDA runtime library:
cudaMemcpy(g_H, &H, <transfer size in bytes>, cudaMemcpyHostToDevice);

If \( H \) is a scalar or a struct, this call is sufficient and the transfer size is simply the size of the allocated global memory. The translation gets more complicated if \( H \) is an array (i.e. an array section is transferred). Since a call to \texttt{cudaMemcpy} only transfers a contiguous chunk of data, it needs to be invoked multiple times (i.e. inside a loop nest) for an array section that contains strided data, e.g. \( H[*][1:2] \) where \( H \) is a \( 5 \times 5 \) matrix.

To formally describe the transfer of an array section, consider a general \( n \)-dimensional array \( A \) of shape \([D_1][D_2]...[D_n]\). Its section \([S_1 : E_1][S_2 : E_2]...[S_n : E_n]\) is allocated in the global memory, and a section \([s_1 : e_1][s_2 : e_2]...[s_n : e_n]\) needs to be copied from the host memory to the global memory. Let \( p \) be the index of the innermost (or rightmost) dimension of the section to be transferred whose range is less than the corresponding dimension size of \( A \). Since all dimensions to the right of dimension \( p \) are fully transferred, the biggest contiguous sub-section that can be transferred includes these dimensions plus dimension \( p \). More formally,

\[
p = \max \{ d \in [1, n] | s_d > 0 \lor e_d < D_d - 1 \}
\]

\[
= \min \{ d \in [1, n] | \forall d' \in [d + 1, n], (s_{d'} = 0 \land e_{d'} = D_{d'} - 1) \}
\]

Figure 4.3a is a conceptual template of the CUDA code to be generated for the data transfer, in which the global memory variable \( gA \) is treated as an \( n \)-dimensional array of shape \([S_1 : E_1][S_2 : E_2]...[S_n : E_n]\). The variable \texttt{batsz} holds the size of the contiguous sub-section to be transferred (in bytes), which is

\[
<\text{batch\_size}> = (e_p - s_p + 1) \times ( \prod_{d=p+1}^{n} D_d ) \times \text{element\_size\_in\_bytes}
\]
The loop nest iterates through the subspace \([s_1 : e_1][s_2 : e_2]...[s_{p-1} : e_{p-1}]\) to transfer all contiguous subsections. The two addresses passed to \texttt{cudaMemcpy} point to the beginning of the subsection in \texttt{g\_A} and \texttt{A} respectively. Since both \texttt{A} and \texttt{g\_A} could be dynamically-allocated arrays, the actual CUDA code must represent these array accesses in the form of \texttt{ptr + offset}. Figure 4.3b shows the actual code template, which factors out the offset components that are constant over iterations (stored in variable \texttt{stride}). In this code,

\[
<\text{stride}> = \prod_{d=p+1}^{n} D_d
\]

\[
<g\_offset> = \sum_{d=1}^{p-1} ((i_d - S_d) \times \prod_{d'=d+1}^{p} (E_{d'} - S_{d'} + 1)) + (s_p - S_p)
\]

\[
<h\_offset> = \sum_{d=1}^{p-1} (i_d \times \prod_{d'=d+1}^{p} D_{d'}) + s_p
\]

Furthermore, since \texttt{<batch\_size>} contains the expression \texttt{<stride>}, it can be simplified as:

\[
<\text{batch\_size}> = (e_p - s_p + 1) \times \texttt{stride} \times \text{element\_size\_in\_bytes}
\]

The data transfer from the global memory to the host memory, specified by the \texttt{copyout} clause of a \texttt{global} directive, is translated in the same way except that the source and destination addresses passed to \texttt{cudaMemcpy} are swapped and the copy direction parameter becomes \texttt{cudaMemcpyDeviceToHost}. 
The copyin clause of a constant directive is handled in the same way except that the call to the CUDA runtime library takes the form:

\[
\text{cudaMemcpyToSymbol("c_H", H, }
\text{<transfer size in bytes>,}
\text{<offset of c_H in bytes>,}
\text{cudaMemcpyHostToDevice);}
\]

Figure 4.4 shows examples of CUDA code translated from global and constant directives. Note that the code generated by the prototype compiler will look different because it performs constant propagation, which will eliminate variables like stride in simple cases.

(a) Translation of global directives.

(b) Translation of constant directives.

Figure 4.4: Translation of global and constant directives.
4.4 Reaching Directives Analysis

The compiler must determine the global and constant directives (in the first form) that reach each kernel region, so that it can redirect data accesses in the region to the corresponding GPU memory variables. Here, the word “reach” means that the kernel region is within the scope bounded by a global/constant directive and its matching directive with a free clause, both of which must appear in the same lexical scope.

Since the data directives “needed” by a kernel region may not be placed in the same procedure as the kernel directive, an inter-procedural reaching directives analysis is used. Figure 4.5 shows a typical example of this case. The global directive for A cannot be placed inside procedure foo or bar because it is needed by both.

```c
void foo(int &a, int len)
{
    #pragma hicuda kernel k_foo1 block(l) thread(4)
    for (int i = 0; i < len; ++i)
        a[i]++;
    #pragma hicuda kernel_end
}

void bar(int &a, int len)
{
    #pragma hicuda kernel k_foo2 block(l) thread(4)
    for (int i = 0; i < len; ++i)
        a[i]--;
    #pragma hicuda kernel_end
}

int main()
{
    int A[5];
    foo(&A, 5);
    bar(&A, 5);
    #pragma hicuda global free A
}
```

Figure 4.5: Motivation of inter-procedural reaching directives analysis.

Data directives are propagated among K- and MK-procedures. More specifically, an annotation is attached to each procedure. It contains information about the global or constant directive (if any) associated with each formal parameter of the procedure. The propagation starts with an “empty” annotation (i.e. no data directive for any formal parameter) in the main function. Figure 4.6 shows the core algorithm of the propaga-
tion. For each function call, a new annotation is constructed for the callee, taking into account the data directives in the caller’s annotation and those local in the caller procedure. A data directive can be associated with a formal parameter of the callee if 1) the corresponding actual parameter is a single variable (as opposed to an expression) that matches the variable in the directive, and 2) the allocated array section specified by the directive can be mapped to the callee space, i.e. all auxiliary variables referenced in section specification (e.g. \texttt{start\_var} and \texttt{end\_var} in alloc \texttt{A[start\_var:end\_var]}) are either global variables or passed to the callee. Once the annotation is constructed, it is added to the callee only if it is unique among the existing annotations in the callee. The equivalence check of two annotations involves symbolic comparison of the section specifications. Figure 4.7a gives an example that demonstrates the propagation algorithm. Procedure \texttt{foo} is called in \texttt{main} and \texttt{foo\_run}. Through the first call, the global directive for \texttt{arr} is propagated to procedure \texttt{foo} as \texttt{a[0:(len-1)]}. Through the second call, the local global directive for \texttt{b} is propagated to procedure \texttt{foo}, even though there is another global directive for \texttt{b} in the annotation of procedure \texttt{foo\_run}. Since the two annotations propagated to procedure \texttt{foo} are equivalent, there is only one annotation in \texttt{foo}.

After data directive propagation, each reachable K-/MK-procedure has one or more unique annotations. The compiler performs cloning on these procedures: one clone for each annotation. Thus, a procedure with \(N\) annotations will be cloned \(N-1\) times. Call edges to and from the clones are constructed based on call path information collected during propagation. Note that the category (i.e. K-/MK-procedure) of each clone follows that of the original node.

Since increase in code size could greatly affect the performance of a GPU program, the compiler must minimize the number of clones to be created. Since some data directives propagated to a K-procedure may not be used by the kernel regions, they should not be considered in the equivalence check of annotations. Therefore, as an optimization, the
for each K-/MK-procedure caller do
    for each new annotation caller_annot do
        for each call in caller do
            if callee is a K-/MK-procedure then
                callee_annot ← an empty annotation for the callee
                for each actual parameter var of the call do
                    if var is a single variable then
                        dir ← data directive associated with var in caller
                        if dir does not exist and var is a formal parameter of caller then
                            dir ← the data directive associated with var in caller_annot
                        end if
                        if dir exists then
                            map dir to the callee space
                            store the mapped dir in callee_annot
                        end if
                    end if
                end for
                if callee_annot is unique among callee’s annotations then
                    add callee_annot to the callee and mark it new
                    indicate the propagation process to be repeated
                end if
            end if
        end for
    end for
end for

Figure 4.6: Core algorithm of reaching directive propagation.

cloning process is delayed until the kernel data access analysis (Section 4.5) is performed, during which the “used” propagated data directives for each kernel region are determined.

To determine all data directives “used” in each K-/MK-procedure, the compiler starts with those marked “used” by the data access analysis and back-propagates this information inter-procedurally (i.e. from callee to caller). Once this is done, “useless” directives (in each K-/MK-procedure) are removed from the annotation(s) and newly equivalent annotations in each procedure are merged. Finally, the compiler performs procedure cloning as mentioned above. Figure 4.7b shows the effect of this optimization. After reaching directive propagation, procedure foo.run has two annotations, resulting from
Chapter 4. The hiCUDA Compiler

43

(a) Basic example.

```c
void foo(float *a, int len) {
    #pragma hicuda kernel l_foo thread(4)
    for (int i = 0; i < len; ++i)
        a[i] = a[i] + 1.0f;
} #pragma hicuda kernelEnd

void foo_run(float *b, int size) {
    b = malloc(size * sizeof(float));
    #pragma hicuda shape b[size]
    #pragma hicuda global Alloc b[*] copyin
    foo(b, size);
    #pragma hicuda global free b
}

int main() {
    float A[5];
    float B[5];
    #pragma hicuda global Alloc A[*] copyin
    foo_run(A, B);
    #pragma hicuda global Alloc B[*] copyin
    foo_run(B, A);
    #pragma hicuda global free A B
}
```

(b) Example that demonstrates the optimization.

```c
void foo(float *arr, int len) {
    #pragma hicuda kernel l_foo thread(4)
    for (int i = 0; i < len; ++i)
        arr[i] = arr[i] + 1.0f;
} #pragma hicuda kernelEnd

void foo_run(float *a, float *b) {
    foo(a, 5);
    b[0] = 0;
}

int main() {
    float A[5];
    float B[5];
    #pragma hicuda global Alloc A[*] copyin
    foo_run(A, B);
    #pragma hicuda global Alloc B[*] copyin
    foo_run(B, A);
    #pragma hicuda global free A B
}
```

Figure 4.7: Examples of reaching directives analysis.

the two invocations in main. The kernel data analysis marks the single annotation in procedure foo “useful”, and back-propagation of this flag results in a[0:4] being marked “useful” in both annotations of procedure foo_run. Thus, b[0:5] is removed from the second annotation and the two annotations are considered equivalent. This results in a single annotation for foo_run and no clones need to be created.

After reaching directives analysis and the translation of all global and constant directives, the compiler expands the list of formal parameters of each K-/MK-procedure so that the GPU memory variables associated with the reaching global and constant directives are passed into the procedure. For each call to these procedures, the list of actual parameters is expanded accordingly. In the example (Figure 4.7a), the prototype of procedure foo and foo_run becomes foo(float *a, int len, float *g_a) and foo_run(float *b, int size, float *g_b) respectively. The call to foo in foo_run takes an extra parameter: the global memory variable associated with the local global directive for b. Similarly, each call in main takes an extra parameter: the global memory variable associated with the local global directive for arr.
4.5 Data Access Analysis

Before transforming a kernel region into a kernel function, the compiler must collect all data accesses (scalars and sections of arrays) made by the region and determine how each of them 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.

Since a kernel region may contain function calls, the data access analysis must be performed inter-procedurally, among K- and IK-procedures. First, the side effect of each procedure is determined, using inter-procedural array section analysis [16]. It includes formal parameters and global variables that are referenced and modified. Note that the side effect of an array variable is its regular sections, each dimension of which has the form [start_idx:end_idx:stride].

Second, the side effect of each kernel region is determined. Array sections accessed are determined by projecting each array access onto index variables of enclosing loops up to the kernel region boundary. This process is essentially the same as the transfer function used in the previous analysis [16] except that the code region is bounded by the kernel directives (as opposed to the entire procedure). 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 going across the region boundary. Finally, the side effect of each function called within the kernel region is mapped to the caller space and merged into the side effect of the kernel region.

Figure 4.8 illustrates the data access analysis using an example. First, the side effect of procedure process_array is determined. The reference to scalar formal count comes from loop i. The accesses to arr and glob_arr are projected onto i. Note that the section
Figure 4.8: Example of inter-procedural data access analysis.

specification of an array variable can use scalar formal parameters (or global variables). Second, the side effect of kernel \texttt{k\_test} is determined. The access to \texttt{local\_arr} is again projected, and the side effect of \texttt{process\_array} is merged in.

After collecting data accesses made by a kernel region, the compiler matches each access with the reaching \texttt{global} or \texttt{constant} directive for the same variable. When there are multiple such directives, the one closest to the kernel region is selected; the rest are ignored. In the example (Figure 4.8), accesses to \texttt{local\_arr} are matched with the \texttt{global} directive for this variable as opposed to the \texttt{constant} directive placed before. If a matched directive is found, the compiler performs the following checks and emits an error if the condition fails:

- The matched directive for a modified access can not be a \texttt{constant} directive.

- The matched directive for an array access must bring a section into the GPU memory that covers the access.

In the previous example, the compiler can not determine whether or not the section \texttt{local\_arr[0:n-1:1]} is covered by \texttt{local\_arr[*]} (i.e. \texttt{local\_arr[0:9:1]}), but it does
not complain because the entire section of local_arr is brought into the global memory. Finally, if no matched directive is found, the compiler first ensures that the access is scalar and read-only, and then marks the variable as a kernel parameter.

Sometimes accurate array section information cannot be determined, due to the use of pointer arithmetic or the limitation of regular section representation [start_idx:end_idx:stride]. In the former case, the compiler will not perform the check of section coverage. In the latter case, the compiler conservatively assumes that the entire array is accessed and emits a warning (as opposed to an error) if the check of section coverage fails.

### 4.6 Kernel Access Redirection

After data access analysis, each access made by a kernel region has a target of redirection, i.e. the GPU memory variable associated with one of the reaching global and constant directives. Before performing the actual redirection, the compiler must inter-procedurally propagate these targets to the functions called within the kernel region.

GPU memory variables are propagated among IK-procedures. A procedure’s annotation contains the following information:

- the GPU memory variable (if any) associated with each formal parameter
- the GPU memory variable associated with each global variable accessed by the procedure (as determined in the data access analysis)

The propagation starts from the IK-procedures that are directly called within kernel regions. Their initial annotations are constructed by propagating GPU memory variables needed by each kernel region (in a K-procedure) across each function call within it. The process then continues by propagating GPU memory variables in each IK-procedure’s annotation(s) to the procedures called within. For each function call, a new annotation
is constructed for the callee. The actual parameters of the call fall into three categories:
1) scalar expressions, 2) addresses of scalar variables, and 3) arrays (static or dynamic).
For variables referenced in parameters in the first category, their associated GPU memory variables (if any) do not need to be propagated to the callee because these parameters are passed by value. The same is true for parameters in the second category because the GPU memory variable associated with a scalar variable has the same type as its address, resulting in no changes to the callee’s function prototype. For each actual parameter in the third category that involves a single array variable, the associated GPU memory variable will be propagated to the corresponding formal parameter of the callee. Note that the compiler can not handle an actual parameter that is an address expression involving an array variable that will be redirected (e.g. &A + 1 while A has a corresponding global memory variable g.A). It emits an error in such cases. Apart from GPU memory variables associated with the actual parameters, those associated with the global variables accessed by the callee are also added to the annotation. Once the annotation is constructed, it is added to the callee only if it is unique among the existing annotations in the callee. Two annotations are considered equivalent if the GPU memory variable associated with each formal and global variable is the same, i.e. they come from the same `global` or `constant` directive in the same procedure.

Figure 4.9 illustrates the propagation of GPU memory variables using an example. From the data access analysis, kernel `k_test` reads and modifies sections of `local_arr1`, `local_arr2` and `glob_arr`, which can be associated with the corresponding `global` directives in the main procedure. After propagating the associated GPU memory variables, each of procedure `foo` and `bar` has two annotations, one from each call of `foo` in `main`. Three observations are worth noting. First, the section specification (if any) of each GPU memory variable is propagated along the way because it will be used during the actual access redirection. Second, variables referenced in section specifications (e.g. n) are not mapped to the callee space when propagating across a function call, because they may
Chapter 4. The \texttt{hiCUDA Compiler}

Figure 4.9: Example of kernel access redirection.

```c
float globalArr[10];
void bar(float *arr)[10]
{
    arr[0] = 0;
}
void foo(float *arr)[10], int *lbd)
{
    int i;
    for (i = lbd; i < 10; ++i)
        globalArr[i] = arr[i];
    bar(arr);
}
int main(int argc, char **argv)
{
    float localArr1[10];
    float localArr2[10];
    int n = min(argc, 9);
    #pragma hicuda global alloc localArr1[n:9]
    #pragma hicuda global alloc localArr2[n:9]
    #pragma hicuda global alloc globalArr[10]
    bar(globalArr);
    #pragma hicuda kernel i-test thkock(1) thread(1)
    foo(localArr1, n);
    foo(localArr2, n);
    #pragma hicuda kernel end
    #pragma hicuda global copyout globalArr[10]
    #pragma hicuda global free globalArr localArr1 localArr2
    return 0;
}
```

not be actual parameters of the call (e.g. \texttt{n} cannot be mapped to the context of procedure \texttt{bar}). Last, the compiler considers the annotations resulted from the two calls of \texttt{foo} different due to the different GPU memory variables propagated to the formal parameter \texttt{arr}, even though the two variables have the same section specification.

After the propagation of GPU memory variables, each IK-procedure that is reachable from kernel regions has one or more unique annotations. The compiler performs cloning on these procedures, which is similar to the one in reaching directives analysis. The cloning of an IK-procedure requires special handling if it is called outside a kernel region, which could come from any non-IK-procedures. In this case, the original IK-procedure must be preserved in order to keep these “regular” function calls correct, which is achieved by associating the procedure with a dummy “empty” annotation. In the example (Figure 4.9), procedure \texttt{bar} will be cloned twice even though it has only two
annotations, because it is called by \texttt{main} outside the kernel region \texttt{k.test}. Figure 4.10a shows the resulting CUDA code, in which procedure \texttt{bar} and the call of \texttt{bar} from \texttt{main} are preserved. Note that \texttt{bar} is now an N-procedure while its clones remain IK-procedures. Therefore, the compiler must perform procedure classification (Section 4.2) again after cloning.

Once procedure cloning is performed, the GPU memory variables that correspond to accesses within each IK-procedure have been determined. Before access redirection, the compiler must map these variables (along with their section specifications) to the local procedure context. Each existing formal parameter is replaced with its associated GPU memory variable (if any). A new formal parameter is created for 1) the GPU memory variable associated with each global variable accessed in the procedure, and 2) each scalar variable used in the section specifications. After the list of formal parameters of each IK-procedure is expanded, the list of actual parameters of each corresponding call is updated accordingly. As shown in Figure 4.10a, a formal parameter is added to each clone of \texttt{bar} (\texttt{barclone0} and \texttt{barclone1}) to pass the lower bound \texttt{n} of the section of \texttt{local$arr1} and \texttt{local$arr2} in the global memory. Note that a new formal parameter is added to \texttt{foo} and its clone \texttt{foocclone0} for the same purpose even though the existing parameter \texttt{lbnd} can be used.

The actual access redirection is very simple. Table 4.1 summarizes the translation for each type of accesses. Three things are worth noting. First, an array access in the form of $*(ptr+offset)$ must be converted into the regular form $ptr[i_1][i_2]...[i_n]$ before applying the translation shown in the table. Second, the offset of a redirected array access must be expanded to 1-D for a multi-dimensional array variable because its corresponding GPU memory variable is dynamically-allocated (in the global memory) or a pointer to a statically-allocated region (in the constant memory) at some offset. Last, references to an array variable (that needs to be redirected) \texttt{outside} array accesses are not allowed. The compiler emits an error in such cases.
Chapter 4. The hiCUDA Compiler

Figure 4.10: CUDA code generated from the example program (Figure 4.9).

(a) Basic version.

```c
#define _global_

void k_test(float * g_localAr1,
float * g_globalArr, float * g_localArr2, int n)
{
  for (i = 0; i < n; i++)
  {
    g_globalArr[i] = g_localArr2[i] + g_globalArr[i];
  }
}

device void barclone(float * g_localArr2, int n)
{
  g_localArr2[0] = 0.0;
}

device void barclone(float * g_localAr1, int n)
{
  g_localAr1[0] = 0.0;
}

device void foo(float * g_localArr2, int lbind,
float * g_globalArr, int n)
{
  int i;
  for(i = lbind; i < n; i++)
  {
    g_globalArr[i] = g_localArr2[i - n] + g_globalArr[i];
    barclone(g_localAr2, n);
  }
}

device void fooclone(float * g_localAr1, int lbind,
float * g_globalArr, int n)
{
  int i;
  for(i = lbind; i < n; i++)
  {
    g_globalArr[i] = g_localAr1[i - n] + g_globalArr[i];
    barclone(g_localAr1, n);
  }
}

void bar(float * arr)
{
  arr[9] = 0.0;
}

int main(int argc, char ** argv)
{
  float localAr1[1001];
  float localAr2[1001];
  int n;
  float * g_localAr1;
  float * g_localAr2;
  float * g_globalArr;
  struct dim3 dimgrid;
  struct dim3 dimgblock;
  ...
  bar(localAr1);
  k_test << dimgrid, dimgblock >> (g_localAr1,
globalArr, g_localAr2, n);
}
...
```

(b) Optimized version.

```c
#define _global_

void k_test(float * g_localAr1,
float * g_globalArr, float * g_localArr2, int n)
{
  foo(g_localAr1, n, g_globalArr);
  foo(g_localArr2, n, g_globalArr);
}

device void barclone(float * g_localAr1, int n)
{
  g_localAr1[9] = 0.0;
}

device void barclone(float * g_localAr2, int n)
{
  g_localAr2[0] = 0.0;
}

device void foo(float * g_localAr2, int lbind,
float * g_globalArr)
{
  int i;
  for(i = lbind; i < n; i++)
  {
    g_globalArr[i] = g_localAr2[i - lbind] + g_globalArr[i];
    barclone(g_localAr2, lbind);
  }
}

void bar(float * arr)
{
  arr[9] = 0.0;
}

int main(int argc, char ** argv)
{
  float localAr1[1001];
  float localAr2[1001];
  int n;
  float * g_localAr1;
  float * g_localAr2;
  float * g_globalArr;
  struct dim3 dimgrid;
  struct dim3 dimgblock;
  ...
  bar(localAr1);
  k_test << dimgrid, dimgblock >> (g_localAr1,
globalArr, g_localAr2, n);
}
...
Table 4.1: Kernel access redirection.

<table>
<thead>
<tr>
<th>Original Access</th>
<th>Redirected Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>*g_scalar(^a)</td>
</tr>
<tr>
<td>&amp;scalar</td>
<td>g_scalar</td>
</tr>
<tr>
<td>arr[i_1][i_2]...[i_n]</td>
<td>g_arr[i_1 - S_1][i_2 - S_2]...[i_n - S_n] (^b)</td>
</tr>
</tbody>
</table>

\(^a\)g\_scalar is a global memory variable. The prefix for a constant memory variable is c\_.

\(^b\)S\_i is defined in Section 4.3.2.

This completes the description of the basic algorithm of inter-procedural access redirection. As shown in the example (Figure 4.10a), this algorithm does not generate the most optimized CUDA code. The two annotations of procedure foo could have been merged (same for bar). The extra formal parameter n of foo and fooclone0 is not necessary. To support these optimizations, variables used in the section specifications must be propagated along with the GPU memory variables and mapped across function calls. Therefore, each procedure’s annotation is expanded to store a mapping between these scalar variables and the corresponding formal parameters (if any) that hold their values. Now the equivalence check of two different GPU memory variables associated with a formal parameter can be more accurate if each variable referenced in the two section specifications can be mapped to a formal parameter. Figure 4.9 shows the expanded annotations (for procedure foo). The GPU memory variables g\_local\_arr1 and g\_local\_arr2 are considered equivalent because both of their sections are essentially [lbnd:9]. As a result, the second annotation for foo (and for bar) is not created during propagation and each procedure has one less clone. In addition, the mapping is used to avoid adding new formal parameters unnecessarily when mapping the section specifications of GPU memory variables to the local context of each IK-procedure, as illustrated in procedure foo in Figure 4.10b. Note that, in this example, procedure barclone0 still requires an extra formal parameter to hold n because n cannot be mapped to bar using an existing formal parameter.
4.7 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 each thread executes. The number of thread blocks and threads to which the loop iterations are distributed is determined by the nesting level of the directive and the virtual thread block and thread spaces specified in the `kernel` directive respectively. Since `loop_partition` directives may be placed in functions called within the kernel region, the compiler must determine their kernel context information inter-procedurally.

4.7.1 Kernel Context Propagation

First and foremost, the compiler verifies that `loop_partition` directives are placed inside kernel regions. With the procedure classification scheme in place, it just needs to do the following checks:

- No `loop_partition` directives in MK-procedures.
- No `loop_partition` directives in K-procedures that are outside kernel regions.

The compiler silently ignores `loop_partition` directives in N-procedures because an N-procedure could be a clone of an IK-procedure (resulting from GPU memory variable propagation).

After placement validation, the compiler propagates kernel context information among IK-procedures. A procedure’s annotation includes:

- the current thread-block and thread nesting level of `loop_partition` directives (i.e. \(LB\) and \(LT\) defined in Section 3.2.2)
- the geometry of the virtual thread block space and the virtual thread space
Similar to the propagation of GPU memory variables, the propagation starts from those procedures directly called inside kernel regions and continues by propagating each IK-procedure’s annotation to procedures called within. Whenever a `loop.partition` directive is encountered, the running kernel context is updated based on the presence of `over.tblock` and `over.thread` clauses. Figure 4.11a illustrates this process using an example. The arrows in the annotations indicate the next nesting levels of the thread block and thread spaces. Since procedure `foo` is nested within loop `i`, which is associated with an `over.tblock` clause, the kernel context propagated to `foo` has `LB` incremented by 1. Similarly, since both calls to `bar` (in `foo` and `main`) are nested within an `over.tblock` clause and an `over.thread` clause, a single kernel context is propagated to `bar`, which has both `LB` and `LT` incremented by 1.

After kernel context propagation, the compiler performs procedure cloning and reclassifies the procedures as done in access redirection. Finally, the compiler determines the kernel context for each `loop.partition` directive (i.e. which dimensions of the thread block and thread spaces to which the associated loop is distributed) in each K- or IK-procedure, based on the kernel region information (in K-procedures) or the kernel context annotation (in IK-procedures).

### 4.7.2 Handling `loop.partition` Directives

For each `loop.partition` directive, the compiler first pre-processes its associated loop, which includes the following three steps:

1. normalizing the comparison operator used in end-condition check to be $\leq$ or $\geq$,

2. verifying the consistency between the end-condition check and the sign of the loop step, and

3. screening out empty loops.
Figure 4.11: Example of kernel loop partitioning.
Currently, the loop step must be a compile-time constant; otherwise the compiler emits an error.

Consider a standard loop in the following form:

\[
\text{for } (i = \text{init}\_\text{expr} ; i \text{ op } \text{end}\_\text{expr} ; i += \text{step}\_\text{expr})
\]

After the first step, \(\text{op}\) must be \(\leq\) or \(\geq\). In the second step, the compiler ensures that 1) \(\text{step}\_\text{expr} \neq 0\), and 2) \(\text{op}\) is \(\leq\) if \(\text{step}\_\text{expr} > 0\) and \(\geq\) otherwise. This check essentially screens out potential infinite loops. In the third step, the compiler checks if the loop is empty, which occurs when \(\text{step}\_\text{expr}\) and \(\text{end}\_\text{expr} - \text{init}\_\text{expr}\) have different signs, or more formally,

\[
\text{step}\_\text{expr} \times (\text{end}\_\text{expr} - \text{init}\_\text{expr}) < 0
\]

Note that the check is performed only when the compiler is able to determine the sign of \(\text{end}\_\text{expr} - \text{init}\_\text{expr}\). If the loop is empty, the compiler issues a warning and ignores the \text{loop}\_\text{partition} directive.

After pre-processing, the compiler generates new bounds and step for the loop based on the distribution strategy specified in the \text{loop}\_\text{partition} directive. Figure 4.12 gives the general code template for each strategy. The code generation for CYCLIC distribution is very simple, as shown in Figure 4.12a. Each thread starts from a loop index offset that is equal to the thread index, and jumps in a step that skips over the batch of iterations executed by all threads. The code for distribution among thread blocks (Figure 4.12b) works exactly the same way. When both \text{over}\_\text{tblock} and \text{over}\_\text{thread} clauses are present (Figure 4.12c), the threads in all thread blocks are considered in a single group, and each thread’s index is constructed to be unique in the group. In all three code templates, \(<\text{num}\_\text{tblocks}>\) and \(<\text{num}\_\text{threads}>\) are the size of the dimension of the thread block and thread spaces associated with the directive, respectively. \(<\text{tblock}\_\text{idx}>\) and \(<\text{thread}\_\text{idx}>\) are the current thread’s index in these two dimensions. \(<\text{num}\_\text{tblocks}>\)
Chapter 4. The hiCUDA Compiler

and <num_threads> can be easily retrieved from the kernel context for this directive. For example, they are 8 and 2 for loop j in Figure 4.11(a). The construction of <tblock_id> and <thread_id> involves mapping the virtual thread block and thread spaces (specified by the kernel context) onto the physical ones supported by CUDA, and is described in Section 4.9.1.

![CUDA code templates for a loop_partition directive](image)

Figure 4.12: CUDA code templates for a loop_partition directive.

The code generation for BLOCK distribution, shown in Figure 4.12(d), requires the calculation of the number of iterations assigned to each thread block, which is used to
construct the value stored in `tblk_stride` (line 2):

\[
tblk\_stride = \left\lceil \frac{\text{tripcount}}{\text{num}\_tblocks} \right\rceil \times \text{step}\_expr
\]

Figure 4.13 shows the code generated to compute the loop’s trip-count (`tripcount`), which is used to replace line 1 of Figure 4.12d and 4.12e. Different code is generated depending on the sign of `step\_expr` in order to avoid integer division of negative numbers, which would occur if Figure 4.13a is used when `step\_expr` is negative. This is because different C implementations interpret integer division of negative numbers differently. Also note that the branch at line 2 (of both cases) is essentially the empty-loop check in the pre-processing stage, and will be omitted if the compiler is able to do this check. Once `tblk_stride` is determined, the starting loop index offset (`i_init`) is generated based on each thread block’s index. The ending index (`i_end`) is essentially the starting index for the next thread block less a single step. This is not true for the last thread block if the loop’s trip-count is not perfectly divisible by `num\_tblocks`, in which case the ending index must be the original `end\_expr`. This is the purpose of line 6 in the figure. Note that the compiler does not generate this line if it is able to verify the perfect-division. When `over\_thread` is present (Figure 4.12d), the code generated looks almost the same except the starting loop index and the step are amended to achieve CYCLIC distribution as described in the previous paragraph.

```
1 tripcount = 0;
2 if (end_expr >= init_expr)
3   tripcount = (end_expr - init_expr) / step_expr + 1;
5 }
```

(a) `step_expr > 0`

```
1 tripcount = 0;
2 if (end_expr <= init_expr)
3   tripcount = (init_expr - end_expr) / (-step_expr) + 1;
5 }
```

(b) `step_expr < 0`

Figure 4.13: CUDA code templates for calculating a loop’s trip-count.

Sometimes the distribution strategy specified in a `loop\_partition` directive results in at most one iteration per thread. In such cases, the compiler does not need to generate a full-fledged loop. More specifically, the compiler generates more optimized code when the
loop’s trip-count is less than or equal to the total number of threads (if the directive has
the over_thread clause) or the total number of thread blocks (otherwise). Note that the
total number of threads is \(<\text{num_blocks}\times<\text{num_threads}\) if the over_tblock clause
is also present. When the condition is equality, the loop index variable \(i\) is assigned
the expression originally for \(i_{\text{init}}\) (in the unoptimized version), and the loop house-
keeping code is removed. For example, Figure 4.12a is optimized to a single assignment
\(i = \text{init_expr} + \text{thread_idx} \times \text{step_expr}\); followed by the loop body. If the condition
is “strictly less than”, a guard needs to be added to the optimized version because some
threads will not execute any loop iterations. The guard ensures that \(i \leq \text{end_expr}\).
It is important to note that optimized code is not generated if the compiler is not able
to verify the condition mentioned above, e.g. when it cannot generate a deterministic
expression for the loop’s trip-count.

This completes the description of CUDA code generation for loop partition
directives. Figure 4.11b shows the generated CUDA code for the example in Figure 4.11a.

4.8 Utilizing the Shared Memory

\(\text{hiCUDA}\) 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 this
directive more complicated than that of a global or 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 (Section 3.3.3). 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. Since a shared directive may
not be placed in the same procedure as the enclosing loops, an inter-procedural propagation
of loop partitioning information is required (detailed in Section 4.8.2). Second, 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 must be free of bank conflicts, and the reads from the global memory must be coalesced [26]. Section 4.8.1 describes the details of data transfer code generation.

Apart from the two challenges mentioned above, the rest of the translation process is straightforward. Before handling any shared directives, the compiler verifies that they are placed inside kernel regions, by performing the same checks as done for loop_partition directives (Section 4.7.1). After handling the allocation of shared memory and data transfer, the compiler redirects accesses within the scope of the directive to the corresponding shared memory variables. This process is very similar to access redirection to global or constant memory variables. In fact, the compiler delays the phase of kernel access redirection (Section 4.6) until the shared directives are handled, so that accesses can be redirected to the shared memory directly (without going through the redirection to the global memory first). Finally, no code is generated for shared remove directives because shared memory variables are statically allocated. Similar to the case of constant memory variables, the compiler employs a register-allocation algorithm to maximally reuse shared memory regions across different shared memory variables.

### 4.8.1 Allocation and Data Transfer

First and foremost, the compiler determines the array section to be allocated in the shared memory (for each thread block). It projects the section specified in the alloc clause onto the range of each enclosing loop’s index variable (referenced in the section specification) that represents concurrently executed iterations in the thread block. During this projection, the compiler only considers the loops that are associated with a loop_partition directive with an over_thread clause, because otherwise all threads in a thread block
execute the loop iterations one at a time and there is no need to do projection for this loop.

Since the over_thread clause specifies a cyclic distribution of loop iterations, it is straightforward to determine the range of iterations concurrently executed in a thread block. Consider a loop associated with directive loop_partition over_thread, the range starts from the iteration assigned to the thread with <thread_idx> = 0 and has length <num_threads>, where <thread_idx> and <num_threads> are defined in Section 4.7.2 (page 55). More specifically, the starting index of the iteration range can be expressed as $i - <thread_idx> * <step_expr>$ where $i$ is the loop index variable after the loop_partition over_thread directive is handled. In the example shown in Figure 4.14, loop $i$ is translated to for ($i = 1 + threadIdx.y; i <= 1024; i += 16$) after the partitioning, and the range of concurrent iterations is $[i - threadIdx.y, i - threadIdx.y + 15]$. Note that the compiler flips the range for negative $<step_expr>$. When $|<step_expr>| > 1$, the range is non-contiguous. For simplicity, the compiler still treats it as a contiguous range.

When the over_tblock(BLOCK) clause is used in the loop_partition directive (like loop $j$ in Figure 4.14), the range of concurrent loop iterations is exactly the same as before (i.e. with over_thread clause only). This is because each thread block is assigned a contiguous set of iterations and the loop index variable (after partitioning) is restricted to this set.

However, the situation becomes complicated when the over_tblock(CYCLIC) clause is used: the concurrent iteration range is strided and spans the entire iteration space. In this case, the compiler assumes that the range is the entire iteration space for simplicity.

After section projection with respect to concurrent loop iterations, the compiler declares a new local variable named “$s.A$” (where A is the host variable) and allocates shared memory for this variable. Since the allocation is static, the compiler first ensures that the size of each dimension is a constant. In the example (Figure 4.14), the projected
Figure 4.14: An example of shared directive handling.

Section for A is $A[i-threadIdx.y-1 : i-threadIdx.y+16][j-threadIdx.x-1 : j-threadIdx.x+16]$, so the new shared memory variable is an $18 \times 18$ matrix. Note that the allocated section is different for different thread blocks even though its size is the same. For some thread blocks, this section may not be within the one brought into the global memory (or even the original host array), even though this is not the case in the example. This is perfectly acceptable because no actual data transfer occurs so far. Similar to the case of constant memory variables, statically-allocated regions for shared memory variables are reused maximally by employing an algorithm similar to register allocation, described in Section 4.10.2.

Before generating code for data transfer, the compiler performs the similar projection on the array section in the copyin/copyout clause (or the one in alloc clause if omitted). Since this section is to be actually transferred, it must be within 1) the section allocated in the shared memory, and 2) the section brought into the global memory, for each thread block. The second check is more complicated than the first one as it involves the comparison between the union of the transferred section across all thread blocks and the section in the global memory. In the example, the union of the concurrent iterations of loop $i$ across all thread blocks is still $[i-threadIdx.y, i-threadIdx.y+15]$, and for loop $j$, a strided range that spans $[1, 1024]$, which is conservatively assumed to be $[1, 1024]$. Therefore, the union of the sections to be transferred for all thread
blocks is $A[i\text{-threadIdx.y}-1 : i\text{-threadIdx.y}+16][0:1025]$. If the first check fails, the compiler emits an error. On the other hand, the failure of the second check simply indicates that the sections to be transferred for some thread blocks fall outside the section in the global memory, and the compiler must generate code to clip the lower and upper bounds of each dimension. Sometimes the compiler is not able to perform the second check accurately, and must conservatively assume that it fails. In such cases, the programmer can disable the check (and the generation of clipping code) by including the nobndcheck option in the clause.

Now the compiler is ready to generate code of cooperative data transfer: multiple threads read data from the global memory and write them to the shared memory in parallel. To achieve optimal performance of the transfer, the reads from the global memory by a warp of threads must be coalesced and the writes to the shared memory by a half-warp of threads must be free of bank conflicts [26]. In CUDA v1.1, the number of threads in a warp and a half-warp ($W$ and $HW$) is 32 and 16 respectively, and the requirements for memory coalescing are:

- The data type $T$ is 4-, 6-, 8-byte.
- The address accessed by the first thread (in a warp) is aligned at the boundary of $32 \times \text{sizeof}(T)$.
- The addresses accessed by the remaining threads must be contiguous following the starting address, i.e. with offset $i \times \text{sizeof}(T)$ where $i$ is the index of the thread in the warp.

Note that if these requirements are fulfilled, accesses are coalesced even if some threads of the warp do not actually access memory.

---

2In CUDA v1.1, the requirements for memory coalescing are actually at the granularity of a half-warp, but the programmers are recommended to fulfill the requirements for the full warp because future versions of CUDA device may necessitate them.
The main challenge in data transfer code generation is to determine which thread transfers which element(s) of the section so that the optimization requirements for both the global and the shared memory are fulfilled. Currently the compiler cannot handle data types that are not 4/8-byte in size and emits an error in such cases, so the first requirement of memory coalescing is automatically respected. To fulfill the third requirement, the compiler simply assigns a contiguous range of \( W \) elements to each warp of threads. Each element in the range is assigned to each thread in the warp in the order of the thread index. This assignment also fulfills the free-of-bank-conflict requirement for the shared memory because consecutive elements (accessed by a half-warp) are mapped to different memory banks and the shared memory has exactly \( HW \) (i.e. 16) banks.

Therefore, the compiler must first determine the biggest sub-section to be transferred that is contiguous in both the shared and the global memory. This process is the same as the one in data transfer code generation for global/constant directives (Section 4.3.2).

In the example (Figure 4.14), the biggest contiguous sub-section is the innermost dimension, i.e. an 18-element row segment \( A[i][j-threadIdx.x-1 : j-threadIdx.x+16] \).

The only requirement left to fulfill is the second one for memory coalescing. Since the starting address of any global memory region allocated through \texttt{cudaMalloc} is aligned to 256 bytes, which is sufficient to guarantee memory coalescing, the compiler only needs to ensure that each warp starts fetching data from the closest \textit{aligned offset} before a contiguous sub-section.

Figure 4.15 shows the code template for data transfer in the general case. A contiguous sub-section (CS) determined above is partitioned into \textit{aligned} segments of \( W \) elements, as shown in Figure 4.16. Note that a portion of the first and last segment could fall outside the CS due to the mis-alignment of the CS’s starting address and the fact that the number of elements in a CS may not be a multiple of \( W \). All the segments (for the entire section to be transferred) are assigned to the warps in a cyclic fashion, resulting in loop \( i \) at line 13 in the Figure 4.15. For each thread, the compiler computes 1) the index
Figure 4.15: Code template for data transfer between the shared and the global memory.

of the warp that contains it (stored in variable `warp_id`), 2) the thread index within this warp (stored in variable `id_within_warp`) and 3) the total number of warps (stored in variable `n_warps`). `warp_id` and `n_warps` are used to determine which segment(s) are assigned to each warp, while `id_within_warp` is used to determine which element in the segment is assigned to each thread (described later). The three values are computed as follows:

\[
\text{<n_warps>} = \frac{\text{num_threads_per_tblock}}{\text{warp_size}}
\]

\[
\text{<warp_id>} = \frac{\text{thread_index_within_tblock}}{\text{warp_size}}
\]

\[
\text{<id_within_warp>} = \text{thread_index_within_tblock} \mod \text{warp_size}
\]

Currently the compiler cannot handle the case when the number of threads per thread
block is not a multiple of the warp size (i.e. 32) and emits an error in such cases. Also, the compiler tries its best to determine a clean expression for the two index variables through symbolic division. In the example (Figure 4.14), \(<\text{n\_warps}>\) is \(16 \times 16 = 256\); \(<\text{warp\_id}>\) and \(<\text{id\_within\_warp}>\) are \text{threadIdx.y} \text{ and } \text{threadIdx.x} \text{ respectively.}

![Segment-partitioning](image)

(a) \(<\text{cs\_size}> = 32K \text{ where } K = 2.\)

(b) \(<\text{cs\_size}> = 32K + 24 \text{ where } K = 1.\)

Figure 4.16: Examples of worst-case partitioning of a contiguous section into segments.

To complete loop \(i\) in Figure 4.15 the compiler also needs to compute the total number of segments (stored in variable \(<\text{n\_segments}>\)), which is the product of the number of CS’s (\(<\text{n\_cs}>\)) and the number of segments per CS (stored in variable \(<\text{n\_segments\_per\_cs}>\)).

To show how \(<\text{n\_cs}>\) is computed, let the array section in the shared memory, the one in the global memory and the one to be transferred be \(A[ss_1 : se_1][ss_2 : se_2]...[ss_n : se_n],\) \(A[gs_1 : ge_1][gs_2 : ge_2]...[gs_n : ge_n]\) and \(A[ts_1 : te_1][ts_2 : te_2]...[ts_n : te_n]\) respectively, where \(A\) is a \(D_1 \times D_2 \times \ldots \times D_n\) array. Then

\[
<\text{n\_cs}> = \prod_{d=1}^{p-1} (te_d - ts_d + 1)
\]

where \(p\) is defined in Section 4.3.2. \(<\text{n\_segments\_per\_cs}>\) is more complicated to compute as it depends on the alignment of each CS’s starting address, which could be different for different CS’s. For simplicity, the compiler considers the worst-case scenario for any CS with the number of elements \(<\text{cs\_size}> = (te_p - ts_p + 1) \times \prod_{d=p+1}^{n} D_d;\)

\[
<\text{n\_segments\_per\_cs}> = \begin{cases} 
K + 1 & \text{if } <\text{cs\_size}> = \text{warp\_size} \times K + 0/1 \\
K + 2 & \text{otherwise}
\end{cases}
\]

Figure 4.16 shows examples of segment-partitioning schemes (due to the worst-case CS
alignment) that results in the expressions above.

Once the segment to be transferred by a warp is identified (through its global index in Figure 4.15), each thread in the warp is responsible for the id\_within\_warp\’th element of the segment. The body of loop i computes the address of the element and performs the transfer if and only if it does not fall outside the CS that contains the segment. More specifically, the code first computes the offset of the parent CS’s starting address in the global memory (line 17 and 22) and aligns it to a multiple of warp\_size (line 26). This aligned offset represents the beginning of the first segment of the CS. Then the code computes the offset of the element to be loaded by the current thread (with respect to the aligned beginning of the CS) and the corresponding write offset in the shared memory region (line 30 and 36). The calculation of CS offset in the global and shared memory (line 22 and 23) uses variables gcs\_sz (scs\_sz) and gcs\_ofst (scs\_ofst), which are computed as follows:

\[
<\text{gcs\_size}> = \prod_{d=p}^{n} (ge_d - gs_d + 1) = (ge_p - gs_p + 1) \times \prod_{d=p+1}^{n} D_d
\]

\[
<\text{gcs\_offset}> = (ts_p - gs_p) \times \prod_{d=p+1}^{n} D_d
\]

\[
<\text{scs\_size}> = \prod_{d=p}^{n} (se_d - ss_d + 1) = (se_p - ss_p + 1) \times \prod_{d=p+1}^{n} D_d
\]

\[
<\text{scs\_offset}> = (ts_p - ss_p) \times \prod_{d=p+1}^{n} D_d
\]

4.8.2 Kernel Loop Context Propagation

To determine the array section to be allocated in (or transferred to) the shared memory (for each thread block), the compiler must inter-procedurally collect information about how enclosing loops are partitioned among the threads in the thread block. The propagation of loop partitioning information occurs among IK-procedures, and starts from
those directly called inside kernel regions. This process is similar to data directive propagation in reaching directives analysis (Section 4.4), and Figure 4.17 illustrates it using a simple example. Annotation is associated with a formal parameter of a procedure if it holds the value of the index variable of a kernel loop with more than one iterations being concurrently executed. mat of procedure `process_row` in the figure is an example. Its annotation contains the `loop_partition` directive associated with the loop (i) along with its kernel context (i.e. kernel region `k_example`). It is important to note that the translation of `shared` directives only needs the partitioning information of loops that are associated with a `loop_partition` directive that contains an `over_thread` clause, so only these loops are considered during the propagation.

```c
void process_row(float (*mat)[5][4], int row) {
  #pragma hicuda shared mat[row][*]
  ...
}
int main()
{
  float mat[5][4];
  int i;
  ...
  #pragma hicuda kernel k_example tblock(1) thread(5)
  #pragma hicuda loop_partition over_thread
  for (i = 0; i < 5; ++i)
    process_row(mat, i);
  #pragma hicuda kernel_end
}
```

Figure 4.17: An example of kernel loop context propagation.

After the propagation, the compiler needs to perform procedure cloning as usual. Since the translation of `shared` directives in a procedure may not need every piece of loop partitioning information propagated, the compiler first determines which annotations are actually used and back-propagates this usage information, as done in reaching directives analysis. This could result in fewer clones to be created. Once procedure cloning and the redirection of call edges are performed, the compiler re-classifies procedures (as done in kernel access redirection) because some of the IK-procedures may become N-procedures.

Now the compiler is ready to handle the allocation of shared memory and the data
transfer for each \texttt{shared} directive, as described in the previous sections. Note that the data transfer code generation and access redirection may reference variables used in the bounds of enclosing loops, which may not be local to the procedure (that contains the \texttt{shared} directive). In order for these variables to be used in the translation, the compiler must expand the list of formal parameters (and the list of corresponding actual parameters) of each procedure involved in the call chain starting from the one that contains the loop. This process is similar to the back-propagation of “useful” loop information described before, except that the information to be propagated is the referenced auxiliary variables.

### 4.9 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. This process consists of the following three steps:

1. Determine the parameters of the new kernel function.

2. Create a kernel function that contains the kernel region as the function body.

3. Replace the original kernel region with an invocation to the kernel function.

The kernel parameters consists of GPU memory variables the kernel region needs and scalar variables (that do not exist in any GPU memories) it reads. Although they have been determined in the data access analysis (Section 4.5), the scalar variables read by the kernel region may have been changed after access redirection (Section 4.6). For example, in Figure 4.18a, the redirection of access \((\ast a)[9]\) in procedure \texttt{foo} requires variable \texttt{lbnd} in the section specification \texttt{arr[lbnd:9]} even though the kernel region \texttt{k.test} does not use \texttt{lbnd} from the result of data access analysis. Therefore, another around of data flow...
analysis is performed in each K-procedure to update the scalar variables read by each kernel region in it.

```c
int bar(int x)
{
    return x + 1;
}

void foo(int (*a)[10])
{
    (*a)[9] = bar(a[9][9]);
}

int main(int argc, char **argv)
{
    int *arr[10];
    int *lbd = argc;
    if (lbd < 0) lbd = 0;
    if (lbd > 9) lbd = 9;

    #pragma hicuda global alloc arr[lbd:9]
    foo(arr);

    #pragma hicuda kernel k_test tblock(2) thread(4)
    foo(arr);

    #pragma hicuda kernel_end

    #pragma hicuda global copyout arr[lbd:9]
    #pragma hicuda global free arr

    return 0;
}
```

(a) *hicuda* code.

```c
__global__ void k_test(int *g_arr, int lbd)
{
    fooclone0(g_arr, lbd);
}

__device__ __host__ int bar(int x)
{
    return x + 1;
}

__device__ void fooclone0(int *g_arr, int lbd)
{
    g_arr[9 - lbd] = bar(g_arr[9 - lbd]);
}

void foo(int *a)
{
    a[9] = bar(a[9]);
}

int main(int argc, char **argv)
{
    int arr[10];
    int *g_arr;

    struct dim3 dimgrid;
    struct dim3 dimBlock;

    ...

    foo(arr);

    (dimgrid).x = 2;
    (dimgrid).y = 1;
    (dimgrid).z = 1;
    (dimBlock).x = 4;
    (dimBlock).y = 1;
    (dimBlock).z = 1;

    k_test <<< dimgrid, dimBlock >>>(g_arr, lbd);

    ...
}
```

(b) CUDA code.

Figure 4.18: Example of kernel outlining.

Once the parameters of a kernel region are determined, a new function is created with these parameters (and no return value). Then the kernel region is moved from the original K-procedure to the new function, and each symbol in the region is replaced with the corresponding formal parameter of the kernel function. Since CUDA requires kernel functions and functions called inside to have specific attributes, the compiler gives:

- each new kernel function the *__global__* attribute,
• each IK-procedure the __device__ attribute, and

• each IK-procedure that is also called by N-procedures the __host__ attribute.

Figure 4.18b shows the CUDA code generated from the example in Figure 4.18a. Note that procedure bar is invoked both inside and outside kernel, so it must have both __device__ and __host__ attributes.

After the kernel function is created, the original kernel region is replaced with an invocation to it. Two local variables of CUDA runtime type struct dim3 are created in the procedure that contains the kernel region, to hold the geometry of the CUDA thread block and thread spaces respectively. As shown in Figure 4.18b they are named dimGrid and dimBlock. Their values are determined by mapping the virtual thread block and thread spaces to the physical ones supported by CUDA, which will be discussed in the next sub-section. Finally, an execution configuration for the kernel function is constructed, passing the parameters and the grid geometry specification:

\[
\text{kernel\_name} \llll (\text{parameter list})
\]

The code templates for hiCUDA data directives and loop partition directives often store intermediate values in temporary variables to save computation in general. However, this could be an over-kill in simple cases, e.g. when some of the intermediate values are constants. Therefore, the compiler performs constant propagation and copy propagation in each procedure (including kernel functions) after all hiCUDA-related transformations.

### 4.9.1 Mapping the Virtual Thread Block and Thread Spaces

hiCUDA allows virtual spaces of thread blocks and threads (with arbitrary dimensionality) to be specified in a kernel directive, so the compiler must map the two spaces to the 2-D thread block space and the 3-D thread space supported by CUDA. This process produces two outputs. First, the dimension sizes of the physical spaces are determined
(as specified by Section 3.2.1), which will be used to initialize \texttt{dimGrid} and \texttt{dimBlock} mentioned previously. Second, the index in each dimension of the virtual spaces is constructed in terms of the dimension indices of the physical spaces, i.e. \texttt{blockIdx.\{x,y\}} and \texttt{threadIdx.\{x,y,z\}}. This is used in generating CUDA code for \texttt{loop.partition} directives in the kernel region (Section 4.7.2).

To formally describe the mapping process, consider a \(v\)-dimensional virtual space \(VD_1 \times VD_2 \times \ldots \times VD_v\) to be mapped to a \(d\)-dimensional physical space \(D_1 \times D_2 \times \ldots \times D_d\). In each space, the rightmost dimension is assumed to be the innermost dimension, i.e. changes the most frequently. Also let the index of each dimension of the virtual and physical spaces be \(VI_i\) \((i = 1..v)\) and \(I_j\) \((j = 1..d)\) respectively. For the thread block space, \(d = 2\), \((D_1, D_2) = (\text{gridDim.z, gridDim.y})\) and \((I_1, I_2) = (\text{blockIdx.z, blockIdx.y})\). For the thread space, \(d = 3\), \((D_1, D_2, D_3) = (\text{blockDim.z, blockDim.y, blockDim.x})\) and \((I_1, I_2, I_3) = (\text{threadIdx.z, threadIdx.y, threadIdx.x})\). There are two cases: 1) \(v < d\), and 2) \(v \geq d\). In the former case, which is simpler:

\[
D_i = \begin{cases} 
1 & \text{if } 1 \leq i \leq (d - v) \\
VD_{i+(v-d)} & \text{if } (d - v + 1) \leq i \leq d
\end{cases}
\]

\[
VI_i = I_{i+(d-v)}, \quad i = 1..v
\]

The offset \(v - d\) in the equation ensures that the inner dimensions are filled \textit{before} the outer ones if not all dimensions are used, which is required by CUDA.

The mapping is slightly more complicated when \(v \geq d\):

\[
D_i = \begin{cases} 
VD_i & \text{if } 1 \leq i \leq (d - 1) \\
\prod_{j=d}^{v} VD_j & \text{if } i = d
\end{cases}
\]
Although mathematically equivalent, the expression generated for $VI_d$ and $VI_v$ could be simplified as follows:

$$V I_d = \begin{cases} 
I_i & \text{if } 1 \leq i \leq (d - 1) \\
I_d \mod \left( \prod_{j=i}^{v} V D_j \right) & \text{if } d \leq i \leq v \\
\prod_{j=i+1}^{v} V D_j & \text{if } d \leq i \leq v
\end{cases}$$

4.10 Supporting Phases

This section describes the translation of auxiliary $hiCUDA$ directives (i.e. singular, barrier and shape), and an optimization for more efficient utilization of the shared and constant memory.

The translation of barrier and singular directives is in-place and straightforward. The former is mapped to the CUDA primitive

```c
__syncthreads();
```

For the latter, the enclosed code region is guarded by a conditional branch that only allows the thread with index 0 (in a thread block) to execute the code. The condition is:

```c
if (threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0)
```

Note that some of the equality checks do not have to be generated if the dimensionality of the virtual thread space is less than 3.

The translation of shape directives is described in the next sub-section.
4.10.1 Support of Dynamic Arrays

In order for dynamically-allocated arrays to be used in the `global` and `constant` directives, hiCUDA provides a `shape` directive so that the programmer can specify the shape of such arrays. The compiler uses the shape information to promote accesses of these arrays (in the form of pointer dereference) to regular array accesses. This promotion has two purposes. First, it facilitates the redirection of these accesses to the corresponding GPU memory variable. Second, it enables more accurate data access analysis of procedures that contain these accesses. Since a dynamically allocated array can be passed to other procedures, the compiler must propagate its shape information inter-procedurally before performing access promotion.

The inter-procedural shape propagation is very similar to the propagation in reaching directives analysis (Section 4.4). The main differences here are: 1) all procedures are involved, and 2) the annotation contains shape information (if any) for each formal parameter. Each procedure starts with no annotation. During the propagation across a function call, the compiler looks for the shape associated with each actual parameter that is a single variable, from the caller’s annotation and local `shape` directives in the caller. The shape is associated with the corresponding formal parameter of the callee only if the auxiliary scalar variables used in the shape specification are also passed as parameters. This ensures that the shape can be mapped to the callee space without modification to the callee function prototype. Once the new annotation is constructed, it is added to the callee only if it is unique among the existing annotations in the callee. Figure 4.19a shows an example of this propagation. The two annotations of procedure `foo` come from the two calls of `foo` in `main`. Note that the shape of the array variable `a` is also propagated because otherwise this information is lost in `foo`. The two annotations are considered different because the shape specifications for formal parameter `matrix` are different. After the propagation, the compiler performs cloning for each procedure with more than one annotations, as done in reaching directives analysis.
Figure 4.19: Example of shape propagation and pointer promotion.

Once all the shape information is available to each procedure, the compiler promotes pointer dereferences to array accesses. The actual implementation consists of two steps. First, the procedure code is modified to reflect the shape information. Based on the annotation, the type of each formal parameter is modified to be a pointer to an array type specified by its associated shape (if any). For each shape directive, a new local variable (whose type is a pointer to an array type specified by the shape) is created and initialized with the original pointer variable in the directive. The new variable replaces all references to the original variable in the scope of the shape directive. Second, for each indirect access *(ptr + offset) where ptr is a pointer to an array type, the compiler promotes it to an array access (*ptr)[i1][i2]...[in]. This process involves \( n - 1 \) steps of symbolic division (of the offset expression) and verifying that the remainder is within the dimension bounds in each step. Currently the compiler can only handle array types whose dimension sizes are either constants or single variables. It will leave the access in the original form if it is not able to do the promotion. Figure 4.19b shows the result
Chapter 4. The hiCUDA Compiler

of this process on the example in Figure 4.19a. Note that this code only exists in the intermediate representation and is not actually compilable if translated to C source. The type of the formal parameter matrix of procedure foo is changed and its access is promoted. On the other hand, in procedure fooclone0 (with the second annotation), the access of matrix cannot be promoted because the values of h and w are unknown. In procedure main, local variable a_da is created to replace da in the scope of its shape directive. No such variable needs to be created for an array variable like a.

After shape propagation and access promotion, the compiler proceeds with the regular analyses and transformations. Data access analysis is now able to accurately determine the section of a_da accessed by kernel k_test, but not for a because of the “messy” indirect access to matrix in procedure fooclone0. Note that this “messy” access must be formulated as an array access in order to be redirected to the global memory variable passed to fooclone0, as shown in Figure 4.19b.

The intermediate form may not be converted to C source code in a clean fashion. For example, the declaration of the formal parameter matrix in procedure foo is not correct C code. The compiler demotes the type of these pointer-to-array variables to pointer-to-element type. For example, matrix will have type float* in the generated code.

4.10.2 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 non-overlapping live ranges. It is critical to optimize this allocation scheme because 1) the size of the shared memory is very limited (64KB per SM in G80 architecture) and 2) reduction of its consumption per thread block could increase the number of active
thread blocks assigned to each SM and thus improving the program performance (by hiding global memory latency).

In the optimized scheme, a single shared memory region is allocated for each kernel, and the shared memory variable associated with each shared directive points to a particular offset of this common region. Figure 4.20 shows an example (and its corresponding CUDA code). There are three shared memory variables (for array C, D and E) to be created in kernel k_name. They share a single shared memory region k_name_smem, whose size is specified as a part of the kernel execution configuration (line 35 in Figure 4.20b). Since constant memory variables are also statically allocated, the compiler performs the same allocation optimization, in which a single constant memory region is allocated globally (line 4 in Figure 4.20b). In order to maintain data alignment, the element size (E) of the common region allocated (i.e. k_name_smem or cmem) is the largest among the element sizes of individual shared memory (or constant memory) variables. The size of the memory region for each variable is padded to a multiple of E.

The optimization problem the compiler needs to solve is to determine the placement of individual variables’ memory regions (in the common region) that minimizes the size of the common region, while still respecting the constraint that the 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 [10], similar to the graph coloring approach in register allocation. The remainder of this section describes an algorithm for solving this problem in the context of shared memory.

In a weighted interference graph, each node represents a shared memory variable and has a weight (w) that equals to the memory region size required by this variable (in multiples of E). Two nodes are connected (with an undirected edge) if and only if their live ranges overlap. In our case, the compiler determines the live range of each shared memory variable inter-procedurally and constructs a single interference graph for the
entire program. The live range of a shared memory variable is composed of two parts: 1) the local range bounded by `shared alloc` and `shared remove`, and 2) all procedures in the call chains that originate within this local range. Note that an inter-procedural propagation is required to determine all possible callee procedures (recursively) associated with a procedure. For two shared memory variables in the same procedure, their live ranges overlap if and only if their local range components overlap. If the two variables are in different procedures, their live ranges overlap if and only if one range’s parent procedure is reachable from procedure calls made within the other range. Figure 4.20 shows the shared memory interference graph for the example code in Figure 4.20.

Once the interference graph is constructed, the compiler solves the interval coloring problem defined as follows:

Given a graph $G = (V, E)$ with positive-integral vertex weights $w$, find an
assignment of an interval $I(v)$ of length $w(v)$ to each vertex $v \in V$, such that $I(u) \cap I(v) = \emptyset$ for every pair of adjacent vertices $u$ and $v$. The goal is to determine $\min\{|\bigcup_{v \in V} I(v)|\}$, which is the interval chromatic number of $G$.

It is well-known that this problem is NP-complete, and there have been quite a few polynomial-time heuristic algorithms [7, 32, 20, 39]. We adopt the one proposed by Clementson and Elphick [7] because it is applicable to a general graph while others are designed for a specific class of graphs (e.g. chordal graphs). In this algorithm, the vertices are first ordered and then interval-colored in accordance with this order. Based on the authors’ experimental results [7], we use the Largest First by chromaticity (LF1) vertex ordering, i.e. decreasing order of vertex weight. Vertices with equal weights are sub-ordered in decreasing chromatic degree order, where the chromatic degree of a vertex is the sum of weights for all adjacent vertices. According to this order, the interval of each vertex is successively assigned. In each step, the smallest starting offset of $I(v_i)$ is determined so that $I(v_i) \cap I(v_j) = \emptyset$ for all $v_j$ adjacent to $v_i$ whose interval has been assigned. To improve this heuristic, the authors also introduced an interchange procedure (in each step) that allows to change exactly one already-assigned interval in order to achieve a “better fit” of $I(v_i)$. Figure 4.21b shows the result of applying this algorithm to the example interference graph (Figure 4.21a).

Although this heuristic algorithm is simple and fast (with $O(|V|^2)$ time complexity), its sub-optimality may not be suitable for optimizing shared memory allocation because
it is crucial to conserve as much shared memory as possible. An exact interval-coloring algorithm has been proposed \[5\]. It is a branch-and-bound algorithm based on the heuristic described above. Its incorporation into the compiler is left as future work.

4.11 Limitations

Currently the compiler has limitations in a number of areas, which are described in this section.

The first and most important limitation is that it does not support aliasing and pointer arithmetic for the GPU code, and may generate incorrect code in such cases. For example, in Figure 4.22a kernel region k.name1 references array B; the current compiler will emit an error complaining that B does not exist in any GPU memories, even though it is an alias of array A which has been brought into the global memory. In Figure 4.22b the call of callee inside kernel region k.name2 takes array A at a certain offset; the current compiler is not able to perform access redirection for A within callee and simply replaces the occurrence of A with its corresponding global memory variable. We leave the correction of these issues as future work because we believe that aliasing and pointer arithmetic occur rarely in data-parallel applications.

```c
float *A, *B;
...
#pragma hicuda shape A[8][8]
#pragma hicuda global alloc A[*][*] copyin
B = A;
#pragma hicuda kernel k_name1 tblock(2) thread(2)
... = B[1];
#pragma hicuda kernel_end
#pragma hicuda global free A
```

```c
float *A;
...
#pragma hicuda shape A[8][8]
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda kernel k_name2 tblock(2) thread(2)
... callee(A + offset);
#pragma hicuda kernel_end
#pragma hicuda global free A
```

(a) Aliasing. (b) Pointer arithmetic.

Figure 4.22: Limitations of the current compiler.
putation partitioning: the compiler should warn the programmer when a `loop.partition` directive is associated with a loop that carries data dependencies. This limitation can be easily addressed by performing data dependence analysis on partitioned loops, which we also leave as future work.

Finally, the current release of the compiler prototype does not support the `texture` directive. Our experience has been that the use of texture memory is very limited, and thus we leave the implementation of the `texture` directive as future work. It should be noted that its implementation is very similar to that of the `constant` directive, and hence we expect it to be straightforward.
Chapter 5

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 [28] to generate binaries. Figure 5.1 shows the entire compilation flow. The hiCUDA compiler is built around Open64 (version 4.1) [29]. 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 Chapter 4. It uses several existing modules in Open64, such as data flow analysis, array section analysis and inter-procedural analysis framework.

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

The implementation details of this compiler can be found in Appendix A.

In order to ensure the quality of the hiCUDA compiler, we have designed a set of regression tests to cover various cases of handling the hiCUDA directives, in both intra- and inter-procedural situations. The tests are outlined in Appendix B and are included in the release of the compiler.
Using our compiler, we evaluate \textit{hiCUDA} in two aspects. One is its performance. Ideally, the compiler-generated code from a \textit{hiCUDA} program should perform as well as the hand-written CUDA code implementing the same algorithm. We used seven CUDA benchmarks to verify that, which will be detailed in the next sub-section. The other aspect of evaluation is the ease of use of \textit{hiCUDA}. We have obtained some preliminary result and feedback on the compiler in accelerating a real-world medical application – Monte Carlo simulation for Multi-Layered media (MCML), which will be discussed in Section 5.2.

\section*{5.1 Performance}

To compare the performance of \textit{hiCUDA} programs against hand-written CUDA versions, we use seven benchmarks listed in Table 5.1. For each benchmark, we start from a sequential version and insert \textit{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 3.1a), and obtained the CUDA code from NVIDIA CUDA SDK [26]. For each of the remaining benchmarks, we obtained two pairs of sequential vs. CUDA versions from the \textit{base, cuda_base, cpu} and \textit{cuda} versions of the Parboil benchmark suite respectively [13]. The \textit{cuda_base} and \textit{cuda} versions are CUDA programs ported from the sequential \textit{base} and \textit{cpu} versions respectively. Compared to the \textit{base} version, the \textit{cpu} version includes optimizations for achieving better performance in the \textit{cuda} version (compared to \textit{cuda_base}). The only Parboil benchmark that is not
selected for performance evaluation is *Petri Net Simulation* (PNS), whose CUDA versions use significantly different algorithms (particularly the random number generator) than the sequential versions.

Table 5.1: CUDA benchmarks for evaluating the hiCUDA compiler.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description (after [26, 13])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiply (MM)</td>
<td>Computes the product of two matrices.</td>
</tr>
<tr>
<td>Coulombic Potential (CP)</td>
<td>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.</td>
</tr>
<tr>
<td>Sum of Absolute Differences (SAD)</td>
<td>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.</td>
</tr>
<tr>
<td>Two Point Angular Correlation Function (TPACF)</td>
<td>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.</td>
</tr>
<tr>
<td>Rys Polynomial Equation Solver (RPES)</td>
<td>Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.</td>
</tr>
<tr>
<td>Magnetic Resonance Imaging Q (MRI-Q)</td>
<td>Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.</td>
</tr>
<tr>
<td>Magnetic Resonance Imaging FHD (MRI-FHD)</td>
<td>Computation of an image-specific matrix FHD, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.</td>
</tr>
</tbody>
</table>

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.

Figure 5.2 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 hand-written CUDA versions within ±1.3% (except noticeable performance improvement in matrix multiply). It is important to note that, in order to achieve the algorithm implemented in the CUDA versions, we did make modifications to the sequential code before inserting hiCUDA directives and to the
CUDA code generated by the hiCUDA compiler. The rest of the section describes what source code changes are made to each benchmark.

![Figure 5.2: Performance comparison between hiCUDA and CUDA programs.](image)

In the MM benchmark, the only manual modification we need to do before inserting hiCUDA directives is to strip-mine the innermost loop $k$ in the sequential code, in order to insert shared directives (as described in Section 3.1). The noticeable 4% performance improvement is purely caused by different ways of writing the same kernel: the compiler-generated version has long expressions for array access offsets while the hand-written version has common sub-expressions factored out of the enclosing loop. The CUDA compiler tool-chain (including the driver) seems to prefer the former, perhaps because it can do more aggressive optimizations that are architecture-specific. Since it is unclear what optimizations are performed in the CUDA driver, we did not further investigate what specific optimizations in this compiler are in effect.

In the CP benchmark, the base and cpu versions are identical. The modifications made to base in order to achieve the algorithm implemented in the cuda_base version include: 1) loop strip-mining and permutation for proper computation partitioning, 2) ensuring all float-point operations are single-precision (e.g. changing constant 1.0 to
1.0F). We also modified the generated CUDA code to use the CUDA-specific math instructions (such as inverse square root and 24-bit integer multiplication). This category of modifications improves the performance by \( \sim 50\% \). Further modifications (in the sequential code) needed in order to achieve the algorithm implemented in the \textit{cuda} version include: 1) off-loading computation common to each GPU thread to the host, and 2) unroll and jam (i.e. loop interchange coupled with unrolling).

In the SAD benchmark, the modifications made to the \textit{base} version include: 1) loop unrolling, 2) loop fusion, 3) induction variable elimination, and 4) data type padding (from 2-byte to 4-byte). Also, we modified the generated CUDA code to implement the use of texture memory. The \textit{cuda} version improves upon \textit{cuda\_base} by utilizing the shared memory. Modifications made to the its sequential version (\textit{cpu}) include: 1) loop permutation and strip-mining, and 2) loop unrolling. We also need to modify the generated CUDA code to pack two 2-byte array elements into one 4-byte element in order to conserve shared memory usage.

In the TPACF benchmark, the \textit{base} and \textit{cpu} versions are identical. The modifications made to \textit{base} include: 1) loop strip-mining, 2) repacking an array of structs to a struct of arrays for coalesced global memory accesses. On the other hand, a significant amount of changes needs to be made to the generated CUDA code, mainly consisting of high-level idioms: 1) histogram idiom, 2) 1-D summing idiom, and 3) an idiom for atomic writes to global memory within a warp (without using atomic instructions). In the \textit{cuda} version, the shared memory is used to cache data, which requires no extra manual modifications.

In the RPES benchmark, the \textit{base} and \textit{cpu} versions are identical. The modifications made to \textit{base} include: 1) collapsing of a four-level loop nest into a flat loop, and the corresponding data repacking, 2) function inlining (that results in 10% improvement in performance), and 3) data type padding for coalesced accesses to the global memory. Modifications made to the generated CUDA code include 1) summing idiom, 2) use of fast math functions, and 3) allocation of an array that is local to the kernel in the shared
memory (which is not supported in the current shared directive). Further optimizations done in the cuda version include 1) more aggressive math operation acceleration (CUDA-specific), and 2) the use of texture memory, both of which are applied in the generated CUDA code.

The processes of porting MRI-Q and MRI-FHD benchmarks to CUDA are very similar. The MRI-Q benchmark does not have a cuda_base version. In the MRI-FHD benchmark, both cuda_base and cuda versions are based on the cpu version. The only difference between the two is that the latter unrolls the main loop in the kernel by a factor of 4. For both benchmarks, no modification was made to the cpu version before inserting hiCUDA directives. The only modification made to the generated CUDA code was loop unrolling, in order to achieve the algorithm of the cuda version.

For all benchmarks, we used the largest input set to compare the performance of the hand-written and compiler-generated CUDA code. It is worth noting that the hand-written versions are designed to work with any input set as the number of thread blocks that execute a kernel is dynamically generated. Since the current compiler only supports a thread block space with constant dimension sizes in a kernel directive, the hiCUDA versions only work with a particular input set. Extending the compiler to support a thread block space with variable dimension sizes is expected to be straightforward and is left as future work.

In summary, the modifications to the sequential code mainly include:

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

Modifications made to the generated CUDA code mainly include:

- High-level programming idioms, such as reduction and histogram.
Chapter 5. Experimental Evaluation

- CUDA-specific math operation acceleration.
- Use of texture memory.
- Flexible usage pattern of shared memory.

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

5.2 A Case Study: MCML

To assess the usability of hiCUDA, we provided an earlier version of the prototype compiler to a medical research group at University of Toronto, in accelerating a real-world application called Monte Carlo simulation for Multi-Layered media (MCML) [38]. 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 Figure 5.3. Clearly, it is massively parallel because the millions of photons are simulated independently (except when making updates to the shared copy of the absorption array).

![Figure 5.3: High-level structure of the MCML program.](image-url)
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 GPU threads.

3. Setting up data in global and constant memories.

To perform these tasks in hiCUDA 17 directives were inserted to the sequential code (that contains the modification described above), including 2 kernel’s, 3 loop.partition’s, 2 barrier’s, 1 shape, 3 global’s, 3 constant’s and 3 shared)’s. Both the CUDA and hiCUDA versions achieved a 18X speedup on a GeForce 8800GTX card over single-thread performance on a 3-GHz Intel Xeon 5160 processor. The impression given by the research group was that hiCUDA is simpler to learn and use compared to CUDA. Note that this impression is a very informal assessment of the language usability. However, 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.

The most important feedback we received from this research group is that the limitation imposed by the compiler (at that time) that a kernel region cannot contain function calls resulted in a significant portion of the porting time being put in manual function inlining (while ensuring program correctness), which almost doubles the number of lines
in the kernel function (from 400 to 800). This was our motivation of extending the prototype compiler to provide inter-procedural support for large applications. With our latest compiler, the 17 hiCUDA directives can be inserted to the sequential code before inlining, and data passed into function calls inside the kernel region will be automatically redirected to the GPU memory.

The latest CUDA version of MCML contains several more optimizations that result in 270X speedup on a GeForce GTX280 card (with 240 streaming processors). The three main optimizations are:

- Use of atomic instructions to update a single absorption array (as opposed to privatizing the array), which results in $\sim 3X$ performance improvement.

- Using the shared memory to cache the most frequently accessed portion of the absorption array, which results in $\sim 2.5X$ performance improvement.

- Algorithmic change in reducing divergent behavior among simulations of different photons, which results in $\sim 2X$ performance improvement.

In the hiCUDA version, the last optimization was easily realized as it was applied to the sequential code. The first optimization was added to the hiCUDA version as well. However, since hiCUDA currently does not support atomic instructions, one regular assignment in the generated CUDA code had to be manually replaced with an atomic operation. This optimized hiCUDA version achieved a speedup of 72X (on the GTX280 card). Unfortunately, the second optimization cannot be applied to the hiCUDA version using the shared directive. This is because, in the optimization, the update to the absorption array $A_{rz}$ is made to either the shared memory or the global memory depending on a condition that is not known at compile-time. On the other hand, the current hiCUDA compiler always redirects this update to the shared memory.
Chapter 6

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 so that more programmers can use CUDA without giving up their favorite languages. PyCUDA [18] is a Python wrapper for the CUDA API, that simplifies object management (derived from Python) and supports meta-programming of CUDA kernels. jCUDA [11] provides a Java binding for the CUDA runtime and driver APIs. In contrast to these library-based extensions, PGI and NVIDIA recently announced that they are working in cooperation to develop the CUDA Fortran language [14] along with the compiler tool-chain, to be available as a part of the PGI Fortran compiler later this year. This group of work is different from ours in that they do not intend to make CUDA programming any easier. The programmer still writes kernel code in a separate function and uses thread index variables to partition the computation. In fact, PyCUDA requires the kernel code (written in CUDA) to be imported as a string and compiles it at run-time; jCUDA provides a method to import binary kernel code in the form of a .cubin file [28], produced by the CUDA compiler tool-chain.

The second category of related work provides high-level abstraction of CUDA pro-
gramming in terms of compiler directives, and therefore is closely related to ours. Lee et.al. [19] proposed a compiler framework for translating an OpenMP [2] 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, some of which are applied to the OpenMP code while others to the generated CUDA code. PGI has recently released a directive-based Accelerator Programming Model [15] for CPU+Accelerator systems, and the latest PGI Fortran and C compiler supports this model on CUDA-enabled NVIDIA GPUs. In this model, directives are used to identify code regions to be executed on the accelerator, specify how parallelizable loops are mapped to hardware parallelism and what data to be transferred between the host and device memories. Compared to hiCUDA both work have a clear advantage that they leverage standard APIs that many programmers are already familiar with and many existing applications are programmed in these APIs. 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. For example, in the OpenMP-to-GPGPU framework, the granularity of a thread block is set uniformly for the entire application, and there is no explicit control on how to utilize the shared memory. Creating an abstraction that closely matches the CUDA model is exactly the reason we designed a new and simpler set of directives. Further, it is unclear to us how both work provide inter-procedural 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. [34, 35] investigated the major optimization principles and concluded that optimizing a CUDA program is a non-trivial task because restrictions of hardware resources make the optimization space discontinuous and often cause the programmer to be trapped in local maxima. Instead of sweeping through the optimization space in brute-force, they developed two metrics to effectively prune this space. CUDA-lite [37], developed by Ueng et.al., uses compiler techniques to automate
some CUDA optimization patterns, with the help of programmer hints. In particular, it includes a compiler transformation that ensures coalesced accesses to the global memory by utilizing the shared memory. Both work are orthogonal 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.
Chapter 7

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 hand-written 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 if the compiler is able to provide inter-procedural support. This encourages us to share this system with the CUDA programming community. We hope that the feedback we receive will allow us to more comprehensively assess the usability 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 and high-level idioms are required to be applied before and after inserting hiCUDA directives respectively. 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 \textit{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. Third, we would like to further simplify or even eliminate some \textit{hiCUDA} directives so that the burden on the programmer is further reduced. For example, we could delegate the work of inserting \textit{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. Finally, since OpenCL supports a data parallel programming model that is very close to CUDA, we do not see any difficulty in porting the \textit{hiCUDA} language and compiler support to OpenCL.
Appendix A

Compiler Implementation

The hiCUDA compiler is built upon the Open64 research compiler [29]. This infrastructure is selected mainly because:

- It starts with a high-level (AST-based) intermediate representation (IR) and supports source-to-source translation for both C and Fortran.

- It includes relatively mature data flow framework, array section analysis and inter-procedural analysis framework, all of which operate on the high-level IR.

Although the hiCUDA compiler currently supports CUDA (for C) only, the fact that Open64 has a front-end and code generator for Fortran makes it easily extensible to support CUDA Fortran.

Figure A.1 shows the phases (in Open64) that are relevant in the implementation of the hiCUDA compiler. Apart from the C front-end (gfec) and code generator (whirl2c), the two main components are ipl and ipa_link. The former performs two tasks on each procedure in each source file: 1) various global optimizations (like constant/copy propagation), implemented in module wopt.so, and 2) collection of local procedure-wise information, such as prototype, callsites and array access summary, implemented in module ipl.so. The latter component (ipa_link) takes the intra-procedural information passed
from *ipl* (on a per-file basis) and builds a merged symbol table and a call graph. It then performs various inter-procedural (IP) analyses such as IP constant propagation and dead procedure elimination. All of these tasks are implemented in module *ipa.so*. The *hiCUDA* compiler extends the IR to support CUDA-specific features (like kernel execution configuration and GPU data attributes), *gfec* to support *hiCUDA* pragma syntax, and *whirl2c* to emit CUDA code. All modules modified are shown in grey in Figure A.1. The rest of the chapter focuses on the extensions made to *ipa_link* (in modules *ipa.so* and *ipl.so*).

Table A.1 lists the ordering of phases performed in *ipa.so*. Those highlighted are already implemented in Open64: P1 and P4 in *ipa.so*; P3 in *ipl.so*; P15 in *wopt.so*. Although the major phase ordering roughly follows the sections in Chapter 4, there are several important notes to make:

- In order to benefit from pointer-based array access promotion (*P2*), data access summary of each procedure must be determined again (*P3*) before performing IP array section analysis. Therefore, the module *ipa.so* now makes backward invocations to functions in *ipl.so*.

- Kernel data access summary relies on the data flow framework and array section analysis available in *ipl*, so it is built as an extension to module *ipl.so*. 

Figure A.1: Major components in the *hiCUDA* compiler.
Table A.1: Phase ordering in `ipa_link`.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Build merged symbol table and call graph</td>
</tr>
<tr>
<td>P2</td>
<td>Shape propagation and pointer-based array access promotion (Section 4.10.1)</td>
</tr>
<tr>
<td>P3</td>
<td>Redo intra-procedural data access summary</td>
</tr>
<tr>
<td>P4</td>
<td>IP array section analysis</td>
</tr>
<tr>
<td>P5</td>
<td>Kernel region validation and procedure classification (Section 4.2)</td>
</tr>
<tr>
<td>P6</td>
<td>Translation of <code>global</code> and <code>constant</code> directives (Section 4.3, 4.10.2)</td>
</tr>
<tr>
<td>P7.1</td>
<td>Reaching directives analysis: propagation of <code>global</code> and <code>constant</code> directives (Section 4.4)</td>
</tr>
<tr>
<td>P8</td>
<td>Determine kernel data access summary and match accesses with reaching data directives (Section 4.5)</td>
</tr>
<tr>
<td>P7.2</td>
<td>Reaching directives analysis: expansion of procedure prototypes to propagate non-local GPU data to kernel regions that consume them (Section 4.4)</td>
</tr>
<tr>
<td>P9</td>
<td>Kernel context propagation (Section 4.7.1)</td>
</tr>
<tr>
<td>P10</td>
<td>Kernel loop context propagation (Section 4.8.2)</td>
</tr>
<tr>
<td>P11</td>
<td>Translation of <code>loop_partition</code> directives (Section 4.7.2)</td>
</tr>
<tr>
<td>P12</td>
<td>Translation of <code>shared</code> directives (Section 4.8.1, 4.10.2)</td>
</tr>
<tr>
<td>P13</td>
<td>Kernel access redirection (Section 4.6)</td>
</tr>
<tr>
<td>P14</td>
<td>Kernel outlining (Section 4.9)</td>
</tr>
<tr>
<td>P15</td>
<td>Intra-procedural constant and copy propagation</td>
</tr>
</tbody>
</table>

- Summarizing data accesses made by kernel regions and matching them with reaching directives (P8) are essentially performed as a part of the reaching directives analysis (P7), because only the GPU variables associated with “used” data directives will be actually propagated to each kernel region.

- The IP analyses for handling both the `loop_partition` and `shared` directives (P8 and P9) occur before the actual directive translation (P10 and P11) because kernel loop context propagation requires the presence of `loop_partition` directives.

- Kernel access redirection (P13) is delayed until the `shared` directives are handled because the redirection to a shared memory variable is very similar to the redirection to a global/constant memory variable and can be processed together.
Appendix B

Compiler Release

The release of the hiCUDA language and compiler is available at:

http://www.hicuda.org

The website provides the language specification, a link to download the compiler source code, and instructions on how to install and use it. We very much welcome application programmers to try the hiCUDA compiler and provide us feedbacks so that we can evolve the language design.

The current hiCUDA compiler has been tested on Linux only. In order to ensure its quality, we include a suite of 30 regression tests to cover various corner cases in each phase. Table B.1 shows a breakdown of test coverage.

<table>
<thead>
<tr>
<th>Category</th>
<th># of Tests</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTA</td>
<td>6</td>
<td>Allocation and data transfer for global and constant directives of statically and dynamically allocated variables; optimized constant memory allocation.</td>
</tr>
<tr>
<td>HTB</td>
<td>4</td>
<td>Translation of the loop_partition directives.</td>
</tr>
<tr>
<td>HTC</td>
<td>6</td>
<td>Allocation and data transfer for shared directives of statically and dynamically allocated variables; optimized shared memory allocation.</td>
</tr>
<tr>
<td>HTD</td>
<td>10</td>
<td>Inter-procedural reaching directives analysis, kernel data access analysis, kernel access redirection and kernel outlining.</td>
</tr>
<tr>
<td>HTE</td>
<td>4</td>
<td>Kernel context propagation, kernel loop context propagation.</td>
</tr>
</tbody>
</table>
Bibliography


[18] Andreas Klockner. Pycuda v0.94beta documentation. \url{http://documen.tician.de/pycuda/}.


[29] Open64 research compiler. [http://open64.sourceforge.net](http://open64.sourceforge.net).


