Linking Scheme code to data-parallel CUDA-C code

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
AKM Rasheduzzamn Chowdhury

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

In *Compute Unified Device Architecture (CUDA)*, programmers must manage memory operations, synchronization, and utility functions of *Central Processing Unit* programs that control and issue data-parallel general purpose programs running on a *Graphics Processing Unit* (GPU). NVIDIA Corporation developed the CUDA framework to enable and develop data-parallel programs for GPUs to accelerate scientific and engineering applications by providing a language extension of C called CUDA-C. A foreign-function interface comprised of Scheme and CUDA-C constructs extends the Gambit Scheme compiler and enables linking of Scheme and data-parallel CUDA-C code to support high-performance parallel computation with reasonably low overhead in runtime. We provide six test cases — implemented both in Scheme and CUDA-C — in order to evaluate performance of our implementation in Gambit and to show 0–35% overhead in the usual case. Our work enables Scheme programmers to develop expressive programs that control and issue data-parallel programs running on GPUs, while also reducing hands-on memory management.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Christopher Dutchyn for the critical comments, remarks and engagement through the learning process of this master thesis. I thank my fellow labmates in Software Research Lab. Last but not the least, I would like to thank my family for supporting me throughout this study.

To my Wife

Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Machine
PRAM	Parallel Random Access Machine
CRCW	Concurrent Read Concurrent Write
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
OOP	Object-Oriented Programming
GPGPU	General Purpose Computations on GPU
SIMD	Single Instruction Multiple Data
ESD	External Symbol Dictionary
RLD	Relocation Dictionary
CESD	Composite External Symbol Dictionary
EOM	End-Of-Module
API	Application Programming Interfaces
3DFD	3D Finite Difference Computing
KB	Kilobyte
GB	Gigabyte
DDR	Double Data Rate
GDDR	Graphics Double Data Rate
PCI	Peripheral Component Interconnect
ms	Millisecond
exp	Expression
fv	Free Variable
bfv	Bound Free Variable
env	Environment
var	Variable
arg	Argument
arg-list	Argument List
dir	Directory
parms	Parameters
cfg	Configuration

Chapter 1

Introduction

Graphics processing units (GPUs) are massively parallel multicore processors. They can be used for extensive data-parallel computations with a large degree of *Single Instruction Multiple Data (SIMD)* parallelism. GPUs with a *Central Processing Unit (CPU)* to accelerate general purpose computations on GPUs are known as the *general purpose computations on GPUs (GPGPU)* [14] computing. GPGPU computing is becoming popular especially for scientific and engineering applications [48, 58, 65, 66, 70] because of its data-parallel capabilities. A program that has some data-parallel properties is well suited for GPU computation. In general, any program that runs on multiple GPU processors can perform data-parallel operations on shared data elements.

The *Random Access Machine(RAM)* is a convenient model of a sequential computer. It consists of a *Central Processing Unit (CPU)* that executes user-defined programs and a *random-access memory*. The random-access memory has a read-only memory-space for input data, a write-only memory-space for output data. The user-defined programs are also stored in the random-access memory in a RAM processor. The instruction set for a RAM processor typically includes arithmetic, logic, comparison, and jump instructions [63].

The *Parallel Random Access Machine (PRAM)* is a natural generalization of a RAM processor. It consists of a bounded number of RAM processors and a shared memory. Each RAM processor can access its own random-access memory as well as the shared memory. In a PRAM model, the RAMs execute the following steps synchronously: they (a) read from the shared memory, (b) perform a local computation, and (c) write to the shared memory [63]. There are four variations of this model depending on how multiple RAM processors are permitted to access the same memory location at the same time.

1. Exclusive Read/Exclusive Write (EREW): No two processors are allowed to read or write the same shared memory cell simultaneously.

2. Concurrent Read/Concurrent Write (CRCW): Multiple processors are allowed to read and write the same shared memory cell simultaneously.

3. Concurrent Read/Exclusive Write (CREW): Simultaneous reads of the same memory cell are allowed, but no two processors can write the same shared memory cell simultaneously.

4. Exclusive Read/Concurrent Write (ERCW): Simultaneous reads of the same memory cell are not allowed, but multiple processors can write the same shared memory cell simultaneously.

The GPGPU computing model exhibits the properties of a PRAM processor. In GPGPU computing, multiple GPU processors can read from a global shared memory. Then, they run the same program in parallel. Finally, they write the results to the same global shared memory. Moreover, multiple GPU processors can read and write shared-memory cells simultaneously. Therefore, the GPGPU model specifically falls into the category of CRCW PRAM processor. In addition, there are some read-only memory spaces in GPUs where multiple GPU processors can read simultaneously. So, GPGPU can also be described as *Concurrent Read(CR)* model.

In parallel computing, a task might be divided into completely independent parts. In this case, an independent part can be executed on separate processors in parallel. This type of computation is called *embarrassingly parallel* and requires no communication [22]. GPGPU computing is embarrassingly parallel since the same data-parallel program runs independently on multiple GPU processors simultaneously on shared data, without any communication. However, initial data must be distributed among the GPU processors and final results also must be collected. So, GPGPU computing requires a small communication among the GPU processors before and after the data-parallel computation. Moreover, some GPGPU computations do require a small communication among the running GPU processors to synchronize data-parallel computations. Therefore, we describe GPGPU computing as a *nearly embarrassingly parallel* computation.

NVIDIA Corporation developed a software framework for GPGPU computations known as the *Compute Unified Device Architecture(CUDA)* [11]. CUDA provides an extension to the C programming language, also known as the CUDA-C to develop data-parallel programs only for the NVIDIA GPUs. CUDA-C provides new language constructs in C to issue and manage general purpose computations on GPUs. It also provides a wide range of library functions for managing data-parallel programs.

Similarly, *Open Computing Language (OpenCL)* is another standardized framework for GPGPU computing [5]. It is the first truly-open and royalty-free programming standard for GPGPU computing and developed by an open standard-committee with representatives from major technology-vendors and managed by Khronos Group. OpenCL provides a language extension to C for developing data-parallel programs for a variety of hardware. This gives GPU programmers portable and efficient access to diverse processing platforms such as CUDA-enabled GPUs, some ATI GPUs, multi-core CPUs from Intel and AMD, and other processors such as the Cell Broadband Engine.

OpenCL and CUDA-C are low-level imperative languages to write data-parallel programs that run on GPUs.

A programming language is a system of formal notations for expressing algorithms [67]. Programs must be readable for both machines and human beings. In [17, p. xvii], authors state

> *Programs must be written for people to read, and incidentally for machines to execute.*

Many software development projects include a team of programmers who often need to read and understand programs written by others. Hence, programs should be easy to understand and debug and their programming language should support that. Programming languages that are expressive can help programmers to write clear and concise programs that are easy to understand and debug [46]. Therefore, expressive power of a programming language is important to make readable, understandable, and maintainable programs.

Language expressiveness is an important criteria in language design. However, there are always penalties associated with expressiveness for a language. The language may have cryptic notations demanding much attention from programmers. The compiler of the language may produce inefficient code. Or, it may run only on a single specific computer. Moreover, there might be no compiler at all for that language, only an interpreter.

In [54, p. 12], Tatsuru Matsushita states

> *Despite all these disadvantages, however, the designer might claim that his or her language has enormous expressive power.*

The reason is, there is no widely accepted definition and formal framework to formalize, measure or compare expressive power of programming languages. As Matthias Fellesian states in [36, p. 1]:

> *The literature on programming languages contains an abundance of informal claims on the relative expressive power of programming languages.*

Further, he also develops a formal framework based on some of the widely-held beliefs about the expressive power of several extensions of high-level functional languages.

Programs written in low-level languages are difficult to understand because programmers still need to give low-level machine details to write programs. Moreover, Tatsuru Matsushita also notes that programs written in low-level languages do not describe problems, but describe solutions to the problems. However, high-level languages hide those details and provide abstractions for the programmers. In [67], Watt notes that

> *High-level languages are so called because they allow algorithms to be expressed in terms that are closer to the way in which we conceptualize these algorithms in our heads.*

Functional programming languages hide those low-level details and provide more abstractions to the programmers. Functional programming languages actually do more than that. The special characteristics and advantages of functional programming languages are often summed up more or less as follows in terms of side effects [18, 45, 61]. Programs written in functional programming languages usually have no side effect at all. A function call can have no effect other than simply to its own computation. Therefore, programs written in functional programming languages are less error-prone [46]. No side effect also makes order of execution irrelevant; and, because of this, an expression can be evaluated at any time. Therefore, it is always safe to run computations written in functional languages in parallel. In programs written in functional languages, variables can be replaced by their values and vice versa. This characteristics is called *referential transparency*. It helps programs to be more tractable mathematically. John Hughes reports that one of the major benefit

of functional programming languages is that program size is shorter compared to imperative languages. John Hughes further states in [46, p. 2]

> *A functional programmer is an order of magnitude more productive than his or her conventional counterpart, because functional programs are an order of magnitude shorter.*

However, the same analysis shows that programs written in imperative languages contain 90% assignment statements. This huge number of assignment statements are often irritating to the programmers. The reason is programmers need to track the meaning of every variable and the order of every assignment. These assignments can be omitted in programs written in functional languages; and, this makes programs shorter compared to their imperative counterparts. Moreover, side effects from assignment statements can change the states of variables unintentionally by the programmers defined in different parts of a program. Often this can create bugs in programs.

As a result, programs written in high-level functional languages are easier to read, understand, and maintain because of their expressiveness. However, in real-world systems, people often use less-expressive languages. Programs written in less-expressive languages often show better performance compared to their high-level counterparts. There are also tremendous amount of legacy code in imperative languages such as C, C++; and, they are not disappearing soon. Operating systems are also written in C, C++, or other similar imperative languages because imperative languages provide such low-level machine details that are required to develop an operating system. Therefore, it is easier to interact with operating system *Application Programming Interfaces (APIs)* with C, C++, or other similar imperative-style programming languages.

Moreover, people often argue that programs written in functional languages are difficult to understand. The reason is because people are unfamiliar with the alternatives to assignments jumps; and, functional programs usually contain few constructs indicating assignments or jumps. In addition, programmers often do not think about programs in terms of mathematical functions. Once they start thinking about programs as functions or predicates, programs written in functional languages become more clear and concise than their imperative counterparts [54].

Programs written in functional languages usually do not contain side effects; and, this makes order of execution irrelevant. Therefore, it is always safe to run computations in parallel. Moreover, parallel computation is fundamentally about avoiding side-effect conflicts [43]; and, functional programming languages can clearly help with this regard. Bob Harper states that [42, p. 201]

> *The only thing that works for parallel programming is functional programming.*

In parallel programs, encapsulation and isolation of both task and memory is the key to achieve good parallel performance. Functional programming languages can naturally provide these requirements for parallel computations. For this reason functional languages such as Erlang [21], Haskell [49] and Scala [56] are becoming popular for parallel computing.

Modeling of real-world objects is the backbone of object-oriented paradigm; and, real-world objects are often parallel. Therefore, parallel computation is also available in *object-oriented programming (OOP)* lan-

guages such as Mentat [24], Sloop [40], Java [15], Oberon [41]. OOP languages can provide better reusability for parallel applications through the mechanism of inheritance and delegation. OOP languages support high-level of abstraction through the separation of services from implementation that helps with portability for parallel applications. In addition, parallel applications can be consistently and naturally developed through object-oriented analysis, design, and programming [59].

Many experimental OOP languages for parallel computations are extensions of C++[24, 40]. Note that all these languages have C style syntax. Moreover, they are actually imperative-style languages with object-oriented features. Therefore, programs written in those languages contain side effects. In addition, these parallel extensions can be large and complex. Therefore, they are not easy to learn, use, and implement efficiently [59]. Interpreted OOP languages such as early Java [15], Self [29], Smalltalk [69], etc, do not provide high runtime-efficiency which is undoubtedly important for high-performance data-parallel computations.

Scheme is a mostly functional programming language [31, 35, 60, 62]. Programs developed in Scheme are clear and concise [38]. Therefore, Scheme programs are more likely to be easy to read, understand, and maintain because of their expressiveness. In [36], Matthias Fellesian explains expressive power of Scheme by comparing two equivalent Scheme programs using his framework. Earlier in this chapter we discussed that programs written in functional languages usually do not contain side effects; and, because of this, it is always safe to run computations in parallel. Scheme is a popular research language and currently no implementation of Scheme has GPGPU support. Therefore, data-parallel computing-facility on GPUs in Scheme is also desirable. Moreover, expressive power of Scheme can give clear and concise data-parallel programs for GPUs that are easy to read, understand, and maintain. As a a first step towards that goal, we explore connecting Scheme programs to manage GPU executions.

Gambit [33] is an implementation of Scheme programming language. Scheme is usually interpreted, but Gambit compiles Scheme code to plain C code and also provides some foreign-function interfaces to link Scheme code to C code. There are a number of research projects that generate low-level GPU code from high-level functional languages [28, 51, 53, 64]. Our work is in the same spirit: we extend Gambit Scheme compiler to link Scheme code to data-parallel GPU code written in CUDA-C that runs on a GPU. Our work also enables Scheme programmers to develop expressive programs that control and issue data-parallel programs for GPUs. Another important long-term objective is data-parallel programs should have sufficiently clean semantic properties to make it possible to write simple yet robust code on top of it [38]. Scheme surely can help with this regard.

We already discussed that OpenCL supports different GPU processor architectures. However, not all OpenCL drivers are mature. Only AMD's and NVIDIA's drivers are reliable. Other drivers such as those from Intel and IBM are not mature enough to get reliable and consistent results. Unlike OpenCL, CUDA has support only for the NVIDIA's GPUs and NVIDIA's GPU drivers are mature and reliable. In addition, OpenCL does not have a centralized library-package like CUDA. Moreover, CUDA is well marketed and also provides more built-in functions and features compared to OpenCL. Hence, NVIDIA's CUDA is a prime

target for researching GPGPU computation in Scheme.

There has been a fair amount of work on performance comparison between CUDA and OpenCL. Much of this work shows a performance loss in OpenCL compared to CUDA [32, 68]. Moreover, OpenCL requires much hardware specific fine-tuning in order to get peak performance [30, 52]. This often varies from one GPU processor architecture to another. Therefore, programmers must apply different optimization techniques for different GPU processor architectures. All of these reasons motivate us to use CUDA instead of OpenCL for this research.

In order to link a Scheme program to data-parallel programs developed in CUDA-C, our implementation generates a foreign-function interface from the skeletons of those data-parallel programs defined in Scheme. We also provide some special constructs in Scheme for GPU in our implementation.

1.1 Motivation

CUDA-C has special constructs to specify data-parallel programs that run on GPUs. In CUDA, programmers must manage memory operations, synchronization and utility functions in a CPU program that controls and issues data-parallel general purpose programs running on GPUs. Library functions in CUDA are often faceted and programmers are frequently required to provide multiple library functions to perform a task.

In this thesis, our task was to link a Scheme program to data-parallel CUDA-C programs by generating foreign-function interfaces from the skeletons of those data-parallel programs defined in Scheme. The generated foreign-function interfaces reduce hands-on memory-management for programmers with a reasonable overhead in execution time. Our implementation allow Scheme programmers to develop expressive programs that control and issue data-parallel computations on GPUs. We also provide some useful library functions in Scheme to manage GPUs. Our work enables Scheme programmers to write clear and concise host programs with sufficiently clean semantic properties that are easy to read, understand, and maintain.

Currently, our implementation in Gambit only works for one-dimensional arrays but it would be convenient for programmers to develop data-parallel programs for two-dimensional arrays as well. Note that two-dimensional arrays are representations of one-dimensional arrays that make it easy for programmers to develop and understand code. Moreover, a program developed with two-dimensional arrays can also be developed for the one-dimensional representations of those arrays. Therefore, it would not be an extensive problem for programmers to develop CUDA-C kernels with one-dimensional arrays.

1.2 Thesis

A foreign-function interface comprised of Scheme and CUDA-C constructs extending the Gambit Scheme compiler enables linking Scheme and data-parallel CUDA-C code to support high-performance parallel computation with reasonably low overhead in runtime.

1.3 Contributions

This research extends Gambit Scheme compiler that links Scheme programs to data-parallel CUDA-C programs. Our work allows programmers to develop expressive programs in Scheme that can issue and control data-parallel CUDA-C programs that run on GPUs. This research also hides many details implementation of memory management in CUDA-C for data-parallel programs. This work also enables synchronization of data-parallel programs in Scheme. Furthermore, the addition of some utility library functions in Scheme for Gambit makes GPU management more convenient. The specific contributions to Gambit Scheme compiler are:

1. Extending Gambit with some special constructs enables GPU computation in Scheme. For now, these constructs can be used to link Scheme code to data-parallel CUDA-C code by generating foreign-function interfaces from the skeletons of those data-parallel programs defined in Scheme. In future, these special constructs can also be used for compiling the whole body of those skeletons to develop data-parallel program in Scheme.

2. Adding utility library functions for:

   (a) Querying GPUs

   (b) Managing executions in GPUs

   (c) Timing executions for data-parallel programs

   (d) Managing versions

1.4 Outline

This thesis is organized as follows:

- First, we introduce background materials related to this thesis in Chapter 2. This includes details of GPU programming models and specifications of CUDA-C programming language. We also give a description of the subprograms linking strategy. Next, we provide a description of C-interface in Gambit that links Scheme code to C code. We also discuss the usage of Gambit Scheme compiler for compiling Scheme code linked to C code to an executable.

- Next, we move on to design chapter of our implementation in Chapter 3. We describe the essential parts of a foreign-function interface that can link a Scheme program to a data-parallel CUDA-C program by using an example. We also describe an example `makefile` that can link a Scheme program to a data-parallel CUDA-C program through the foreign-function interface and builds a GPU executable.

- Chapter 4 illustrates the implementation of our thesis in Gambit. We describe how to generate the foreign-function interface from a special construct in Scheme that defines a skeleton for a data-parallel

program. We describe our implemented constructs in Scheme for GPUs. We also provide a template `makefile` that can manage the generation of a foreign-function interface from a skeleton for a data-parallel program and build a GPU executable. We also describe the intermediate steps for building a GPU executable. In this chapter, we also discuss our implemented library functions in Scheme for GPUs.

- In Chapter 5, we evaluate our implementation in Gambit by comparing the performance of test cases implemented both in Scheme and CUDA-C. These test cases cover various language constructs of our implementation in Gambit and CUDA-C.

- In Chapter 6, we summarize our work and provide some scopes for future research.

Chapter 2

Background

At present, Graphics processor Units (GPUs) can be used to run general-purpose data-parallel programs to accelerate scientific and engineering applications. Programmable GPUs supported by a CPU can run extensive data-parallel programs. These data-parallel programs are usually loaded into GPUs from a CPU, and execute data-parallel computations on shared data elements.

NVIDIA Corporation developed the *Compute Unified Device Architecture (CUDA)*[11] framework to enable and develop data-parallel programs for GPUs. This accelerates scientific and engineering applications by providing a language extension of C called CUDA-C. The CUDA framework consists of hardware and software architectures that can issue and manage computations on GPUs.

In this thesis, we link Scheme code in Gambit system[33] — an implementation of Scheme programming language — to data-parallel programs for GPUs developed in CUDA-C by generating a foreign-function interface. Therefore, we begin this chapter by reviewing the GPU programming model. First we describe the compute capability of a GPU device, followed by the concurrency basics of GPUs. Then we give a review of the GPU memory hierarchy. Next, we give details of the C programming language extensions for CUDA.

In this chapter, we also review the Gambit system and discuss the working strategy of the Gambit linker. Next, we describe issues for data type mapping from Scheme to C. We also describe the special constructs provided by Gambit to interface Scheme code with C code. Finally, we close this chapter by giving an overview of the usages of Gambit Scheme compiler.

2.1 Programmer's model of an NVIDIA GPU

A Graphics Processing Unit (GPU) is a highly parallel, multi-threaded, multi-core processor. Its data-parallel capabilities along with very high memory bandwidth can be used in real time, high-definition 3D applications.

GPUs with a CPU to accelerate general purpose scientific computations on GPUs are known as the General Purpose Computations on GPUs (GPGPU). More specifically, the same program is running on the different GPU processors on the shared data elements in parallel, and the data-parallel program distributes data elements to parallel processing threads.

The GPU is viewed as compute *device* capable of running thousands of threads in parallel and assisting a CPU or *host*. The data-parallel portions of an application are off-loaded onto a *device* from a host. These

data-parallel or compute-extensive portions can be isolated into a special function called a *kernel*.

Both a host and a device maintain their own logical memory, known as *host memory* and *device memory*, respectively. Data can be copied from one memory to another through *Application Program Interface (API)* calls. In this section we describe the data-parallel computation capabilities of devices. Next, we describe the parallelism through thread and block organization. We also describe the different device memory spaces. Finally, we review the execution model of a GPU.

2.1.1 Compute capability of a device

A GPU application needs to adjust according to the data-parallel capability of an underlying device. The features of a device defines its data-parallel capability, also known as its *compute capability*. Devices with different core architectures may have the same compute capability [8]. The compute capability of a device is specified by a major revision number and a minor revision number. Devices with the same major revision number will have the same core architecture, whereas a minor revision number specifies an incremental improvement of a particular device's architecture.

Table 2.1: Device specifications for different compute capabilities

	Compute Capability							
Specifications	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Maximum dimensionality of a grid	2				3			
Maximum dimensionality of a thread block	3							
Maximum x-,y- and z-dimension of a grid	65535						2^{31} - 1	
Maximum x- or y-dimension of a thread block	512				1024			
Maximum z-dimension of a thread block	64							
Maximum number of threads per block	512				1024			
Maximum number of 32-bit registers per thread	128				63			255
Maximum amount of shared memory per microprocessor	16 KB				48 KB			
Double-precision floating-point operations	No				Yes			

It is important to know the compute capability of a device for which a data-parallel program is being developed because compiled device code must be suited to the data-parallel capabilities of an underlying device. For example, CUDA devices with a compute capability of 2.0 or greater only allow three-dimensional thread blocks in a grid. Therefore, a CUDA kernel programmed to run on a three-dimensional thread block cannot launch in devices with a compute capability of 1.x.

In Table 2.1 we provide device specifications based on different compute capabilities. We will learn about these device specifications throughout this chapter.

2.1.2 Basic processing on GPU through a thread

A GPU is a multiprocessor made up of a group of *stream processors*. Each stream processor is known as a *GPU core*.

A thread running on a GPU core is the basic processing element of a GPU. We refer to this thread as a *GPU thread*. A GPU thread is different compared to a conventional *CPU thread*. A conventional CPU thread is heavyweight. However, a GPU thread is extremely lightweight. Context switching (storing or restoring state of a thread or process) is a computationally expensive operation for a GPU but is not a costly operation for a GPU.

In CUDA, GPU threads are usually grouped together and run the same kernel in parallel. Each thread in a group has a unique *thread ID*. A thread ID can be a one-component, two-component, or three-component entity. In the Figure 2.1, each thread is represented by a red square with a two-component thread ID.

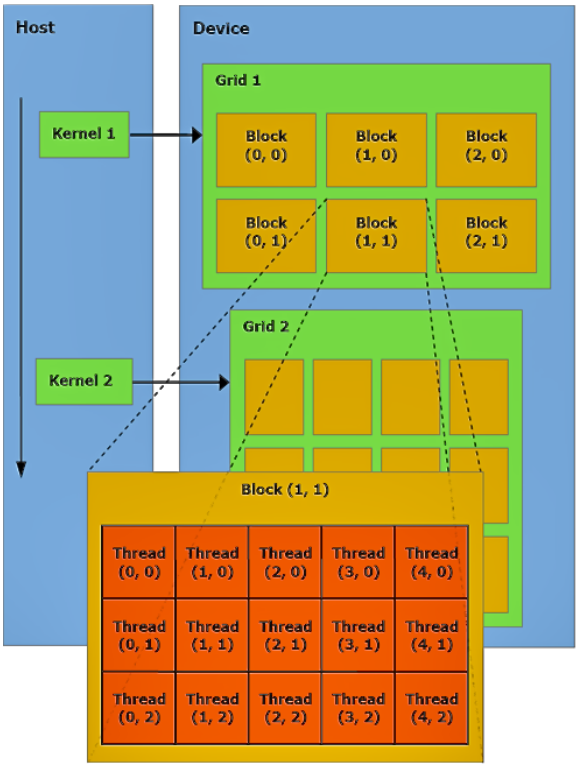


Figure 2.1: Organization of grid and thread blocks (© NVIDIA Corporation, used with permission)

2.1.3 Parallelism through thread batching

In a GPU, a batch of threads running the same kernel is known as a *thread block*. These blocks can cooperate by sharing data through fast *shared memory*, described in section 2.1.5. All threads in a thread block running the same kernel can be synchronized by synchronization points mentioned in that kernel code.

Every thread block has a unique *block ID*. How threads in a thread block are organized is displayed in Figure 2.1, where thread blocks are represented by yellow rectangles. Threads in a thread block can also be organized as a two- or three-dimensional array depending on the compute capability of device architecture. In that case, each thread can be identified using a two- or three-component index.

Both thread ID and block ID can be used for addressing a location in device memory. Many applications use complex addressing using both thread IDs and block IDs to get the maximum parallelism.

2.1.4 Concurrency through a grid of blocks

A thread block has a limit on the maximum number of threads it contains. This number depends on the compute capability of a device's architecture. To overcome this, a group of thread blocks with the same dimension are organized into a *grid*, as shown in Figure 2.1. A grid of thread blocks can be launched together so that the total number of threads launching the same kernel is much larger. A grid of blocks can also be a two- or three-dimensional array of arbitrary size. In that case, each block can be indexed using a two- or three-component index, as described in section 2.2.1.

Threads from different thread blocks in the same grid cannot be synchronized with each other. A device may run all thread blocks in a grid sequentially for its limited parallel capabilities, in parallel if it has many parallel capabilities, or in a random order.

In Figure 2.1, the host and device are represented by the blue rectangles. In the host, two kernels (kernel1 and kernel2) are represented by green rectangles. In the device, two grids (grid1 and grid2) are represented by green rectangles. The arrow from kernel1 to grid1 means all the threads in grid1 run kernel1 in parallel. Similarly, the arrow from kernel2 to grid2 means all the threads in grid2 run kernel2 in parallel. Inside each grid, thread blocks are represented by yellow squares. In this example, the thread blocks are organized as a two-dimensional array and addressed using two-dimensional indices.

A detailed view of block (1,1) in grid1 is shown in Figure 2.1. Threads are organized as a two-dimensional array. Each thread is represented by a red square with a two-dimensional index.

2.1.5 A variety of memory spaces with different tradeoffs

A GPU thread running a kernel for data-parallel computation independently can only access device memory space. Device memory is divided into several different spaces, each with different features and performances. Runtime performance of a GPU application can be enhanced by the correct uses of different device memory spaces. Therefore, it is important to know the following features and performances:

- Register: A register can only be accessed through its designated thread. A thread can read or write on its own register space. It is the fastest form of on-chip memory in a device and has the lifetime of a thread. Registers are partitioned among all the resident threads running the same kernel. The number of registers per thread depends on the available resources for registers of an underlying device. In general, a small array of integers defined in a kernel is allocated to registers.

- Local Memory: A thread-local read-write memory space 150x slower than a register. It is off-chip and not cached. If an array size is larger than the total number of registers then the local memory space is utilized. Local memory shares the lifetime of its thread.

- Shared Memory: All threads organized in the same block can access an on-chip shared read-write space. It can be as fast as accessing a register when there are no bank conflicts [6] or when reading the same addresses. It has the lifetime of all the threads in the same thread block. All devices with the compute capability 2.x and greater have 48KB of available shared memory per multiprocessor.

- Global Memory: All threads for a current program have read-write access to global memory. Global memory usually resides in a device, but recent versions of CUDA can map CPU memory to device memory address space (device must support it). It is slower compared to shared memory and is not cached. Global memory has a lifetime from allocation to deallocation. Data can be directly copied from CPU memory to global memory.

- Texture Memory: A read-only memory space accessible to all threads running the same kernel. It is cached but slower than shared memory. It is initialized by a host and read by a device.

- Constant Memory: Also a cached read-only memory accessible to all threads in a grid. It can be accessed as fast as registers. Each device has a total of 64KB constant memory space. It is also initialized by a host and read by a device.

In Table 2.2, we provide features of different device memory spaces.

Table 2.2: Features of different device memory spaces

Memory	Location	Cached	Type of access	Who can access
Local	Off-chip	No	Read/Write	One thread
Shared	On-chip	N/A	Read/Write	All threads in a block
Global	Off-chip	No	Read/Write	All threads + CPU
Constant	Off-chip	Yes	Read/Write	All threads + CPU
Texture	Off-chip	Yes	Read/Write	All threads + CPU

The memory model of a device is shown in Figure 2.2. In this figure the grid is represented by a blue rectangle. Unlike Figure 2.1, threads are represented by green rectangles. Each thread block is represented by a yellow rectangle. Registers and local memory spaces are represented by red rectangles.

There are two kinds of arrows: uni-directional and bi-directional. A uni-directional arrow from a memory space to a thread means that thread has read-only access to that memory. A bi-directional arrow between a memory space and a thread means that thread has both read and write access to that memory. The arrows from each thread to registers and local memory depict that they are local to each thread. Each shared memory space is represented by a red rectangle. All threads in a same thread block have access to a same shared memory space, as depicted by the arrows from threads in the same thread blocks to the same shared memory spaces.

Global memory, constant memory, and texture memory are represented by red rectangles. Arrows from each thread in the grid to these three memories means they have access to all three memory types.

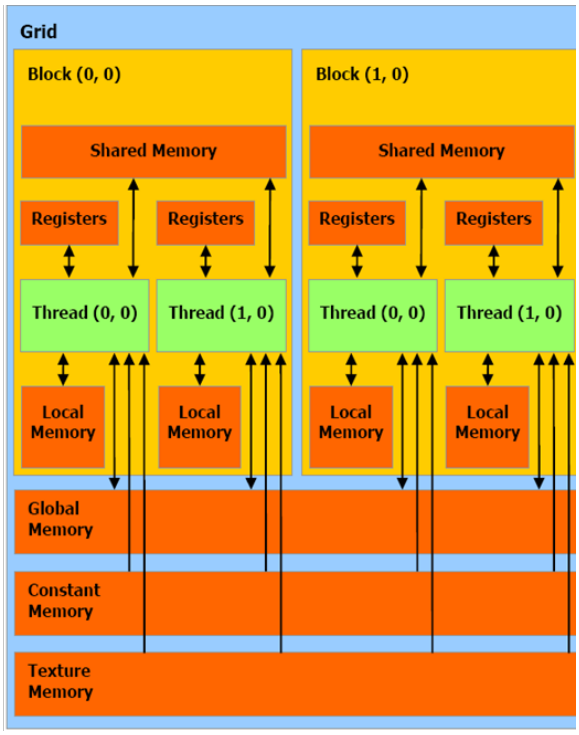


Figure 2.2: Memory model of a device (© NVIDIA Corporation, used with permission)

In this section we described various device memory spaces and features. Certain device memory spaces can also be accessible from a host. A host can read and write global, texture, and constant device memory spaces, which remains persistent across kernel launches by the same host program.

2.1.6 Execution model of a device

All thread blocks in a grid cannot be processed at the same time. A GPU scheduler is involved to schedule batches of thread blocks among the device multiprocessors. An in-device multiprocessor is responsible for processing a batch of thread blocks.

In general, multiprocessors process thread blocks one after another from a batch. The number of blocks a multiprocessor can process in a batch depends on the memory requirement of the given kernel for parallel execution. The reason is multiprocessors registers and shared memory space are split among all the threads from all thread blocks in the same batch executing the same kernel. If memory spaces cannot split atleast one thread block then a kernel launch will fail. Moreover, a thread in a thread block cannot access the shared memory of another thread block. Shared memory of multiprocessors is split among the thread blocks in the same batch. Therefore, one thread block is allocated to one multiprocessor to access shared memory quickly.

Thread blocks processed by a multiprocessor in one batch are referred to as an *active block*. Each active block splits into a number of thread groups of equal size, known as a *wrap*. All the wraps in an active block execute in a time-sliced fashion.

The order of wrap execution in a block is undefined. Their execution can be synchronized. However, the order of block execution in a grid is also undefined and there is no synchronization mechanism among the blocks. That is, unless all thread blocks finish their execution and control returns to the host program.

Multiple thread blocks can access the same locations in global memory but their execution cannot be synchronized. Therefore, block communication through global memory is not safe.

2.2 CUDA-C programming interface for GPU

Compute Unified Device Architecture (CUDA) is a new software architecture for issuing and managing data-parallel computation in GPUs. It provides CUDA-C programming language, which is an extension of C programming language [11]. CUDA-C provides programming extensions to develop data-parallel programs that run on devices. It also provides a wide range of library functions for effectively managing device memory spaces, devices, and kernel executions on GPUs. In this section, we define elaborate the constructs of CUDA-C programming language.

As a programming language, CUDA-C has four folds:

1. A new directive to call kernels,
2. Four built-in variables that specify grid and thread block dimensions, and with indices for thread block and individual thread,
3. Function type qualifiers to specify weather a function executes on device or host,
4. Variable type qualifiers to whether specify spaces in device memory.

C programs using these extensions must be compiled by the CUDA compiler, `nvcc`. The front end of `nvcc` processes CUDA source files according to C++ syntax. The host part of a CUDA program supports full C++, but the device parts only support the C subset of C++. Moreover, `nvcc` is actually a compiler driver that mimics the behaviour of the GNU compiler `gcc`. It hides the complicated details of CUDA compilation from developers. `nvcc` separates device code from host code. Then compiles device code and forwards host code to a general purpose C compiler such as `gcc`. Then, the C compiler compiles host code.

CUDA also allows the file extension `.cu` to develop data-parallel programs (kernels) for GPUs. In general, a host program can be developed in a `.c`, `.cpp`, or `.cu` file, but kernel definitions must be in a `.cu` files.

2.2.1 New directive to call a kernel

Any call to a kernel must specify an *execution configuration* that defines the dimensions of a grid and thread block that participate in data-parallel computations on a device. It can be specified by adding an expression of the form `<<< gridD, blockD, sharedM >>>` between a kernel name and kernel's argument list. For

example, a kernel is declared as:

```
__global__ void kernel (int *array);
```

and a call to this kernel appears as:

```
kernel <<< gridD, blockD, sharedM >>> (array);
```

CUDA also provides built-in vectors which can be used as runtime components. They can be used by both device and host functions. These vectors are structures derived from basic integer types and floating-point types. The components of these structures are accessible through the fields x, y, and z, respectively. CUDA provides type `dim3` for these built-in variables and it is used to define the dimensions of thread blocks and grids. Any component of type `dim3` variable that is unspecified is automatically initialized to one. For example, the execution configuration

```
<<< gridD, blockD, sharedM >>>
```

has three arguments, where:

1. *gridD* is a variable of type `dim3`. It specifies the dimensions of the grid. Here, *gridD*`.x * `*gridD*`.y` is the total number of thread blocks in the grid.
2. *blockD* is also a variable of type `dim3`. It specifies the dimensions of the thread blocks. Here, *blockD*`.x * `*blockD*`.y * `*blockD*`.z` is the total number of threads in a thread block.
3. *sharedM* is a variable of type `size_t`, and it determines a size for the dynamic shared memory at runtime.

A code example for an execution configuration is also provided in the Listing 2.1.

2.2.2 Built-in variables for dimensional information

CUDA-C also provides four built-in variables to get the dimensions of a grid and thread blocks and indices of thread blocks and threads at runtime. In a kernel definition, these variables can be used for addressing locations of an array. These four variables are as follows:

1. `gridDim` contains the dimensions of a grid. We can get the dimensions of a one-dimensional or two-dimensional grid using this variable. Its type is `dim3`.
2. `blockIdx` contains an index of a thread block in a grid. Its type is `unit3`.
3. `blockDim` contains the dimensions of a thread block. A thread block can be defined as one-dimensional, two-dimensional or three-dimensional. Its type is `dim3`.

4. `threadIdx` contains an index of a thread in a thread block. Its type is also `unit3`.

Note that all these four variables are not allowed to assign with values and take their addresses.

2.2.3 Function type qualifiers

CUDA provides function type qualifiers to enable programmers to define where a function should run. The CUDA C function type qualifiers are as follows:

- `__device__` (Device function): A `__device__` qualifier declares a function that runs on devices. Henceforth we will refer to this as a *device function*. A device function is callable only from device code running on a device. It does not support recursion and cannot reference static variables. Function pointers to a device function are not permitted.

- `__global__` (kernel): A `__global__` qualifier declares a special device function that also run on devices known as kernels. A kernel is callable only from a host program. It also does not support recursion or static variables. However, function pointers to a kernel are supported. A kernel must have `void` return type and calls to a kernel must specify its execution configuration. Host program calls to multiple kernels are asynchronous.

- `__host__` (Host function): A `__host__` qualifier declares a function that runs on a host or CPU. Henceforth, we will refer it as a *host function*. A host function is callable from a host program running only on a CPU. Since this is default, use of this qualifier is optional. One important use is using `__device__` and `__host__` together to create both a host and a device version of the same function. Note that `__host__` cannot be used with `__global__`.

In Table 2.3, we provide some features of CUDA function type qualifiers.

Table 2.3: Features of CUDA function type qualifiers

Function type qualifiers	Where it runs	What it can
`__device__`	GPU	kernels
`__global__`	GPU	host functions
`__host__`	CPU	host functions

2.2.4 Variable type qualifiers

CUDA provides variable type qualifiers to specify the location of a variable in device memory. The CUDA-C variable type qualifiers are as follows:

- `__device__`: The `__device__` qualifier declares a variable that resides in global memory. It is accessible from all the threads in a grid and from a host program through CUDA runtime library functions. A

`__device__` variable has the lifetime of an application and automatically freed when the application ends. `__device__` variables are only allowed at file scope and they cannot have `extern` linkages.

- `__constant__`: The `__constant__` qualifier declares variables that reside in constant memory. It is also accessible from all threads within a grid and from a host. It shares the lifetime of an application and is freed automatically when the application ends. `__constant__` variables cannot be written by a device; they can only be loaded by a host program through CUDA runtime library functions. `__constant__` variables are only allowed at file scope and they cannot have `extern` linkage. Constant memory size is 64 KB for devices with 1.x–3.x compute capabilities .

- `__shared__`: The `__shared__` qualifier declares variables that reside in shared memory and are accessible only to threads in a thread block. It has the lifetime of a thread block. The `__shared__` variables cannot be initialized as a part of their declaration. The size of a shared memory can be determined at runtime by passing the number of bytes as the third argument in the execution configuration of a kernel call. Sizes of shared memory for devices with 1.x–3.x compute capabilities are referred in Table 2.1.

Table 2.4 summarizes some features of CUDA variable type qualifiers.

Table 2.4: Features of CUDA variable type qualifiers

Variable type qualifiers	Memory space	What can access	Lifetime	Allocation	Scoping
`__device__`	global	all threads + host	Application	Static	file
`__constant__`	constant	all threads + host	Application	Static	file
`__shared__`	shared	all threads + host	thread block	Dynamic	global scope

2.2.5 A simple CUDA-C program

In this section we give an example of a CUDA-C program (see in Listing 2.1). In this program, all participating threads are running the same kernel performing add operations in parallel on the unique addresses of two linear arrays. Unique addresses are computed at runtime for each thread. Results are stored in an another array on the same unique addresses.

The program has two parts: host code and device code. The device code (lines 2–8) shows a CUDA-C kernel `parallel_add`. The host code (lines 10–37) shows CUDA-C code for allocating device memory, transferring vectors between host memory to device memory, a kernel call with an execution configuration, and the deallocation of device memory. On line 2 of the device code, the header of the function

```
__global__ void parallel_add (float * x, float * y, int n)
```

```
1  //The device code
2  __global__ void parallel_add(float  * x, float * y, int n){
3          //complex addressing using the CUDA C built-in variables
4          int idx= blockIdx.x * blockDim.x + threadIdx.x;
5          if(idx <  n){
6                  x[idx] = x[idx] * y[idx];
7          }
8  }
9  //The host code
10 int main(){
11         float *x_h, *y_h; //declaring host pointers
12         float *x_d, *y_d; //declaring device pointers
13         int n= 20; size_t size= n * sizeof(float);
14         //allocating vectors in the host memory
15         x_h= (float *)malloc(size); y_h= (float *)malloc(size);
16         int i;
17         for(i=0; i < n; i++){
18                 x_h[i]= (float) i; y_h[i]= (float) i;
19         }
20         // making device pointers
21         cudaMalloc((void **) $x_d, size);
22         cudaMalloc((void **) $y_d, size);
23         //copying vectors from host to device using CUDA library functions.
24         cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
25         cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);
26         int block_size= 4;
27         int num_blocks= (n + block_size - 1) / block_size;
28         //Invoke the kernel with the execution-configuration
29         parallel_add <<<num_blocks,  block_size>>> (x_d, y_d, n);
30         //copying vectors from  device to  host using CUDA library functions.
31         cudaMemcpy(x_h, x_d, size, cudaMemcpyDeviceToHost);
32         //free  host memory
33         free(x_h);
34         free(y_h);
35         //free  device memory
36         cudaFree(x_d);
37         cudaFree(y_d);
38 }
```

Listing 2.1: A simple CUDA-C program

opens with the keyword `__global__` and its return type is `void`. Therefore, `parallel_add` is a kernel. Three parameters are supplied by the host: `float* x`, `float* y`, and `int n`. the kernel definition (line 3)

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

is an example of addressing locations by participating threads to perform data-parallel computation using CUDA-C built-in variables `blockIdx.x`, `blockDim.x`, and `threadIdx.x`. Here, `blockIdx.x` gets the block ID in x-dimension of a participating thread block `blockDim.x` is the x-dimension of the participating grid and `threadId.x` is a thread ID in x-dimension of a participating thread in a thread block. Each participating thread computes a unique value during execution and assigns it to the variable `idx`. Each thread then performs the add operations (line 6),

```
x[idx] = x[idx] * y[idx];
```

on the address specified by the `idx`. Note that this add operation is determined for a thread by a conditional guard (line 5)

```
if (idx < n)
```

specifying that only threads computing values of `idx` that are less than `n` can perform the add operations.

In the host part, first we declare two host memory pointers: `float* x_h`, and `float* y_h` (lines 11 and 12). Two device memory pointers: `float *x_d`, and `float* y_d` are also declared. On line 15 two arrays pointed by the pointers `x_h`, and `y_h` are allocated in host memory and assigned values (lines 17–19). Next, pointers `x_d`, and `y_d` are converted to device memory pointers (lines 21 and 22) using the CUDA library function `cudaMalloc()`.

Next, these two vectors are copied to device memory to perform parallel computation by the participating threads (lines 24 and 25) and the CUDA library function `cudaMemcpy()` is used to perform the copy operations. On line 24,

```
cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
```

host memory pointer `x_h` is copied to device memory pointer `x_d` to transfer data to device memory. Here, `size` represents the size of both arrays. The size of both arrays must be same. `cudaMemcpyHostToDevice` is a CUDA transfer type specifying that the transfer is from host memory to the device memory. Similarly, on line 25,

```
cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);
```

host memory pointer `y_h` is copied to device memory pointer `y_d` to transfer data to device memory.

In the host part, the number of participating thread blocks must be defined. Line 26,

```
int block_size = 4;
```

defines an one-dimensional grid of four thread blocks, and line 24,

```
int num_blocks = (n + block_size - 1) / block_size;
```

defines the number of participating threads in a thread block.

On line 29, the kernel `parallel_add` is called with three arguments (`x_d`, `y_d`, and `n`) along with the execution configuration, where

```
parallel_add <<< num_blocks, block_size >>> (x_d, y_d, n);
```

Here,

```
<<<  num_blocks, block_size  >>>
```

is the execution configuration. `num_blocks` specifies the number of participating thread blocks in the grid and `block_size` specifies the number of participating threads in a thread block.

After kernel execution, control returns to the host program. The resultant vectors are then copied back to host memory (line 31)

```
cudaMemcpy(x_h, x_d, size, cudaMemcpyDeviceToHost);
```

CUDA transfer type `cudaMemcpyDeviceToHost` specifies that the transfer is from device to host.

Lastly, host and device memory spaces are freed. On line 36,

```
cudaFree(x_d);
```

and line 37,

```
cudaFree(y_d);
```

device memory pointers `x_d` and `y_d` are freed using the CUDA library `cudaFree` to release the allocated device memory spaces. Similar, host memory pointers `x_h` and `y_h` are also freed on line 33,

```
free (x_h);
```

and line 34,

```
free (y_h);
```

2.2.6 Synchronization in CUDA

In CUDA, sometimes a kernel depends on the results of other kernels [1]. Sometimes, a kernel might also depend on the results of some CPU functions. It might also happen that a CPU function depends on the results of others kernel. In this situation, kernels and CPU functions must be synchronized. This means a kernel or a CPU function that depends on the results of other kernels or CPU functions must be blocked until those prior kernels and CPU functions are finished. In order to do that, CUDA provides library function `cudaDeviceSynchronize()` [12] that can be used to synchronize kernels and CPU functions.

```
1  __global__  void kernel1 (float* A, float *B)  {
2          int i = threadIdx.x;
3          B[i] = B[i] * 2;
4          __syncthreads()
5          A[i] = B[i] + B[i];
6  }
7  __global__  void kernel2 (float* A, float *B)  {
8          int i = threadIdx.x,
9          A[i] = A[i] - 4*B[i];
10         __syncthreads();
11         B[i] = B[i]/2;
12 }
13 int main(){
14         /*A and B are transferred to device memory*/
15         //calling first kernel
16         kernel1<<<1, N>>>(A, B);
17         /*A and B are transferred back to host memory*/
18         cudaDeviceSynchronize();
19         //calling a CPU function
20         CPU_function(B);
21         /*A and B are transferred to device memory again*/
22         cudaDeviceSynchronize();
23         //calling second kernel
24         kernel2<<<1, N>>>(A, B);
25 }
```

Listing 2.2: An example of CUDA synchronization

[1] This further diminishes the "embarassingly parallel" nature of more GPU code; and, as a result, most GPU programmers avoid algorithms that entail synchronization.

In CUDA, sometimes all the participating GPU threads running a same kernel need to be synchronized. This type of synchronization can be done by specifying synchronization points in a kernel definition by calling the `syncthreads()` function. It acts as a barrier where all participating threads in the same thread block must wait before following the instruction. When all the threads reach that barrier then they can proceed and follow the instructions in the kernel. In Listing 2.2 we provide an example that demonstrates these two types of synchronization in CUDA.

Here, two kernels `kernel11` and `kernel2` are defined ion lines 1–6 and on lines 7–12. In `kernel1`, `syncthreads()` (line 4) specifies a synchronization point. Therefore, threads in a same thread block running `kernel1` must wait on line 4 until all of them reach this synchronization point. Similarly, in `kernel2`, threads in same thread block must wait until all of them reach the synchronization point on line 10.

In this example, `kernel1` is called on line 16 with two arrays `A` and `B`. Then, on line 18, `cudaDeviceSynchronize()` blocks the CPU function `CPU_function` on line 20. The is because `CPU_function` is called with the modified array `B`, which is an output of `kernel1`. Another `cudaDeviceSynchronize()` is mentioned on line 22 that blocks `kernel2` called on line 25, until `CPU_function` is finished. This is because `kernel2` is called with two modified arrays - `A` and `B` - where `B` is an output of `CPU_function`. Note that in this example code for memory transfer operations are not shown.

2.2.7 CUDA library functions

CUDA provides a wide range of library functions [12]. These library functions can be used for querying devices, handling errors generated from CUDA code, managing host and device memory spaces, timing events, managing streams for concurrent executions, managing versions and access to device memory of a peer device. The following addresses these categories of library functions:

- Device Management

  CUDA provides twenty library functions for managing devices. These categories of library functions enables programmers to choose, set, and reset devices to run data-parallel programs. For example,

  `cudaGetDeviceProperties(struct cudaDeviceProp * prop, int dev)`

  returns the properties of device *dev* into a CUDA built-in structure `cudaDeviceProp` type variable *prop* that has fifty-one fields as the properties of device *dev*. `cudaDeviceProp` covers properties such as device name, total global memory, shared memory per thread block, maximum threads per thread block, maximum dimensions of thread blocks and grids, and so on.

  CUDA also provides the library function `cudaChooseDevice()` that allows programmers to choose a device based on the setting of `cudaDeviceProp` structure. Library function `cudaSetDevice(int dev)` sets current *dev* for kernel execution. `cudaGetDevice(int *dev)` returns in **dev* the current

device on which active host executes kernel or device functions. Library function `cudaDeviceReset()` destroys and cleans up all previous resource allocation with the current device.

- Memory Management

  CUDA provides fifty-one library functions for managing different device memory and host memory spaces. In Listing 2.1 we discussed using `cudaMalloc()` for allocating one-dimensional array in device memory space. Now, we address some other library functions. First of all,

  ```
  cudaHostAlloc (void ** pHost, size_t size, unsigned int flags)
  ```

  is used to allocate *size* bytes of page-locked host memory which is accessible to a device. Note that excessive amount of pinned memory may degrade system performance because it reduces available memory for paging. Here *flags* parameters are used to specify different options that affect the allocation.

  CUDA also provides library functions for managing two- or three-dimensional arrays. For example,

  ```
  cudaMemset2D (void * devPtr, size_t pitch, int value, size_t width, size_t height)
  ```

  is used to allocate a two-dimensional array in device memory. Here, device memory pointer *devPtr* allocates a two-dimensional array of *height* rows and *width* columns, and *value* is the specified initial value for the two-dimensional array pointed by *devPtr*. *pitch* specifies the width in bytes of two-dimensional array. Moreover, CUDA also provides library functions `cudaMemset3D()` and `cudaMalloc3D()` to allocate three-dimensional arrays in device memory.

  In order to allocate read-only constant memory CUDA provides library function

  ```
  cudaMemcpyToSymbol(const char * symbol, const void * src, size_t count, size_t offset
                     = 0, enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)
  ```

  Here, *count* bytes are copied to a device memory pointer (may point to constant memory space) *symbol* from host pointer *src*. Here, *offset* is the offset from the start of the symbol in bytes. *kind* specifies the type of transfer.

  CUDA also provides seven library functions for managing texture memory. One of the library functions, `cudaBindTexture()` is used to bind a memory space to a texture. Similarly, library function `cudaBindTextureToArray()` binds an array to a texture.

  Note that for each kind of memory space there is also a memory-transfer operation associated with it. For example, `cudaMemcpy3D()` copies data between two three-dimensional objects. Similarly, `cudaMemcpy2D()` copies data between two-dimensional arrays.

- Event Management

CUDA also provides library functions for setting, synchronizing, and destroying events. These events are used to measure execution time for a kernel in a device. `cudaEvent_t` type variables are used to declare events. Library function `cudaEventCreate (cudaEvent_t *event)` creates an event *event*. `cudaEventRecord(cudaEvent_t event)` records *event*. In order to measure elapsed time between two events library function

`cudaEventElapsedTime(float * ms, cudaEvent_t start, cudaEvent_t end)`

is provided. Events can be destroyed by library function `cudaEventDestroy(cudaEvent_t event)`.

- Error Management

  CUDA provides library functions to manage error generated from CUDA code. Library function `cudaGetLastError (void)` returns a `cudaError_t` type error if any operations related to a device raises an error. In order to get the error message a string library function `cudaGetErrorString(cudaError_t error )` takes `cudaError_t` type variable *error* and returns an error message string associated with *error* .

- Stream Management

  CUDA provides library functions for managing streams. In CUDA, a stream is a sequence of operations that executes in an order. Stream gives the ability to run CUDA operations in different streams concurrently. CUDA operations in different streams can also be interleaved. In CUDA, `cudaStreamCreate (cudaStream_t * ptrStream)` creates a new asynchronous stream (*ptrStream*), and `cudaStreamDestroy(cudaStream_t ptrStream)` destroys and cleans up the asynchronous stream specified as *ptrStream*. Library function `cudaStreamSynchronize(cudaStream_t ptrStream)` blocks until *ptrStream* has completed all the operations in a device.

- Version Management

  CUDA also provides library functions to get the installed CUDA driver version and the CUDA runtime version. Library function `cudaDriverGetVersion(int *versionDriver)` returns the version number of the an installed CUDA driver in **versionDriver*, and `cudaRuntimeGetVersion (int *versionRuntime)` returns the version number of an installed CUDA runtime in **versionRuntime*.

- Peer Device Memory Access

  CUDA provides library functions for managing access to device memory of a peer device. Library function

  `cudaDeviceCanAccessPeer (int * pAccess, int dev, int pDev)`

returns in `*pAccess` a `1` if `dev` can directly access the memory of `pDev` and `0` otherwise. `cudaDeviceEnablePeerAccess(int pDev)` allows peer device `pDev` to access the device memory of the current device. Peer access can be disabled for device `pDev` by `cudaDeviceDisablePeerAccess(int pDev)`.

These library functions are declared either in header-file `cuda_runtime_api.h` or `cuda_runtime.h`. Header-file `cuda_runtime_api.h` is a C-style interface, and library functions declared in this file do not require compiling with `nvcc` compiler. However, header-file `cuda_runtime.h` is a C++ style interface, and the library functions declared in this file must be compiled through `nvcc`. For example, `cudaMallocHost()` must be compiled through `nvcc` because it is declared in file `cuda_runtime.h`. Overall twenty library functions, also known as *C++ API Routines* [12] are declared in `cuda_runtime.h`.

So far in this chapter we discussed parallelism in a GPU programming model. We also covered different language constructs of CUDA-C to develop data-parallel programs. We also discussed some categories of CUDA library functions. In the rest of this chapter, we will review the Gambit Scheme compiler and the C-interfaces in Gambit for linking Scheme code to C code.

2.3 Gambit Scheme an implementation of Scheme programming language

Scheme, a dialect of Lisp, is a mostly functional programming language. It is also a lexically-scoped language and uses dynamically-typed variables. One of the implementations of Scheme is Gambit. Gambit's Scheme compiler can compile Scheme code to C code. Gambit provides C-interface special constructs to link C code from Scheme code. Gambit also implements a full numerical tower of numerical types : numbers, complex, real, rational and integer for Scheme. Moreover, Gambit Scheme is well-maintained, stable, and robust.

In this section, we describe the linking strategy of a linker which is followed by a description of the Gambit Linker. Then we describe the Scheme data types and how they can be mapped to C types. We also describe the C-interface special constructs in order to link C code. Finally, we describe the Gambit compiler usage and flags.

2.3.1 Linking subprograms by a linker

In Gambit, a Scheme program written in multiple files as subprograms can be translated to C code. Those translated subprograms can be translated to object code by C compiler. The object code of subprograms are linked together into a composite program known as *linking*. The program that performs this composition is known as the *linker*.

In order to obtain flexibility and better utilization of main memory, compilers generate *relocatable code*(object code), also know as the *relocatable binary*. This is a program that can be loaded into any location of main memory. In relocatable binary code:

1. Address fields have been translated relative to zero.

2. Relocation information is associated with the program to be loaded to indicate which address field must be relocated.

The linker combines those relocatable subprograms into a single relocatable program. The input to a linker consists of one or more subprograms in *binary symbolic form*. Binary symbolic form is similar to relocatable binary except that an additional table known as the *External Symbol Dictionary* [4], or *ESD*, is included with each subprogram and is translated to relocation binary code. ESD is used to indicate the definition and the use of *external symbols* referenced by other subprograms. Independently translated subprograms communicate each other through the external symbols. It is the responsibility of a linker to combine input subprograms into a single relocatable output in which all the external references have been resolved.

ESD is a table that contains an entry for each external symbol defined within a subprogram. There are two kinds of external symbols:

- *External Name*: a symbol which is referenced from other translated subprograms and is being linked together with the subprogram containing the symbol [2].

- *External Reference*: a symbol which is mentioned as External Name in an another translated subprogram [3].

The part of a binary symbolic file that contains relocatable machine language instruction and data is known as the *Text*. Moreover, binary symbolic files also contain another piece of information known as the *Relocation Dictionary* or *RLD*. It contains one entry for each address that must be relocated when the composite program is loaded into main memory. It is the responsibility of a language translator (compiler or assembler) to group relocation information into RLD. In RLD, each entry is actually a pointer to a machine language instruction that must have its address field relocated.

In liking together a set of subprograms, the linker merges ESDs into a *Composite External Symbol Dictionary(CESD)* [4]. The linker assigns consecutive relative addresses to each External Name. This is done by assigning an address of zero to the first External Name and then assigning addresses relative to this origin to all other External Names. All ESD entries are updated to reflect the new addresses that were assigned.

The CESD is followed by the a sequence of Text and RLD. Each Text and RLD corresponds to the Text and RLD portion of an object file.

Once contiguous addresses have been assigned to the External Symbols in the object files, all entries in the RLD must be relocated relative to the beginning of the composite output being created by the linker. The end of an composite file is indicated by an *END-Of-Module record*, or *EOM*. The composite file is then ready to be loaded in memory by a *loader* [23]. A loader that loads relocation binary form and updates all relative addresses known as *relocating loader*. A relocating loader first requests for space in main memory for the composite program and sets the starting address for the composite program. Note that the allocation

of memory to a given program by a relocation loader will remain fixed for the duration of that program's execution.

Gambit also has a linker that does not actually link subprograms rather it generates a link file from generated subprograms produced by the Gambit compiler. This link file is actually used later by the C-linker (running under `gcc` compiler) in order to generate a composite relocating binary file.

2.3.2 Gambit linker

C files generated by the Gambit compiler must be linked together to generate a composite program. In order to do that the Gambit linker generates a link file that contains various linking information gathered from the generated C files. It contains a set of all symbols and a list of subprograms and global variables used in a group of subprograms. This link file also contains linking information for external references. For example, If a Scheme procedure makes reference to an external symbol, this linking information is actually stated in the link file generated by the Gambit linker. This linking information of external references is utilized later by the C-linker that generates a composite program from subprograms. This link file is also needed to initialize the Scheme runtime system.

In general, a program for Gambit is composed of a set of Scheme subprograms or C subprograms. Some of the subprograms are part of Gambit runtime library and others are supplied by the programmers. When a program is started it must set up various global tables containing all the symbols and the supplied global variables. Then all Scheme subprograms are executed sequentially. The information required for setting up the global tables and sequential execution of subprograms is contained in the link file. This link file makes it possible to use external procedures and variables in a program. Note that this link file is a C file and it must be compiled to an object file by the C compiler. Later it is given to linker as an object file to generate the composite program along with the compiled object files.

In this thesis, we need to ensure that CUDA-C code can be linked to Scheme code. In order to do that, Scheme code refers symbols defined in CUDA-C code. Therefore, a C compiler requires linking information about Scheme code. This linking information is actually provided by the link file generated by Gambit linker.

2.3.3 Mapping of types between Scheme and C

Scheme and C languages do not provide the same set of data types. Gambit Scheme provides *foreign object* types [33] in order to map values from Scheme to C. Gambit also adds new C types [33] for those foreign object types. A foreign object type is internally represented as a C pointer. It is important to know which Scheme types are compatible with C and how the compatible types can be mapped back and forth.

In Scheme, a vector type is a sequence of values in memory, like an array in C. Gambit provides homogeneous vectors of raw numbers for both signed and unsigned exact integers and inexact reals. The foreign object type for a Scheme vector is `scheme-object`, which is also the universal type for Scheme objects. The added C-type for this universal type is `___SCMOBJ` (defined in `gambit.h` header file). Therefore, `___SCMOBJ` is

also the C type for Scheme vector. Note that for a Scheme vector it is not enough to convert it from Scheme to C in order to access it from C code. It also requires a pointer-casting operation for that vector. This casting operation is called `___CAST` discussed with the example Scheme program in Listing 2.3. The reason for this is a Scheme vector contains extra information used by Scheme wrapped around a C pointer to an actual C array.

In Gambit, a converted argument can be referred in C code through the built-in variable `___arg`; this is suffixed by the position of a variable in a parameter list. For example, the first argument can be referred as `___arg1`, the second argument can be referred as `___arg2`, and so on. The foreign object type for a scalar is enough to convert it to C. It does not need the extra type-casting operation. A scalar type converted argument can be referred by built-in variable `___arg` followed by its position in parameter list in C code.

In this these in order to link Scheme code to CUDA-C code, we will need to pass data from Scheme code to CUDA-C code. CUDA-C requires types for data because it is an extension of C. Therefore, we will need to convert Scheme data to C data to their appropriate types. This conversion also gives the ability to access Scheme data from C.

2.3.4 Linking a C code sequence using `c-lambda`

Gambit has the ability to link Scheme code to C code using the C-interface construct `c-lambda`. This special form gives a way of representing a Scheme procedure that will act as a representative of a sequence of C code. The first subform of a `c-lambda` function is a list that contains types of arguments. When a `c-lambda` function is called all the arguments are coerced [47] to their C types. The return type of the procedure is given next. The last sub-form is a string that contains a sequence of C code.

In Listing 2.3 `c-lambda` function `assign_vector` (lines 2–13) provides a sequence of C code. The keyword `c-lambda`, on line 3 specifies that it is a `c-lambda` function. This function takes a vector and its length and assigns each element of that vector. In the parameter list (line 3),

```
(scheme-object int)
```

`scheme-object` is the foreign object type for the Scheme vector and `int` is the foreign object type for the length. This `c-lambda` function is called

```
(assign_vector vec N)
```

on line 15. Here, first argument `vec` is a vector of 32-bit unsigned integers defined on line 15 as

```
(define vec (make-u32vector 512 0))
```

```
1  ;;Scheme procedure with a sequence of C code
2  (define assign_vector
3      (c-lambda (scheme-object int)           ;;parameter list
4                void                          ;;return type
5  #<<c-lambda-end
6        ;;--------------- C code start----------------------------
7          int i;
8          ___U32* a_h = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
9          for (i=0; i< ___arg2; i++){
10             a_h[i] = i;
11          }
12       ;;--------------- C code end-----------------------------
13 c-lambda-end))
14
15 (define vec (make-u32vector 512 0))
16 (define (top)
17   (let ([N (u32vector-length vec)])
18      (assign_vector vec  N)
19      (display vec))
```

Listing 2.3: A Scheme procedure calling a `c-lambda` function containing a sequence of C code-fragment

Its length is `512` elements, and initially its elements are assigned with `0`. The second argument `N` is the length of this vector which is calculated on line 16 as

```
([N (u32vector-length vec)])
```

Therefore, in parameter list, `scheme-object` and `int` are the types for `vec` and `N` respectively. These types are appropriate to be converted from Scheme to C. The return type of this function is `void` on line 4 because this function does not return anything to Scheme.

A sequence of C code in a `c-lambda` function can be defined within the scope of `#«c-lambda-end`, on lines 5 and 13. In this example, the pointer-casting operation for the vector `vec` is shown on line 8 as

```
___U32* a_h = ___CAST(___U32,___BODY_AS(___arg1, ___tSUBTYPED));
```

Here, C pointer type `___U32`(defined in `gambit.h`) with variable `a_h` is assigned with the cast pointer by macro `___CAST`. The pointer is extracted by macro `___BODY_AS` (also defined in `gambit.h`) before the casting operation and `___arg1` refers to the first parameter Scheme vector `vec`. We need to specify a tag for a vector because it is a memory-allocated object in Scheme. For Gambit, tag `___tSUBTYPED` (defined in `gambit.h`) is

provided to specify a memory allocated object which is also not a pair. In Gambit, a pair is also a memory allocated object. In order to specify a pair, another tag `___tPAIR` (defined in `gambit.h`) is provided.

However, for a scalar type only the foreign object type is required in order to map from Scheme to C. On line 9, `___arg2` refers to type `int` which is the length `N` of vector `vec`. A C `for` loop is used to loop through the vector, on lines 9–11, and assigns value to each element on line 10,

```
a_h[i] = i;
```

of this vector.

Note that a `c-lambda` function must be in a file with a `.scm` extension. In Gambit it is also possible to link an external stand-alone C function in a C file from Scheme code through the `c-lambda` function. In this case, `c-lambda` function converts data from Scheme to C and calls the C-function, but it requires another C-interface construct `c-declare` to provide the forward declaration of that C function.

2.3.5 Linking a C-function using `c-lambda` and `c-declare`

Gambit also has the ability to link Scheme code to an external C function in a `.c` file. Therefore, Gambit provides a C-interface construct -`c-declare` -to link a C function, but this must be assisted by a `c-lambda` function for type conversion. Listings 2.4 and 2.5 show how a C function in a `.c` file can be linked from Scheme.

In Listing 2.4 lines 1–6, a C function `assign_vector_C`, in a file `c_function.c` is linked from Scheme. This C function takes two arguments a C-pointer `a_h` of type `uint32_t*` pointing to a vector in C, and the length of that vector `N` of type `int`. A `for` loop on lines 4–6, loops through and assigns values to each element of vector `a_h`.

```
1 //file: c_function.c
2 void assign_vector_C (uint32_t* a_h, int N ){
3         int i;
4         for (i = 0; i < N; i++){
5                 a_h[i] = i;
6         }
7 }
```

Listing 2.4: C function `assign_vector_C` in `c_function.c` file

In Listing 2.5 we show Scheme code that calls this C function `assign_vector_C`. First, it calls a `c-lambda` function `assign_vector` with two arguments - `vec`, and `N` - on line 17,

```
(assign_vector vec N)
```

Here, `vec` is a Scheme vector defined on line 14. The `c-lambda` function on line 7 is called to convert data types from Scheme to C. This `c-lambda` function also calls the C function `assign_vector_C` on line 11 with two arguments,

```
assign_vector_C (a_h, ___arg2);
```

Here, `a_h` is the cast pointer for Scheme vector `vec` and `___arg2` refers to second parameter `int` in the parameter list on line 7; this represents length `N` on line 17 of this vector. Note that on line of Listing 2.5, type `___U32` is compatible with the type `uint_32` for pointer `a_h` in C function `assign_vector_C`. Gambit does not allow a vector to pass to a C function using `___arg` prefix. The pointer must be extracted and cast from Scheme. Then the pointer is passed to a C function. For a scalar type `___arg` prefix is used for passing it to a C function.

```
1  (c-declare  #<<c-declare-end
2       void  assign_vector_C(int* a_h, int N);
3  c-declare-end
4  )
5
6  (define assign_vector
7    (c-lambda (scheme-object int)           ;;parameter list
8              void                          ;;return type
9  #<<c-lambda-end
10         ___U32* a_h = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
11         assign_vector_C(a_h, ___arg2);    ;;calling C function
12 c-lambda-end
13 ))
14 (define vec (make-u32vector 512 0))
15 (define (top)
16   (let ([N (u32vector-length vec)])
17     (assign_vector vec  N)
18     (display vec)))
```

Listing 2.5: Linking C function `assign_vector_C` from file `scheme_driver.scm`

The definition of `assign_vector_C` is in file `c_function.c`, but the call is in file `scheme_vector.scm`. Therefore, a forward declaration of `assign_vector_C` must be provided in `scheme_vector.scm`. The forward declaration for `assign_vector_C` is provided within the `c-declare` construct on line 2 of Listing 2.5:

```
void assign_vector_C (uint32_t* a_h, int N);
```

Finally, newly assigned values to vector `vec` are displayed, on line 18, as

```
(display vec)
```

We have showed how Scheme code can be linked to a sequence of C code using a `c-lambda` construct. We have also showed how an external C function in a `.c` file can be linked using both the `c-lambda` and `c-declare` constructs. Through our discussion, we also showed that `c-lambda` functions convert data from Scheme to C types. We also found that a Scheme vector can be passed to a C function by passing the extracted C pointer of that vector. Note that `c-lambda` and `c-declare` constructs act as interfaces to C code. The actual linking of these subprograms to an executable program is done by the C-linker.

In Table 2.5 we show added C types for Gambit with their equivalent original C types.

Table 2.5: Compatible C types in Gambit Scheme

Scheme	C
___U8	uint8_t
___S8	int8_t
___U16	uint16_t
___S16	int16_t
___U32	uint32_t
___S32	int32_t
___U64	uint64_t
___S64	int64_t
___F32	float
___F64	double

2.3.6 Gambit Scheme Compiler usage and flags

The Gambit Scheme compiler is a program called `gsc`. Usually `gsc` accepts files with the `.scm` extension containing Scheme code. It also accepts `.c` files containing C code.

In order to generate an executable, `gsc` follows the following four steps:

1. First, `gsc` generates C code in `.c` files from `.scm` files with the same name using the command line option `-c`.

2. Then the C files generated by the Gambit compiler are used by the Gambit Linker using the command `-link` to generate a link file or a C file that contains various linking information. By default, the name of the link file is suffixed with a `_` , and the name is the last generated C file's name in the command line.

3. Next, the command line option `-obj` is used to compile `.c` files to object files with `.o` extensions by `gsc`. This actually invokes `gcc` compiler internally.

4. Finally, C compiler builds executables from those `.o` files.

However, `gsc` can use two command-line options - `-o` and `-exe` - to make an executable in a single step by implicitly invoking the intermediate steps.

In Listing 2.6, a `makefile` shows how the Scheme file `scheme_driver.scm` in Listing 2.5 and the C file `c_function.c`, in Listing 2.4 can be compiled by `gsc`. First, `gsc` uses command-line options `-c` to compile `scheme_driver.scm` to `scheme_driver.c` on line 9.

```
gsc -c scheme_driver.scm
```

```
1 GAMBIT_DIR = /usr/local/Gambit-C
2 scheme_driver_.c: scheme_driver.c
3         gsc -link scheme_driver.c
4         gsc -obj scheme_driver.c scheme_driver_.c c_function.c
5         gcc -I$(GAMBIT_DIR)/include\
6             -L$(GAMBIT_DIR)/lib\
7             scheme_driver.o scheme_driver_.o c_function.o -lgambc
8 scheme_driver.c: scheme_driver.scm
9         gsc -c scheme_driver.scm
```

Listing 2.6: `gsc` commands to build executable from a `.scm` file and a `.c` file

Next, `gsc` uses option `-link` to create a link file (`scheme_driver_.c`) by invoking the Gambit linker. The linker gathers linking information such as all the symbols and supplied global variables from compiled C file (`scheme_driver.c` on line 3.

```
gsc -link scheme_driver.c
```

Note that, `scheme_driver.c` is the last and only generated `.c` file by `gsc`. Therefore, the name of the link file is `scheme_driver_.c`. Next, `gsc` creates object files `scheme_driver.o`, `scheme_driver_.o`, and `c_function.o` from `scheme_dri- ver.c`, `scheme_driver_.c`, and `c_function.c`, on line 4,

```
gsc -obj scheme_driver.c scheme_driver_.c c_function.c
```

Here, `c_function.c` contains plain C code. Finally, system C compiler `gcc` generates default executable `a.out` from object files `scheme_driver.o`, `scheme_driver_.o`, and `c_function.o`, on lines 5–7. The C compiler needs to know the include path and the runtime libraries of Gambit systems. Therefore, line 5,

```
-I$(GAMBIT_DIR/include)
```

provides the location of `gambit.h` header-file to C compiler using the C flag `-I`. On line 6,

```
-L$(GAMBIT_DIR/lib)
```

provides the location of Gambit's runtime libraries using the flag `-L`.

2.4 Summary

In this chapter, we gave details about the GPU programming model. We also described the different types of GPU memory with their features. Then we gave details about CUDA-C language constructs to develop data-parallel programs. We also described the linking strategy of a linker. Then we described data type mapping between Scheme to C, and the C-interfaces to link Scheme code to C code in Gambit.

The primary motivation of our work is to link Scheme code to CUDA-C kernels by generating a foreign-function interface from Scheme that reduces hands-on memory management with reasonable overhead in runtime. The next chapter will describe the designing of the generated interface in Gambit that links Scheme code to CUDA-C kernels.

Chapter 3

Design

Gambit provides special constructs to link C code with Scheme code. These special constructs can be used in a foreign-function interface that links a CUDA-C kernel to Scheme. In this chapter we describe our design for such a foreign-function interface.

At the beginning of this chapter we describe the necessary parts of a foreign-function interface to link a CUDA-C kernel from Scheme. First we describe the parts designed using Scheme constructs, and next we describe the parts of the foreign-function interface designed using CUDA-C constructs. We then we provide the complete code of a foreign-function interface in Listings 3.2 and 3.3. Finally, we give an example of `makefile` that shows the necessary command-lines to compile this foreign-function interface through Gambit.

3.1 A foreign-function interface for linking Scheme code to a CUDA-C kernel

In this thesis we use both Scheme and CUDA-C constructs to design our foreign-function interface. Gambit provides C-interfaces such as `c-lambda` and `c-declare` to link C code from Scheme.

These parts of the foreign-function interface designed with Scheme constructs — referred to collectively as the *Scheme shim* — reside in a file with a `.scm` extension. The Scheme shim links a function call in Scheme to CUDA-C constructs of this foreign-function interface. These constructs include operations in device memory and kernel call with execution configuration.

Those parts of the foreign-function interface designed with CUDA-C constructs — referred to collectively as the *CUDA-C shim* — reside in a file with a `.cu` extension. The CUDA-C shim connects the Scheme shim to a CUDA-C kernel. We separate the Scheme constructs from the CUDA-C constructs into two different files, because Gambit forbids some standard C constructs in C-interface constructs.

In our approach, a function call in Scheme calls to a CUDA-C kernel through the both shims of this foreign-function interface. Therefore, kernel arguments of a function call in Scheme linking to a CUDA-C kernel are passed through both shims of this foreign-function interface to that CUDA-C kernel.

A CUDA-C kernel must be called from a C host function and the call for that kernel must have an execution configuration. Scheme does not have any construct to define the execution configuration. Therefore, we pass

the parameters defining a execution configuration for a CUDA-C kernel through the Scheme shim to the CUDA-C shim, and the kernel is called from CUDA-C shim. In this foreign-function interface, the first seven arguments of a function call in Scheme linking to a CUDA-C kernel represent execution configuration parameters. The first three of them define x-, y- and z-dimensions of a grid; the next three defines x-, y- and z-dimensions of a thread block; and the seventh argument defines the amount of dynamic shared memory needed by that CUDA-C kernel. These seven arguments are not passed directly to a CUDA-C kernel. Instead, CUDA-C shim uses them to define an execution configuration as a part of a kernel call in CUDA-C. They are converted to C types in Scheme shim because `gsc` requires that Scheme to C data type conversions must happen in C code defined with C-interfaces

The rest of the arguments of a function call in Scheme are the actual kernel arguments. They are passed through the Scheme shim and the CUDA-C shim of this foreign-function interface to a CUDA-C kernel. Note that these actual kernel arguments are defined in Scheme; this means they reside in host memory. They need to be transferred to device memory for data-parallel computation by a kernel because a CUDA-C kernel can only access device memory. Furthermore, because they are Scheme values they are also converted to C types. In CUDA-C, the copy of a vector in device memory is actually passed to the CUDA-C kernel. Moreover, to allocate space in device memory for each vector, the lengths of those vectors are also required. Lengths are only available to Scheme. Therefore, lengths of Scheme vectors are also calculated in Scheme shim and passed to CUDA-C shim to allocate space in device memory.

In many parallel applications the C code requires the vector length. Therefore we also pass lengths of Scheme vectors to a CUDA-C kernel. The benefit of this is that programmers don't need to send lengths of vectors to a kernel manually. However, in cases where vector lengths may not be used for data-parallel computation in a kernel, these extra arguments are superfluous.

The Scheme shim of this foreign-function interface links a function call in Scheme to a CUDA-C shim has three parts:

1. A vector-length-calculation helper function calculates length of a Scheme vector passed as an argument to a CUDA-C kernel. It takes all arguments from a function call in Scheme and calls a `c-lambda` function with all the arguments it takes. For each vector it passes one extra arguments that calculates the length of that vector.

2. A `c-lambda` function performs Scheme to C data type conversion and calls into the CUDA-C shim with the converted C arguments in host memory. Data defined in Scheme does not have type information, but CUDA-C shim needs type information for those Scheme data. We therefore need to convert those Scheme data to C-type data.

3. A forward declaration of the CUDA-C shim entry point defined within a `c-declare` construct. The `c-lambda` function calls the CUDA-C shim.

The CUDA-C shim of this foreign-function interface links the Scheme shim to a CUDA-C kernel. The CUDA-C shim executes certain operations:

- It declares a device memory pointer for each converted C vector in host memory allocating space in device memory for each device memory pointer using the supplied lengths of vectors.
- It transfers each vector from host to device memory.
- It also defines CUDA-C built-in vector data types to define the execution configuration parameters.
- Next, the CUDA-C shim calls a kernel with the execution configuration parameters. It also passes the actual kernel arguments to the CUDA-C kernel. Note that for a vector as kernel argument length of that vector is also passed to the kernel.
- After a kernel finished its execution, CUDA-C shim copies vectors from device to host memory to return resultant data.
- Finally, it deallocates device memory occupied by the device memory pointers.

In Figure 3.1 we show how our foreign-function interface links a function call in Scheme to a CUDA-C kernel. In this diagram, the blue area represents the Scheme shim, and green represents the CUDA-C shim.

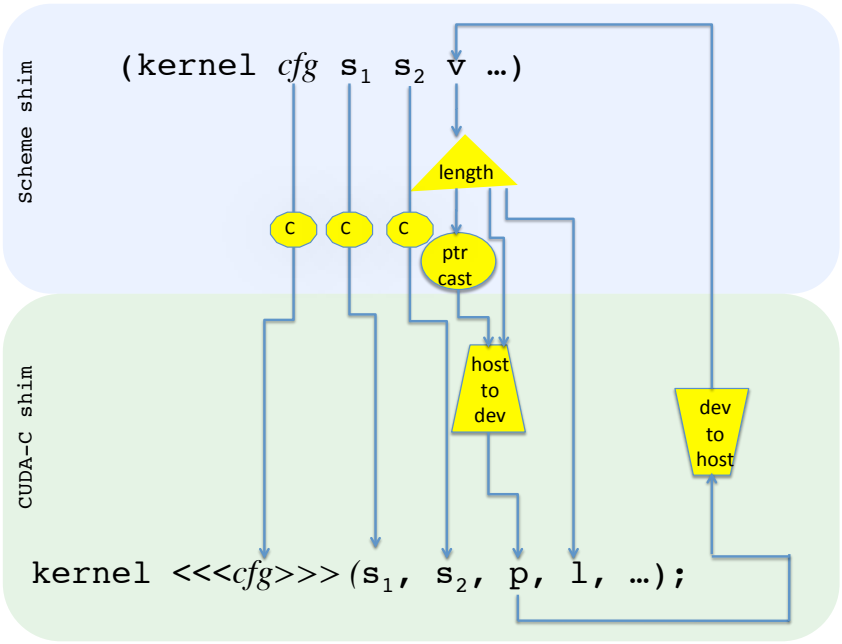


Figure 3.1: Foreign-function interface linking a function call in Scheme to a CUDA-C kernel

In this diagram, a CUDA-C kernel is called from Scheme as a function call in Scheme

```
(kernel cfg s1 s2 v ...)
```

Here, `kernel` is the name of the kernel. It passes seven execution configuration parameters which is represented by the first argument `cfg`. It also passes the scalars `s1, s2, ...` and a vector `v, ...` to the CUDA-C kernel.

Each arrow in this diagram represents a direction of data flow. The following line

```
kernel  <<<  cfg >>>  (s1, s2, p, l, ...);
```

is a kernel call in CUDA-C shim. The name of the kernel is `kernel` which is same as in Scheme. This is because we want to make the name of a kernel consistent to the programmers. Changing the name in CUDA C shim may create confusion for programmers because the name in function call would be different with the actual definition in CUDA-C.

The arrow from `cfg` in function call in Scheme pointing to `cfg` of CUDA-C call represents the seven execution configuration parameters passed through the Scheme shim to CUDA-C shim. These appear at the kernel call in CUDA-C to define the execution configuration. Similarly, arrows from scalar types `s1`, and `s2` from function call in Scheme point to `s1`, and `s2` in CUDA-C kernel. This signifies they passed through the Scheme shim to the CUDA-C shim and passed to the kernel. Note that seven execution configuration parameters and scalar types are converted to C types by Scheme Shim's `c-lambda` function before passing them to CUDA-C shim, as represented by yellow dodecagons with `C`.

In this diagram we also show how vector type `v` is passed to a CUDA-C kernel. The arrow from vector `v` to the yellow triangle in Scheme shim denotes that this vector is passed to the vector-length-calculation helper function. From this triangle one arrow is pointing to the yellow oval in Scheme shim; this denotes that the vector will pass to the `c-lambda` function to cast the C pointer. The arrow from the yellow oval to the yellow trapezoid in CUDA-C shim denotes that the C pointer is passed to the CUDA-C shim to be copied to device memory. In order to copy this vector to device memory, the length of this vector is also required. The length of vector `v` is also passed to CUDA-C shim for copy operations, as denoted by another arrow from the yellow triangle to the same yellow trapezoid. Next, the arrow from the yellow trapezoid to the kernel argument `p` of CUDA-C kernel call denotes that after the copy operation the device memory pointer `p` is passed to the kernel. The arrow from the yellow triangle in Scheme shim to kernel argument `l` denotes that the length of vector `v` is also passed to the kernel. Note that length `l` is exactly the same as the device memory pointer `p` corresponding to the host pointer of vector `v`.

Finally, the arrow from `p` to the other yellow trapezoid in CUDA-C shim denotes that the pointer `p` is copied back to host memory. The arrow from this yellow trapezoid pointing to vector `v` represents that after data is copied to host memory it returns back to Scheme.

Note that if the vector is passed to that kernel only to assign it with the kernelâĂŹs resultant data, a vector might not need to be copied to device memory before a kernel call. Similarly, if that vector is passed to that kernel only to serve as input vector, it might need not to be copied back to host memory after a kernel call.

In the following sections we describe parts of Scheme shim and CUDA-C shim based on code provided in Listings 3.2 and 3.3.

3.2 Components of Scheme shim

The aforementioned three parts of the Scheme shim perform necessary transformations to connect with CUDA-C code from Scheme code. A function call in Scheme interacts with Scheme shim's vector-length-calculation helper function, which calls the `c-lambda` function for data type conversion. This `c-lambda` function also calls the CUDA-C shim residing in a `.cu` file. A forward declaration of that CUDA-C shim defined in a `c-declare` construct acts as an entry point.

In Listings 3.1 and 3.2 we provide code that demonstrates how a function call in Scheme actually interacts with a Scheme shim [1]. The code in Listing 3.1 shows data types defined in Scheme and a function call to the Scheme shim. In Listing 3.2 we provide code for that Scheme shim. For this example foreign-function interface we append Scheme shim in Listing 3.2 with the function call in Listing 3.1 in the file `main.scm`. Note that our implementation generates the Scheme shim in a separate `.scm` file and `gsc` combines it with the file that has the function call.

```
1 (define N 100)
2 (define src (make-u32vector N 0))
3 (define constant 786)
4 (let ((nblocks 1)
5       (blockSize N)
6       (size-int32 4)) ;-----------------cfg-------------------------    ;s    ;v
7   (vector_addition  nblocks 1 1 blockSize 1 1  (* (* 2 N) size-int32) constant src)
8   (display src))
```

Listing 3.1: Data types defined in Scheme and a call to the Scheme shim of the foreign-function interface

The goal of calling the Scheme shim of our foreign-function interface from Scheme is to call a CUDA-C kernel, which requires an execution configuration. Therefore, in the foreign-function interface, we are passing seven arguments along with the actual kernel arguments through the parts of the Scheme the shim to the

[1] In Listing 3.2, lines 3–8, 13–24, 29–37, are code generated from our implementation. We have added some comments and adjusted spaces to improve readability.

CUDA-C shim to define the execution configuration and dynamic shared memory. The first six arguments define the execution configuration, and the seventh one defines the size of the dynamic shared memory.

To understand how this all works, we begin at the function call in Scheme to the CUDA-C kernel in Listing 3.1 (line 7):

```
(vector_addition nblocks 1 1 blockSize 1 1 (* (* 2 N) size-int32) constant src)
```

Here, `vector_addition` is the name of the kernel called. This kernel takes two arguments — a constant and a vector — from the host. Each participating GPU thread adds the constant with its corresponding element in the list. After these parallel additions by participating threads, data is returned back to the host. Finally, the host program displays the result.

This kernel is called with nine arguments: `nblocks`, `1`, `1`, `blockSize`, `1`, `1`, `(* (* 2 N) size-int32)`, `constant` and `src`. This kernel call actually calls the Scheme shim of the foreign-function interface of this kernel. Of the first six, the first three (`nblocks`, `1`, and `1`) are passed to define the x-, y-, and z-dimensions of the grid. Next, three more arguments (`blockSize`, `1`, and `1`) are passed to define x-, y-, and z-dimensions of a thread block. Although this example does not need to use shared memory, the seventh argument `(* (* 2 N) size-int32)`, supplies the value to determine the size of dynamic shared memory. The last two arguments (`constant` and `src`) are the actual kernel arguments that are passed through the Scheme shim and CUDA-C shim to the CUDA-C kernel. Here, variable `constant` is defined in Listing 3.1 (line 3) as

```
(define constant 786)
```

and `src` is a Scheme vector of 32-bit unsigned integer of length `N` initialized to `0` which is defined on line 2 as

```
(define src (make-u32vector N 0))
```

The actual definition of kernel `vector_addition` is in file `kernel.cu` but we do not show the code. We generate the forward declaration of this kernel with the CUDA C shim, on lines 2–3 of Listing 3.3. This takes only three arguments: `u32_constant`, `u32v_srcl` and `u32v_src_len`. In this foreign-function interface, the first seven arguments are passed through Scheme shim to CUDA-C shim to define the execution configuration in a call to this kernel. The rest of the arguments are the actual kernel arguments passed through the Scheme shim to the CUDA-C shim. Our implementation does not generate the body of a kernel, it just generates the skeleton of a kernel. Therefore, in Listing 3.3, we only provide the skeleton of kernel `vector_addition`.

The call to `vector_addition` kernel on line 7 of Listing 3.1 actually calls the Scheme shim's vector-length-calculation helper function `vector_addition`, shown on lines 29–37 of Listing 3.2. Note that the name of the CUDA-C kernel and vector-length-calculation helper function in Scheme are identical. The reason for this is that our implementation in Gambit compiler actually generates this foreign-function interface and

```
;;-------------------------------Scheme shim Start---------------------------
;; A c-declare construct to provide a forward declaration of the CUDA C shim.
(c-declare #<<c-declare-end
        //forward declaration for CUDA C shim
        void vector_addition_cu_driver();

c-declare-end
)
;;the c-lambda function performs data type conversion and calls the CUDA C shim
;;vector_addition_scm_driver :: int * int * int * int * int *
;;                              int * int * uint32_t *
;;                              u32vector * int -> null
(define vector_addition_scm_driver
  (c-lambda (int int int int int int int unsigned-int32 scheme-object int)
            void
#<<c-lambda-end
        //casting to C pointer
        ___U32* host_u32v_src =___CAST(___U32*,___BODY_AS(___arg9,___tSUBTYPED));
        //calling the CUDA-C shim
        vector_addition_cu_driver( ___arg1, ___arg2, ___arg3, ___arg4, ___arg5,
                                   ___arg6, ___arg7, ___arg8,
                                   host_u32v_src, ___arg10);
c-lambda-end
))
;;vector-length-calculation helper function
;;vector_addition :: int * int * int * int * int *
;;                   int * int * uint32_t *
;;                   u32vector -> null
(define (vector_addition gDx gDy gDz bDx bDy bDz shared-size u32_constant u32v_src)
;;calling the c-lambda function vector_addition_scm_driver along  with the
;;vector-length-calculation  as an argument
  (vector_addition_scm_driver
    gDx gDy gDz bDx bDy bDz shared-size
    u32_constant
    u32v_src
    ;;vector-length-calculation function
    (u32vector-length u32v_src)))
;;-----------------------Scheme shim End-------------------------------------
```

Listing 3.2: Scheme shim links a function call in Scheme to CUDA-C shim

the CUDA-C kernel skeleton from a kernel defined in Scheme. In order to link the CUDA-C kernel from the a function call in Scheme, we must call a part of the foreign-function interface. In this foreign-function interface, the vector-length-calculation helper function of Scheme shim is called. We can generate a different name for the skeleton of a CUDA-C kernel, but in that case the actual definition of that kernel in Scheme would be different. This may create confusion for programmers. Therefore, the name of the vector-length-calculation helper function is same as the name of the CUDA-C kernel.

In Listing 3.2 (lines 29–37) we show the code for the vector-length-calculation helper function `vector_addition` that is actually called when we call the kernel from Scheme on line 7 of Listing 3.1. This helper function has the exact same name as the CUDA-C kernel, shown on line 2 of Listing 3.3. This helper function takes nine parameters: `gDx`, `gDy`, `gDz`, `bDx`, `bDy`, `bDz`, `shared-size`, `u32_constant`, and `u32v_src`, as shown on line 29 of Listing 3.2.

3.2.1 A vector-length-calculation helper function

In Scheme shim, a vector-length-calculation helper function is used to calculate lengths of Scheme vectors passed as arguments to a CUDA-C kernel. Before passing them to a kernel, these vectors are passed to the CUDA-C shim to allocate space in device memory. Therefore, we need to know the lengths of these vectors for device memory allocation. These vectors are defined in Scheme, therefore their lengths are implicit to the Scheme environment. Since the C environment does not have access to Scheme environment, a Scheme function needs to calculate lengths of these vectors. Therefore, in Scheme shim a vector-length-calculation helper function — which is also a Scheme function — calculates lengths of Scheme vectors and passes them as arguments to the `c-lambda` function. In the argument list, the position of length calculation for a vector is next to it, because it is easier for programmers to recognize the length calculation for a vector. In `c-lambda` function lengths are converted from Scheme to C types and passed to the CUDA-C shim to allocate device memory for their corresponding vectors.

We show the code for the vector-length-calculation helper function `vector_addition` on Listing 3.2 (lines 29–37). This helper function takes nine parameters: `gDx`, `gDy`, `gDz`, `bDx`, `bDy`, `bDz`, `shared-size`, `u32_constant`, and `u32v_src`. All of these parameters, along with the length-calculation function call for the vectors, are passed to the `c-lambda` function.

We follow a naming convention to name the parameters used in this function. In the parameter list, the first three parameters represents dimensions of a grid. Their names names starts with a `g` for grid, are followed by a `D` for dimension, and end with a specific dimension, either `x`, `y`, or `z`. Similarly, the next three parameters represent dimensions of thread blocks and follow almost the same naming convention. Instead of `g`, they starts with a `b` for thread block, are followed by a `D` for dimension, and end with a dimension of either `x`, `y` or `z`. In the parameter list, we name the seventh parameter for dynamic shared memory as `shared-size`. Here, `shared` is for shared memory, and `size` is for runtime size. By looking at lines 29 and 33 in Listing 3.2, we can see that `gDx`, `gDy`, and `gDz` represent grid dimensions; `bDx`, `bDy`, `bDz` represent block

dimensions; and `shared-size` represents the size of dynamic shared memory.

We also follow a naming convention to name the actual kernel arguments in this helper function by embedding type information with the name. The naming convention for a scalar type is different from a vector type. For a scalar type, the naming convention is

`[s, u, f]`*`SIZE_NAME`*

Here, the name of a scalar type starts with either an `s`, `u` or `f`, for signed number, unsigned number, or floating-point number, respectively. Next, *`SIZE`* is for the size of a variable in bits. *SIZE* may differ based on the type of number.

- For a signed (`s`) or unsigned (`u`) number, *`SIZE`* may be either `8`, `16`, `32`, or `64`.
- For a floating-point (`f`) number, *`SIZE`* is either `32` or `64`.

`NAME` is the name of that variable defined in Scheme. There is a `_` between *`SIZE`* and *`NAME`*; this helps the programmer to differentiate type information from the actual name. It also makes the parameter names more readable. On line 7 of Listing 3.1, the eighth argument `constant` is an actual kernel argument of scalar type. Therefore, on line 29 of Listing 3.2 it is named as `u32_constant` and it says that it is 32-bit unsigned integer of scalar type. Note that seven execution configuration parameters do not have the type information in their names because we know in advance that their types are going to be `int` types. This also helps the programmer to clearly identify them from the actual kernel arguments.

For a vector in parameter list, our naming convention is

`[s, u, f]`*`SIZE`*`v_[IN\OUT]`*`NAME`*

A `v` in a vector-type's name after the *`SIZE`* differentiates it from a scalar type, but in all other regards it is the same. Two optional notations - either `IN` or `OUT` - after `v_` specify a direction of a copy operation for a vector between host and device memory. Here, `IN` specifies that a vector is only copied to device from host memory, and will not be copied back to host memory after the execution of a kernel. `OUT` specifies that a vector is to be only copied back to host from device memory after the execution of a kernel. Initially, it is not copied to device memory before a kernel call. If neither `IN` not `OUT` is specified then a vector is copied back and forth between host and device memory. We provide these two optional notations to avoid unnecessary memory transfer operations that create runtime overhead. It also helps programmers to identify copy operations related to a vector.

For a vector type in this foreign-function interface, we declare a device pointer, allocate device memory, and copy data back and forth to its corresponding host memory pointer. However, for a scalar type we don't need to perform these operations. We put `v` after the type information of a vector type since there are extra operations for a vector type. The optional notation `[IN\OUT]` after `v_` specifies the direction of vector

transfer operation. On line 7 of Listing 3.1, the ninth argument `src` is a vector of 32-bit unsigned integers. Therefore, it appears as `u32v_src` in parameter list on line 29 of Listing 3.2. `u32v_src` does not have any optional symbol `IN` or `OUT` that specifies whether that this vector will be transferred to device from host memory before a kernel call, and returned back to host memory afterwards.

In our implementation, the parameters of a kernel defined in Scheme follow exactly the same naming conventions for scalar and vector types. Our implementation generates type information in the foreign-function interface based on embedded type information in a kernel's parameter names.

In Listing 3.2 (lines 32–37), the vector-length-calculation helper function calls the `c-lambda` function `vector_addition_scm_driver` with ten arguments: `gDx`, `gDy`, `gDz`, `bDx`, `bDy`, `bDz`, `shared-size`, `u32_constant`, `u32v_src`, and `(u32vector-length u32v_src)`. Here, the ninth argument `u32v_src` is a Scheme vector. Therefore, this helper function calculates the length of this vector as it is the tenth argument of the `c-lambda` call, shown on line 37

```
(u32vector-length u32v_src)
```

Here, `u32v_src` is a vector for 32-bit unsigned integers. Therefore, we use the Scheme-library function `u32vector-length` for length calculation, which is compatible with this type of vector. Note that these length calculation library functions must be compatible with the types of vectors defined in Scheme.

3.2.2 A `c-lambda` function

In the Scheme shim, the `c-lambda` function is called from the vector-length-calculation helper function. In parameter list of a `c-lambda` function, the first six parameters describe the execution configuration and the seventh describes the size of dynamic shared memory to be defined in CUDA-C shim. The rest of the parameters are the actual kernel arguments and lengths of vectors. In the parameter list, if a parameter is an actual argument and it is a vector, then the next parameter will be the length of that vector. The `c-lambda` function performs Scheme to C data type conversions, executes casting operations for Scheme vectors, and calls the CUDA-C shim with the converted C parameters. In Listing 3.2 (lines 13–24), we provide the definition of the `c-lambda` function `vector_addition_scm_driver`.

In this example, `vector_addition_scm_driver` takes ten parameters. On line 14, the parameter list:

```
(int int int int int int int unsigned-int32 scheme-object int)
```

shows types for the parameters, because a `c-lambda` special form only accepts types of its parameters in the parameter list. The last three are actual kernel arguments and the type `unsigned-int32` is the type for the argument `u32_constant`, in line 34. Then, the type `scheme-object` is the type for the argument Scheme vector `u32v_src`, in line 35, and the last parameter `int` type is for the argument vector-length-calculation on line 37

```
(u32vector-length u32v_src)
```

Note that this type information is used during the time of compilation by the Gambit compiler to link with C code.

In the CUDA-C shim, we need to allocate device memory for each vectors; therefore, we need to convert each Scheme vector to its corresponding C type pointer. Gambit provides library functions in C for casting Scheme vectors to C pointers and only allows those library functions in `c-lambda` or `c-declare` constructs. Therefore, the pointer-casting operation must be done within a `c-lambda` or a C-function defined within a `c-declare` construct. Both of these constructs must also be defined in a `.scm` file. Therefore, the pointer-casting operations for the Scheme vectors in this foreign-function interface reside in the Scheme shim's `c-lambda` function in `main.scm` file. Note that this `c-lambda` function calls the CUDA-C shim.

In this example, pointer-casting operations and the call to the CUDA-C shim can also be defined in a C function within a `c-declare` construct. Because Gambit compiler does not allow a C function in a `c-declare` construct to be linked from a different file, that C function cannot have any external linkage. We cannot use the simpler `c-declare` in our implementation because we generate this foreign-function interface in a different file. Instead we put the pointer-casting operations and a call to the CUDA-C shim in a `c-lambda` function, like an interface generated by our implementation. We also put a forward declaration of CUDA-C shim within a `c-declare` construct.

The pointer-casting operation for the Scheme vector of type `scheme-object` to its corresponding Gambit allowed C type is shown on line 18 of Listing 3.2,

```
___U32 *host_u32v_src = ___CAST(___U32*,___BODY_AS(___arg9, ___tSUBTYPED));
```

This operation extracts the pointer of the parameter `___arg9` (referring to the type `scheme-object`, which is the ninth parameter in the parameter list, shown on line 14 of Listing 3.2) from Scheme using macro `___BODY_AS`. Here, a tag `___tSUBTYPED` in `___BODY_AS` specifies that it is a memory allocated object. Next, `___CAST` macro performs the type casting for the extracted pointer to type `___U32` because `___arg9` is a vector of unsigned 32-bits integers. The pointer is then assigned to a `___U32` variable `host_u32v_src`. The C pointer type `___U32` is defined in `gambit.h` along with `___U64`. In this example note that the C pointer type `___U32` is used because the vector contains 32-bit unsigned-integers defined on line 2 of Listing 3.1.

We follow a naming convention to name the cast-pointer for a Scheme vector, and it is

`host_[u, s, f]`*SIZE*`v_[IN\OUT]`*NAME*

The name of a pointer starts with a prefix `host`, followed by the same naming convention used in the vector-

length-calculation helper function. The prefix `host` signifies that it is a host memory pointer in C and needs to be copied to device memory by the CUDA-C shim. Type representation information is also embedded with it. This helps maintenance programmers to recognize the appropriate type in CUDA-C shim for further improvement of our implementation. In this example, the vector `src` is defined in Scheme. In the helper function it becomes `u32v_src` and in the `c-lambda` function it becomes `host_u32v_src`. Note that this consistent changing of names in different parts of Scheme shim helps to clearly identify the appearances of a vector with different names and purposes.

Finally, we call the CUDA-C shim, `vector_addition_cu_driver`, with ten arguments (lines 20–22):

```
vector_addition_cu_driver( ___arg1, ___arg2, ___arg3, ___arg4, ___arg5, ___arg6, ___arg7,
                           ___arg8, host_u32v_src, ___arg10);
```

We pass all the parameters of the `c-lambda` function to the CUDA-C shim. In the argument list, the first seven arguments (`___arg1`, `___arg2`, `___arg3`, `___arg4`, `___arg5`, `___arg6` and `___arg7`) refer to the first seven `int` type parameters in the parameter list on line 14. The eighth argument (`___arg8`) refers to the type `unsigned-int32` in the parameter list. For the Scheme vector, we do not pass the type `scheme-object`, but rather the converted C value `host_u32v_src`. We also pass the length computed by the helper function in Listing 3.2 (line 37) of the converted C type vector. This is the last argument, `___arg10`, which refers to the last parameter type `int` in the parameter list on line 14. Note that the scalar types `int` and `unsigned-int32` shown on line 14 of Listing 3.2 are auto-converted to C by `c-lambda`.

3.2.3 A forward declaration of CUDA-C shim

A forward declaration for the CUDA-C shim function must be given within a `c-declare` construct because the actual definition for the CUDA-C shim resides in file `kernel_driver.cu` along with the CUDA-C kernel forward declaration. In Listing 3.3 our `c-lambda` function calls the CUDA-C shim, which resides in the file `main.scm`. The Gambit-compiler compiles that `main.scm` to a `main.c` . When `nvcc` compiles this `main.c`, and finds a call to the CUDA-C shim, but the definition of the CUDA-C shim is not in `main.c` file, then it gives a link-time error. Therefore, we need the forward declaration for the CUDA-C shim.

In Listing 3.2 (lines 3–8) we provide the code of a forward declaration for the CUDA-C shim function `vector_addition_ cu_driver` within a `c-declare` construct. On line 5

```
void vector_addition_cu_driver();
```

is the forward declaration for the CUDA-C shim.

3.3 Operations of CUDA-C shim

The CUDA-C shim links the Scheme shim to the programmer-supplied CUDA-C kernel by connecting to the `c-lambda` function in the Scheme shim. CUDA-C shim allocates space in device memory for each vector and transfers each vector from host memory to the allocated device memory space. It also defines the variables, specifying execution configuration for a kernel. It then calls the kernel with the actual kernel arguments and the vector lengths. After kernel execution, the CUDA-C shim copies back each vector from device to host memory. Finally, it deallocates space for each vector from device memory.

On lines 7–32 of Listing 3.3 we provide code for the CUDA-C shim[2] `vector_addition_cu_driver` used in this sample foreign-function interface. It resides in the same file (`kernel_driver.cu`) with the forward declaration of CUDA-C kernel `vector_addition` on lines 2–3. The programmers must supply the actual definition of the CUDA-C kernel in a separate `.cu` file. The advantage of this separation is that programmers can change the Scheme source without touching the kernel definition.

We follow a naming convention to name the CUDA-C shim function *`KERNEL-NAME`*`_cu _driver` in out implementation. Here, *KERNEL-NAME* is the name of a kernel, `cu` is for CUDA, and `driver` indicates that the this function calls a CUDA-C kernel. Our implementation generates CUDA-C shims for multiple kernels in the same file, so this naming convention helps programmers to identify a CUDA-C shim for a particular kernel.

In CUDA, because `.cu` files are effectively compiled as C++ files functions within `.cu` files need to be declared as `extern "C"` in order to be visible by C functions in ordinary `.c` files. Therefore, we declare the CUDA-C shim as `extern "C"` on line 7 because it is linked from the compiled `c-lambda` function (the Gambit compiler compiles the c-lambda function into a C function) in a `main.c` file.

In the following sections we describe the operations of the CUDA-C shim `vector_addition_cu_driver`. It is called from the `c-lambda` function in Listing 3.2 (lines 20–22), and it takes ten parameters: `gDx`, `gDy`, `gDz`, `bDx`, `bDy`, `bDz`, `shared_size`, `u32_constant`, `h_u32v_src` and `h_u32v_src_len`. The first six parameters are passed to define the execution configuration and the seventh one is passed to define the size of the dynamic shared memory.

We follow the same naming convention used in vector-length-calculation helper function to name the parameters defining grid and thread blocks in CUDA-C shim. The dimensions of a grid start with `g` for grid, `D` for dimension, and either dimension `x`, `y` or `z` at the end. Similarly, dimensions of thread block start with `b` for block and are followed by `D`, and either `x`, `y`, and `z`. The seventh parameter for shared memory (`shared-size`) is same as it is in the Scheme shim's helper function on line 29 of Listing 3.2.

The eighth parameter, `u32_constant`, is an actual kernel argument and is passed to the CUDA-C kernel. Note that we do not change the name and scalar parameter as it is first used in Scheme shim's helper function

[2]This CUDA-C shim is a generated code by our implementation. We have added some comments and adjusted spaces to improve readability.

on line 29 of Listing 3.2. We follow the exact same naming convention for the scalar parameters as they are in the Scheme shim helper function. For scalar types we do not need to provide extra copy operations in a CUDA-C shim.

```
1  //-----------------------------kernel forward declaration -----------------------
2  __global__  void vector_addition (uint32_t u32_constant, uint32_t* u32v_src,
3                                    int u32v_src_len);
4
5  //-----------------------------Kernel forward declaration-------------------------
6  //-----------------------------CUDA-C shim Start----------------------------------
7  extern "C" {
8   void vector_addition_cu_driver (int gDx, int gDy, int gDz, int bDx,
9                                   int bDy, int bDz, int shared_size,
10                                  uint32_t u32_constant, uint32_t* h_u32v_src,
11                                  int h_u32v_src_len ){
12    //device pointers
13    uint32_t* d_u32v_src;
14    //calculating the size of device memory
15    size_t size_u32v_src = h_u32v_src_len * sizeof(uint32_t);
16    //allocating device memory
17    cudaMalloc((void **) &d_u32v_src, size_u32v_src);
18    //copying host to device
19    cudaMemcpy(d_u32v_src, h_u32v_src, size_u32v_src, cudaMemcpyHostToDevice);
20    //defining Grid configuration
21    dim3 dimGrid(gDx, gDy, gDz);
22    //defining Block configuration
23    dim3 dimBlock(bDx, bDy, bDz);
24    size_t size = shared_size;
25    //Now, at this point it calls the kernel
26    vector_addition <<< dimGrid, dimBlock, size >>> (u32_constant, d_u32v_src,
27                                                     h_32v_src_len);
28    //copying device to host
29    cudaMemcpy(h_u32v_src, d_u32v_src, size_u32v_src, cudaMemcpyDeviceToHost);
30    //deallocation of device memory
31    cudaFree(d_u32v_src);
32   }
33 }
34 //-----------------------------------CUDA-C shim End-----------------------------
```

Listing 3.3: CUDA-C shim links Scheme shim to CUDA-C kernel

The ninth parameter, `h_u32v_src`, is a C pointer type vector for which we cast the Scheme vector in `c-lambda` function in the Scheme shim on line 18 of Listing 3.2. We change the naming convention as it is used in the `c-lambda` function with a slightly change: instead of `host`, it starts with a `h` and followed by the same naming convention as it is used in the Scheme shim's `c-lambda` function as `h_[u, s, f]`*`SIZE`*`v_[IN\OUT]`*`NAME`*. We reduce the size because the C pointer of a vector appears in many operations in CUDA-C shim. We try to fit those operations on single lines to make the program readable.

In this example we can also check types for both scalar and vector types by looking at their names. Here, type for scalar type `u32_constant` is `uint32_t` on line 10; this is consistent with the name. For vector type `h_u32v_src`, its type is `uint32_t*`. It is a pointer because of `*`, and in its name we find a `v` for that.

In this foreign-function interface, we do not pass the C pointer `h_u32v_src` to the CUDA-C kernel because it is a pointer to host memory. In CUDA, a vector in host memory is not passed to the kernel; rather, a vector is copied to device memory and the pointer to that device memory is actually passed to the kernel. Therefore, `h_u32v_src` is available to the kernel. We pass it to the CUDA-C shim for copying the vector to device memory it is pointing, and the pointer to that device memory is passed to the kernel.

The last parameter, `h_u32v_src_len`, is the length of the vector pointed by `h_u32v_src`. It is passed to the CUDA-C shim to allow for correct allocation in device memory where the vector pointed by `h_u32v_src` is copied. It is also not an actual kernel argument. We follow a naming convention to name the parameter defining length of a C pointer. We add the suffix `_len` after the name of the C pointer to help programmers to identify a length associated with its C pointer easily.

Note that we follow consistent changing of name conventions for a Scheme vector passing from Scheme shim to CUDA-C shim. We can clearly understand that `h_u32v_src` is a C pointer of vector `src` which is defined in Scheme on line 2 of Listing 3.1, where `h_u32v_src_len` is its length. On line 7 of Listing 3.1, vector `src` is passed through as a kernel argument to vector-length-calculation helper function as `u32v_src`, on line 29 of Listing 3.2. It becomes `host_u32v_src` on line 18 of Listing 3.2 as a C pointer in `c-lambda` function by extracting Scheme vector `src` before passing it to the CUDA-C shim.

3.3.1 Allocation of device memory for each vector

In the CUDA-C shim we need to allocate device memory for each vector and copy each vector to device memory. Therefore, the C pointer for a Scheme vector, along with its length, is passed through the `c-lambda` function of Scheme shim to this CUDA-C shim. Note that for each vector both host and device memory pointers must be same type and size. The foreign-function interface does the device memory allocation as well as performs data transfer operations between the host and device memory. The advantage here is that programmers only need to define vectors in Scheme.

In Listing 3.2, we generate the following pointer in device memory on line 13

```
uint32_t* d_u32v_src;
```

Our naming convention to name a device pointer is almost like its corresponding host pointer. Instead of `h`, it starts with a `d` for device pointer. It is same as defined in Scheme shim's vector-length-calculation helper function. It also carries type information as in the Scheme shim. The actual name defined in Scheme is `src`. This helps programmers to identify, for which Scheme vector a device memory pointer is declared. It also helps to identify the associated host pointer and cast-pointer in the `c-lambda` function.

We calculate size in bytes to allocate space in device memory as on line 15:

```
size_t size_u32v_src = h_u32v_src_len * sizeof(uint32_t);
```

We use the supplied length (`h_u32v_src_len`), of C pointer (`h_u32v_src`) and the size of information for its type. The variable `size_u32v_src` of type `size_t` is used to allocate space in device memory. We also follow a naming convention to name this variable. It starts with a prefix `size` and is followed by the same naming convention used in Scheme shim's helper function. This naming convention helps the future improvement programmers to identify the device pointer for which this is calculated. In this case, the device pointer is `h_u32v_src`. It also helps to check the host pointer and its corresponding length variable in CUDA-C shim. In this example, the host pointer is `h_u32v_src`, and its length (`h_u32v_src_len`) is used in the size calculation for device memory.

After getting the size, we allocate space in device memory (line 17), denoted by

```
cudaMalloc((void **) &d_u32v_src, size_u32v_src);
```

for the device memory pointer `d_u32v_src` using CUDA-C library function `cudaMalloc`. At this point, device memory is allocated by the host memory data pointed by `h_u32v_src`.

3.3.2 Transferring vectors from host to device memory

Vectors defined in Scheme are actually in host memory. In this example, the vector made available to the kernel is passed through the foreign-function interface to a kernel. In Scheme shim, we can get the length of the vector and the cast C pointers of those Scheme vectors, passing each of them to CUDA-C shim to allocate space in device memory. From here the vectors can be copied from the host to device memory because a CUDA-C kernel can perform data-parallel operations only in device memory.

In Listing 3.3 (line 19), the vector pointed by C pointer `h_u32v_src` is copied to device memory pointed by `d_u32v_src` using CUDA-C library function `cudaMemcpy`.

```
cudaMemcpy(d_u32v_src, h_u32v_src, size_u32v_src, cudaMemcpyHostToDevice);
```

`size_u32v_src` specifies the size in bytes for this copy operation, which is the individual lengths of allocated spaces by these two pointers. `cudaMemcpyHostToDevice` specifies that data is to be copied from host to

device memory.

Host to device memory transfer is not needed for a vector if the notation `OUT` is mentioned after `v_` in the name of that vector. For example, if vector `u32v_src` is written as `u32v_OUTsrc` then the host pointer for vector `u32v_OUTsrc` shim would appear as `h_u32v_OUTsrc` in CUDA-C. Then, the device memory pointer would then appear as `d_u32v_OUTsrc`. Here, `OUT` in specifies that it is not necessary for host memory pointer `h_u32v_OUTsrc` to be copied to device memory pointer `d_u32v_OUTsrc` using library function `cudaMemcpy()` . For this reason, line 19 is not needed for these two pointers `d_u32v_OUTsrc` and `h_u32v_OUTsrc` to avoid host to device memory transfer.

3.3.3 Defining built-in vector data types for an execution configuration

CUDA-C extension provides built-in structures as runtime components. Structures such as `dim3` are used to define variables for specifying dimensions of thread blocks and grid for a kernel execution configuration. In the CUDA-C shim of the example's foreign-function interface, the first six parameters (`gDx`, `gDy`, `gDz`, `bDx`, `bDy` and `bDz`) are passed to define dimensions of a grid and thread blocks.

On line 21 we define x-, y- and z-dimensions of a grid as:

```
dim3 dimGrid(gDx, gDy, gDz);
```

Here, `dim3` type structure `dimGrid` takes three parameters: `gDx`, `gDy`, and `gDz`. These represent x-,y-, and z-dimensions. We name this variable `dimGrid`, where `dim` is for dimension and `Grid` is for a grid of thread blocks. We do not specify any dimension `x`, `y`, or `z`, because it is a `dim3` vector that takes three arguments to define three dimensions of a grid for an execution configuration.

On line 23 we define the dimensions of thread blocks as:

```
dim3 dimBlock(bDx, bDy, bDz);
```

Here, structure `dimBlock` takes three parameters: `bDx`, `bDy`, and `bDz`. These define x-,y-, and z-dimensions. We name this variable `dimBlock`, where `dim` is for dimension and `Block` is for thread blocks. Since it takes three arguments to specify three dimensions and its type is `dim3`, this variable clearly states that it contains dimension of a thread block for an execution configuration.

In the function call in Scheme on line 7 of Listing 3.1,

```
(vector_addition nblocks 1 1 blockSize 1 1 (* (* 2 N) size-int32) constant src)
```

the arguments `nblocks`, and `blockSize` define x-dimensions for a grid and thread blocks, respectively. `dimGrid` and `dimBlock` variables require three dimensions. Therefore, four `1`s are also passed to the CUDA-C shim to define y- and z-dimensions both for a grid and thread blocks.

3.3.4 Calling the kernel with an execution configuration

In this foreign-function interface, a CUDA-C kernel is called with an execution configuration via the CUDA-C shim. We define two `dim3` type variables to specify dimensions of grid and thread blocks. In the parameter list of this CUDA C shim, kernel arguments are `u32_constant` of type `uint32_t` and `h_u32v_src` of type `uint32_t*`. Note that `h_u32v_src` is a host memory pointer, therefore it is not passed to the CUDA-C kernel. Instead, device memory-pointer `d_u32v_src` and size `h_u32v_src_len` of type `int` are passed to the kernel, which has the same type and size of pointer as `h_u32v_src`.

In this example we are also passing the Scheme expression, `((* (* 2 N) size-int32))` on line 7 of Listing 3.1 as the seventh argument to the CUDA-C shim to define the size of the dynamic shared memory. The seventh parameter of CUDA-C shim `shared_size` of type `int` on line 9 of Listing 3.3 contains the size of dynamic shared memory. But we need to assign it to a variable of type `size_t`. Because it is the type to define the size of dynamic shared memory in an execution configuration, we need to assign it to a variable of type `size_t`. On line 24,

```
size_t size = shared_size;
```

assigns variable `shared_size` to variable `size` of type `size_t`. Next, we call the kernel `vector_addition` with the actual kernel arguments `u32_constant` and `d_u32v_src` along with an execution configuration as on line 26:

```
vector_addition <<< dimGrid, dimBlock, size >>> ( u32_constant, d_u32v_src,
                                                  h_u32v_src_len );
```

In execution configuration, `dimGrid` and `dimBlock` define the grid and thread blocks organizations, respectively. `size` defines the size of the dynamic shared memory. At this point, the GPU kernel execution is initiated.

In Listing 3.3 (lines 2–3), we provide a forward declaration for kernel `vector_addition` because it is the definition of this kernel is in another `.cu` file. `vector_addition` takes three parameters: `u32_constant` of type `uint32_t`, `u32v_src` of type `uint32_t*`, and `u32v_src_len` of type `int` . We followed naming conventions here for a scalar type `[u, s, f]`*`SIZE_NAME`* and for a vector type `[u, s, f]`*`SIZE`*`v_[IN\OUT]`*`NAME`*. This naming convention is exactly the same as was used in the Scheme shim's vector-length-calculation helper function. It helps programmers to identify which arguments are passed through from Scheme to a kernel.

Although the name of length parameter `u32v_src_len` does not reflect its type, it is identifiable that `u32v_src_len` is the length for vector `u32v_src`.

3.3.5 Transferring results from device to host memory

When a kernel finishes its execution, the CPU may need the result. Since a CUDA-C kernel performs data-parallel operations only in device memory, we need to transfer resultant data from device to host memory.

We already know that CUDA-C provides library function `cudaMemcpy` for the memory transfer operations.

In Listing 3.3 (line 29),

```
cudaMemcpy(h_u32v_src, d_u32v_src, size_u32v_src, cudaMemcpyDeviceToHost);
```

`cudaMemcpy` copies data to host memory pointed by `h_u32v_src` from device memory pointed by `d_u32v_src`. CUDA memory copy type, `cudaMemcpyDeviceToHost` specifies that data is to be copied to host from device memory. The argument `size_u32v_src` is the size of both vectors. Note that the host pointer `h_u32v_src` represents the Scheme vector `src`. After the kernel finishes its execution, resultant data is copied from device to host memory pointed by `h_u32v_src`. In Scheme, host pointer `h_u32v_src` is actually the vector `src`, so we can access the result from Scheme as shown on line 8 of Listing 3.1.

Device to host memory transfer is not needed for a vector if the notation `IN` is mentioned after `v_` in the vector name. For example, device memory pointer `d_u32v_INsrc` for vector `u32v_INsrc` will not be copied to host memory pointer `h_u32v_INsrc` after the kernel execution using CUDA library function `cudaMemcpy()` . Line 29 is therefore not needed for these two pointers (`d_u32v_INsrc` and `h_u32v_INsrc`) to avoid device to host memory transfer.

3.3.6 Freeing device memory

In this foreign-function interface, a device memory space pointed by a device memory pointer is freed after copying data to host memory because the device memory copy is no longer needed. CUDA provides the library function `cudaFree` to free device memory. In this example, we free the device memory by passing the device memory pointer `d_u32v_src` to `cudaFree` on line 31 of Listing 3.3:

```
cudaFree(d_u32v_src);
```

In CUDA-C, the same device memory pointer can be reused in an another kernel, but we do not send the same device memory pointer to an another kernel in our implementation. A more sophisticated Scheme shim might recognize repeated use of a Scheme vector and reuse device memory. Therefore, if we want to pass the same Scheme vector to a different kernel, it must be passed through the foreign-function interface of that kernel. In this case, another C pointer of that vector is copied to a different device memory space pointed by a different device memory pointer from that interface.

Our implementation in Gambit Scheme compiler generates both shims to link Scheme to CUDA-C kernels. The whole linking process is transparent to the programmers. Therefore, they do not need to see the both generated shims. However, our implementation also allows programmers to retain both shims as temporary files for debugging purpose after building a GPU executable. Since the shims are transparent and any intentional changes in the temporary generated shims do not affect an executable, we do not use the `constant`

keyword to declare the generated parameters and variables as constants in both shims.

3.4 Gambit command-line options to build a GPU executable

For this example we compiled three files - – `main.scm`, `kernel_driver.cu`, and `kernel.cu` — through the Gambit compiler `gsc` to use the CUDA C compiler `nvcc`, for building a GPU executable. In order to do this, we extended the Gambit compiler so that it accepted the `.cu` file extension. Here, `gsc` compiles Scheme code to C code. Then it forwards C code internally to the `nvcc` compiler; then `nvcc` compiles generated C code to object code. Therefore, `gsc` is also a compiler driver since it forwards generated C code to `nvcc` internally. Both a CUDA-C shim and the CUDA-C kernel must be in `.cu` files. In this example, CUDA C shim is in `kernel_driver.cu` and the kernel definition is in `kernel.cu`. We also identify `gsc` command-line options for passing appropriate compiler flags to `nvcc` compiler.

`gsc` uses command-line options `-o` and `-exe` to build an executable. We need to use `gsc` command-line options `'-cc-options'` to pass the directories of header files, and `'-id-options-prelude'` to pass the directories of runtime libraries of both Gambit and CUDA to `nvcc` compiler.

In order to compile this example program, `gsc` first compiles `main.scm` to `main.c` containing the compiled C code. `gsc` also generates the incremental link file `main_.c` by implicitly invoking option `-link`. `gsc` also implicitly invokes option `-obj` to compile `main.c`, `main_.c`, and `kernel_driver.cu` to objects files `main.o`, `main_.o`, and `kernel_driver.o`. This is done by implicitly invoking `nvcc`, as `nvcc` invokes the GNU `gcc` compiler. Finally, `gsc` invokes `nvcc` to compile `main.o`, `main_.o`, `kernel_driver.o`, and `kernel.o` to generate GPU executable.

```
1  GAMBIT_DIR = location of the Gambit installation directory
2  CUDA_DIR = location of the CUDA installation directory
3  CC = $(GAMBIT_DIR)/bin/gsc
4  INCLUDE_DIR = "-I $(CUDA_DIR)/include -I $(GAMBIT_DIR)/include"
5  LIB_DIR = "-L $(CUDA_DIR)/lib64 -lcuda -lcudart -L $(GAMBIT_DIR)/lib"
6  SRCS = main.scm kernel_driver.cu kernel.cu
7  EXE = kernel.exe
8
9  $(EXE): $(SRCS)
10         $(CC)
11         -cc-options $(INCLUDE_DIR)
12         -ld-options-prelude $(LIB_DIR)
13         -keep-c -o  $(EXE) -exe  $(SRCS)
```

Listing 3.4: A `makefile` to build a GPU executable through the extended Gambit compiler

In Listing 3.3 we provide the command-line options in a `makefile` to compile `main.scm`, `kernel_driver.cu`, and `kernel.cu` into a GPU executable (`kernel.exe`). The code for `main.scm` is shown in Listings 3.1 and

3.2, and code for `kernel_driver.cu` shown in Listing 3.3. Note that we do not show the definition of kernel `vector_addition` in file `kernel.cu`.

We use constants to generalize this `makefile`. On line 1 of Listing 3.3, constant `GAMBIT_DIR` specifies the path of the Gambit installation directory. On line 2, `CUDA_DIR` specifies the CUDA installation directory. Programmers must supply these two paths according to their installation destination.

On line 3, `CC` specifies path for the Gambit compiler executable `gsc` as

```
CC = $(GAMBIT_DIR)/bin/gsc
```

On line 4, `INCLUDE_DIR` specifies the directories of header files CUDA, `$(CUDA_DIR)/include`, and Gambit `$(GAMBIT_DIR)/include`

```
INCLUDE_DIR = "-I $(CUDA_DIR)/include -I $(GAMBIT_DIR)/include"
```

Here, C compiler flag `-I` is used to indicate the paths for header files. Directories of runtime libraries are specified by `LIB_DIR` on line 5

```
LIB_DIR = "-L $(CUDA_DIR)/lib64 -lcuda -lcudart -L $(GAMBIT_DIR)/lib"
```

Here, `$(CUDA_DIR)/lib64` is the directory for the CUDA runtime library and `$(GAMBIT_DIR)/lib` is the directory for Gambit. `-lcuda` and `-lcudart` are provided to link objects files `main.o`, `main_.o`, `kernel.o`, and `kernel_driver.o` with library files `libcuda.dylib` and `libcudart.dylib` in standard library directory. C flag `-L` is used to specify the locations of runtime libraries. On line 6, `SRCS` is used to specify the sources `main.scm`, `kernel_driver.cu`, and `kernel.cu`. On line 7, `EXE` specifies the GPU executable `kernel.exe`.

The options `-o` and `-exe` on line 13 of Listing 3.4 tell `gsc` compiler `$(CC)` on line 10 to build the executable `$(EXE)` from the sources `main.scm`, `kernel_driver.cu` and `kernel.cu` (represented by `($SRCS)`) by implicitly invoking `nvcc`.

On line 11, command-line option

```
-cc-options $(INCLUDE_DIR)
```

passes both directories of header files for the Gambit compiler and CUDA to `nvcc`. On line 12, command-line option

```
-ld-options-prelude "$(LIB_DIR)"
```

passes the directories of runtime libraries for Gambit and CUDA to C linker. This is also invoked by `nvcc`.

Gambit command line option `-keep-c` on line 13 of Listing 3.4 keeps temporary files `main.c` and `main_.c` generated by `gsc`.

Note that this `makefile` generates a GPU executable from Scheme with the provided foreign-function interface and a provided CUDA-C kernel. It cannot handle the generation of the foreign-function interface from Scheme. For our implementation, we provide a template `makefile` that generates the foreign-function interface from a Scheme binding. Then, it builds a GPU executable from the provided CUDA-C kernels and the C files generated by Gambit.

In Chapter 1, we discussed how the OpenCL framework that supports diverse data-parallel platforms including CUDA GPUs, some ATI GPUs, multi-core CPUs from Intel and AMD. OpenCL shares some core ideas with CUDA. Both of them have similar platform models, memory models, execution models and programming models [5, 11]. Like CUDA, the OpenCL programming model also consists of a host, and one or more devices that are massively parallel processors. It also has similar device memory spaces such as global memory, constant memory, local memory (termed as shared memory in CUDA), private memory (termed as local memory in CUDA). Moreover, syntax for various keywords and built-in functions are almost identical to each other. A GPU thread in OpenCL is called a *work item* and a thread block is called a *work group* [32]. In OpenCL, it is also required to define an execution configuration to lunch a kernel like CUDA. Programmers also need to manage memory transfer operations between host and device memory.

Hence, it is also possible to link Scheme or other functional languages to data-parallel OpenCL by generating shims similar to those discussed earlier in this chapter. In order to do this, high-level languages should have foreign-function interface support, like Gambit's `c-lambda` or `c-declare` constructs, to link data-parallel OpenCL code. Moreover, the shims should also contain data-type mapping-operations from high-level languages to low-level OpenCL, memory-transfer operations between host and device memory, kernel-call primitives in OpenCL with execution configuration, and allocation/deallocation operations.

3.5 Summary

In this chapter we show that a CUDA-C kernel can be linked from Scheme by a foreign-function interface comprised of both Scheme and CUDA-C constructs. In our example, when a kernel is called from Scheme this, in fact, calls the Scheme shim of our foreign-function interface is actually called. It calculates lengths of vectors, converts data from Scheme to C types, casts C pointers for Scheme vectors, and calls the CUDA-C shim of our foreign-function interface. A CUDA-C shim allocates space in device memory and transfers data between host to device memory for each vector. It also defines CUDA built-in vectors for the execution configuration and calls the kernel along with execution configuration.

This example offers the necessary parts of a foreign-function interface we want to generate. For our implementation, we provide new special forms in Scheme for GPU computation. We also follow a strict naming convention to name the parameters of a kernel binding in Scheme. This is because Scheme does not provide type information, whereas CUDA-C requires type information for its data types. Therefore, we need to extract type information for kernel parameters from Scheme. While we can use type-inference techniques

to infer the type of a kernel's parameter, it may not be time- and cost-effective for this project. In our implementation, acceptable prefixes for kernel parameters are as follows:

- `u8[v]_[IN\OUT]`*NAME* for 8-bit unsigned-integers
- `s8[v]_[IN\OUT]`*NAME* for 8-bit signed-integers
- `u16[v]_[IN\OUT]`*NAME* for 16-bit unsigned-integers
- `s16[v]_[IN\OUT]`*NAME* for 16-bit signed-integers
- `u32[v]_[IN\OUT]`*NAME* for 32-bit unsigned-integers
- `s32[v]_[IN\OUT]`*NAME* for 32-bit signed-integers
- `u64[v]_[IN\OUT]`*NAME* for 64-bit unsigned-integers
- `s64[v]_[IN\OUT]`*NAME* for 64-bit signed-integers
- `f32[v]_[IN\OUT]`*NAME* for 32-bit floating-point numbers
- `f64[v]_[IN\OUT]`*NAME* for 64-bit floating-point numbers

Including `v` after the size of a number is optional and is used only for vector types. In this chapter, we also identify necessary command-line options of the Gambit compiler to compile the foreign-function interface through the `nvcc` compiler.

Chapter 4

Implementation

In this chapter we discuss our implementation using Gambit to link a CUDA-C kernel from Scheme. Our implementation generates the interface from a Scheme binding of a kernel skeleton that contains the name of the kernel and the parameters that have type information in their names.

We begin this chapter with a discussion of new, special constructs for our implementation in Gambit for GPUs. Second, we describe how to extract necessary information from those constructs in the form of parse tree nodes to generate the foreign-function interface. Third, we discuss how to generate of the foreign-function interface by using this extracted information. Fourth, we demonstrate how to convert a special construct for calling a kernel into an ordinary Scheme function call that links the generated foreing-function-interface and a CUDA-C kernel. Fifth, we also discuss some useful library functions in Scheme for GPUs that enables Scheme programmers to manage GPUs from Scheme. Finally, we describe a template `makefile` that can manage interface generation from a Scheme bindings of a kernel skeleton and build a GPU executable from a provided CUDA-C kernel.

4.1 Special forms for GPU

Our implementation in Gambit generates interfaces from Scheme bindings that call CUDA-C kernels. These Scheme bindings must contain special keywords that enable Gambit compiler to generate the interfaces. Therefore, we provide some special forms in Scheme for GPU computation in Gambit. These special forms differentiate new, special forms for GPU computations from normal Scheme special forms for CPU computations.

4.1.1 Special form for a kernel

In CUDA, kernels are used for extensive data-parallel operations on GPUs. We provide a special form in Scheme to define a kernel, as follows:

```
(define kernel-name
  (kernel (parameter-list ...)
    expression ...))
```

In our implementation, a kernel skeleton is a top-level definition because it must be visible for other

Scheme constructs to call it. Here, *kernel-name* is the name of a kernel. The keyword `kernel` specifies that this is a special form that runs on GPUs. Next, *parameter-list* defines the parameters it takes. In this thesis, we follow a naming convention for the kernel parameters. *expression* defines a sequence of Scheme expressions in its body. Note that our implementation in Gambit does not compile the body of a kernel. Gambit can recognize the body of a kernel definition and extract the body of that kernel but it does not generate code for it. Therefore, Scheme programmers do not need to provide *expression* and define the body of a kernel skeleton. We provide this body *expression* as future work for this thesis.

In CUDA-C, keyword `__global__` specifies a C function as a kernel that runs on GPUs and is only callable from host functions. However, global as a term in programming language is usually used to refer to an object that has global scope; this does not reflect the idea that it is a kernel or a special function that runs on GPUs. Therefore, we prefer the keyword `kernel` over `global` in a top-level definition to specify a kernel in Scheme for Gambit.

In our implementation, kernel definitions are similar to `lambda` definitions in Gambit, but with a keyword change from `lambda` to `kernel` :

```
(define kernel-name
  (lambda (parameter-list ...)
    expression ...))
```

In Gambit, `lambda` special form is used to define a procedure that can be called from other Scheme expressions. Similarly, in CUDA-C a kernel is a special C-function that can be called from other C-functions. Therefore, we chose the structure of `lambda` special form to define a kernel in Gambit.

4.1.2 Special form for a device function

The special form for a device function is similar to a `lambda` special form in Scheme:

```
(define device-function-name
  (device (parameter-list ...)
    expression ...))
```

Here, *device-function-name* is the name of a device function. The keyword `device` denotes that this is a device function. In CUDA-C, the keyword `__device__` is used to define a device function that runs on a GPU but is only callable from kernels. We chose the keyword `device` to define a device function in Scheme. Here, *parameter-list* defines the parameters passed from a kernel. The parameters of a device function also follow the same naming convention used to name kernel parameters. *expression* defines the body of a device function.

For now, Gambit can accept a device function defined in Scheme. Gambit then extracts the parse tree nodes constructed from the device function. Our implementation does not generate the Scheme and CUDA-C shims for a device function. Note that a host cannot call a device function; a device function is only callable from kernels. This special form is an area for future work for our implementation in Gambit.

4.1.3 Special form for calling kernel with execution configuration

We provide a special form in Scheme to call a kernel. Every kernel call must have an execution configuration that defines the dimensions of a grid and thread blocks. It also defines a size for dynamic shared memory. The following is a kernel call special form in Scheme:

```
(symbol <<< (exp [exp exp]) (exp [exp exp]) [exp] >>> arg ... )
```

Here, *symbol* denotes the identical name of a CUDA-C kernel which is being called. Note that *symbol* must not have any special character that are to `nvcc` compiler (e.g., space, #, ?, –).. Next, the expression,

```
<<< (exp [exp exp]) (exp [exp exp]) [exp] >>>
```

defines the execution configuration for a launching kernel. There are three parts to the execution configuration. The first two parts - `(`*exp* `[`*exp exp*`])` and `(`*exp* `[`*exp exp*`])` - are mandatory for defining the dimensions of a grid and thread blocks. The third part `[`*exp*`]` is optional and defines the size of dynamic shared memory. `(`*exp* `[`*exp exp*`])` defines dimensions of a grid. Its first *exp* is mandatory, as it defines x-dimension of a grid, but the next two two *exp*s (`[`*exp exp*`]`) are optional. These define the y- and z-dimensions of a grid. Similarly, the second part `(`*exp* `[`*exp exp*`])` defines the x-,y-, and z-dimensions of a thread block and follows the same constructions used for a grid. Note that enclosed `[ ]` symbols are not part of our provided kernel call construct, but separate optional from mandatory expressions. Next, *arg* ... represents the kernel arguments that are passed to a kernel.

In the following, we provide some allowed execution configurations for our implementation:

- Defining only x-dimensions of a grid and thread blocks

  ```
  ( symbol <<< ( exp )( exp ) >>> arg ... )
  ```

- Defining x-dimension of a grid and x-, y-dimensions of thread blocks and dynamic shared memory

  ```
  ( symbol <<< ( exp )( exp exp ) exp >>> arg ... )
  ```

- Defining x-, y-dimensions of a grid and x-dimension of thread blocks

  ```
  ( symbol <<< ( exp exp )( exp ) >>> arg ... )
  ```

- Defining x-, y-dimensions of a grid and x-,y-, and z-dimensions of thread blocks

  ```
  ( symbol <<< ( exp exp )( exp exp exp ) >>> arg ... )
  ```

- Defining x-, y-, and z-dimensions of a grid, thread blocks, and dynamic shared memory

  ```
  ( symbol <<< ( exp exp exp )( exp exp exp ) exp >>> arg ... )
  ```

Note that *exp*s used in an execution configuration must return integer values.

4.1.4 Synchronization of multiple kernel executions

In CUDA-C, sometimes the execution of multiple kernels may need to be synchronized. Therefore, we provide special form `sync` to synchronize executions of multiple kernels. It takes kernel calls as its arguments.

```
(sync (k1 <<< (exp [exp exp]) (exp [exp exp]) [exp] >>> arg-list...)
      (k2 <<< (exp [exp exp]) (exp [exp exp]) [exp] >>> arg-list...)
      ...)
```

Here, execution of kernels *k1* and *k2* are synchronized. *k2* and the kernels following *k2* will not start until all participating threads running *k1* are finished. This way, kernels following *k2* will not start until *k2* is finished. Note that `sync` can take just one kernel call as argument. In that case, operations in host following that kernel will be blocked until device has completed the execution of that kernel. Note that the execution of multiple kernels without `sync` is by default, asynchronous. We also provide a special form `async` to specify the kernels that are not synchronized. `async` takes kernel calls as arguments.

```
(async (k1 <<< (exp [exp exp]) (exp [exp exp]) [exp] >>> arg-list...)
       (k2 <<< (exp [exp exp]) (exp [exp exp]) [exp] >>> arg-list...)
       ...)
```

Here, the execution of *k1* and *k2* is independent. It is not mandatory to use special form `async` to specify kernels that are not synchronized, but this helps programmers to clearly distinguish synchronized kernels from non-synchronized kernels.

4.2 Naming kernel parameters with the constant prefixes

Scheme is a dynamically-typed language. Therefore, Gambit compiler does not check for type compatibility at compile time. It does not have a type-inference system that can automatically deduce the type of an expression. In this thesis, we link Scheme code to CUDA-C kernels, where CUDA-C is a typed language. Therefore, we must supply type information for kernel parameters to Gambit in order to generate appropriate types for them in both the Scheme and CUDA-C shim that link a CUDA-C kernel. In order get type information from Scheme we follow a strict naming convention with constant prefixes for kernel parameters. Gambit compiler can extract type information for kernel parameters from their names, check them, and generate appropriate type information to be used by `nvcc` .

In our implementation, constant prefixes for kernel parameters' names help programmers to identify types easily. It also enables programmers to follow a standard so that others can easily read and understand type information. These constant prefixes help programmers to guess the generated operations both in Scheme and CUDA-C shims, as they depend strictly on types.

In our implementation, a vector as a kernel argument passes through the generated foreign-function interface to a CUDA-C kernel. In Scheme shim, a pointer-casting operation for that vector is generated to extract its C-pointer of that vector. This pointer-casting operation depends on the C-type for that vector. In

CUDA-C shim, operations such as the declaration of a device memory pointer, allocation of space in device memory, and transfer operations between host and device memory are also generated for that vector. All of these operations depend strictly on the type of vector extracted from a constant prefix attached to the vector's name.

Moreover, there are some other reasons to use constant prefixes to name kernel parameters in our implementation. First of all, suppose we want to pass a Scheme vector of 64-bit unsigned integers to a kernel. The vector must pass through the generated foreign-function interface of that kernel. In that interface, all the operations related to this vector must be compatible with its 64-bit unsigned-integer type. Suppose, Gambit generated code for 32-bit signed-integers instead of 64-bit unsigned-integers due to a lack of available type information regarding size in bits for the elements of that vector. In that case, allocated integers in that vector may contain values that are out of the range for 64-bit signed-integers and integer overflow may occur.

Second, we can also save space in device memory by specifying the size of vector elements. For example, if we want to pass a vector of 8-bit unsigned integers of ten elements to CUDA-C kernel, then Gambit should generate code for allocating space in device memory for 8-bit unsigned-integers with ten elements. However, due to the a lack of size information for the vector elements, Gambit instead generates code to allocate 32-bit integers. Therefore, instead of allocating 10 bytes of space, 40 bytes of space is allocated in device memory, and 30 bytes of device memory would be unused. It is preferable, then, to generate appropriate size in memory allocation code for a vector to reduce unnecessary device memory consumption. Moreover, a vector of 40 bytes would take more time than a vector of 10 bytes to be transferred between host and device memory. This may affect the overall execution time and create overhead.

Moreover, following a strict naming convention for data types makes it easier for programmers to understand types and sizes of variables. The implementation of these constant prefixes in Gambit is easy. However, it is less flexible to change which may sometimes be irritating to programmers.

We can implement a type-inference system in Gambit in order to get types for kernel parameters. In that case, programmers do not need to use constant prefixes to name kernel parameters. However, implementation of a type-inference system is much harder compared to the implementation of constant prefixes in Gambit. Therefore, we chose constant prefixes to name kernel parameters in order to get their type information.

On line 2–3 of Listing 4.1 we provide an example of a kernel skeleton binding in Scheme for kernel `vector_addition`. The keyword `kernel` specifies that it is a binding for a kernel. This kernel takes two arguments - `u32_constant` and `u32v_src` - specified with constant prefixes on line 3. Here, `u32_constant` is a scalar type of 32-bit unsigned-integer and `u32v_src` is a vector of 32-bit unsigned-integers, as defined on line 6. The kernel skeleton does not have any *`expression`* as its body because our extension in Gambit does not compile the body of a kernel defined in Scheme. Our implementation requires a skeleton of a kernel in Scheme to generate both shims that link a kernel implemented in CUDA-C.

This kernel's execution configuration is called, on lines 12–14 with with two arguments (`constant` and `src`):

```
(vector_addition <<< (nblocks) (blockSize) (* (* 2 N) size-int32) >>>
                 constant
                 src)
```

In the execution configuration, `nblocks` specifies the organization of the grid; `blockSize` specifies the size of the participating thread blocks; and `(* (* 2 N) size-int32)` determines the size of dynamic shared memory.

```
1  ;;Binding for a kernel skeleton
2  (define vector_addition
3    (kernel (u32_constant u32v_src)))
4
5  (define N 100)
6  (define src (make-u32vector N 0))
7  (define constant 786)
8  (let ((nblocks 1)
9        (blockSize N)
10       (size-int32 4))
11                      ;; Grid-x     Block-x       shared memory
12   (vector_addition <<< (nblocks) (blockSize) (*(* 2 N)size-int32) >>>
13                    constant ;;scalar type
14                    src)     ;;vector type
15   (display src))
```

Listing 4.1: Binding of a kernel skeleton and a call to that kernel with execution configuration special form

Our implementation generates the Scheme shim and CUDA-C shim from a kernel skeleton. Therefore, we replace the Scheme shim shown in Listing 3.2 of Chapter 3 as file `main.scm`, with the kernel skeleton of `vector_addition` on lines 2–3 of Listing 4.1. Our implementation generates the Scheme shim (Listing 4.2) and the CUDA-C shim (Listing 4.3) from this kernel skeleton.

The kernel call on line 7 of Listing 3.1 is also changed with the new special form for calling a kernel in Scheme, shown on lines 12–14 of Listing 4.1. Our implementation converts this special form to an ordinary function call that calls the Scheme shim to link the CUDA-C kernel.

4.3 Recognizing GPU special forms

We provide some special forms in Gambit Scheme as device syntax for device computation. Gambit can check Scheme source expressions for GPU and builds a list of parse tree nodes from those Scheme expressions.

We add the extra flag `-cu` to Gambit in order to check for device syntax and generate the shims. This flag give a hint to Gambit to check for device syntax in a source file. Without this flag Gambit has to check

all Scheme expressions for device syntax every time, even though a source file may not contain any GPU special forms. This flag separates device syntax from ordinary Scheme syntax.

In order to generate the foreign-function interface, our implementation requires the name of a kernel and the names of its parameters with type information. This information can be extracted from the parse trees constructed by Gambit from Scheme source expressions defining a kernel/device function.

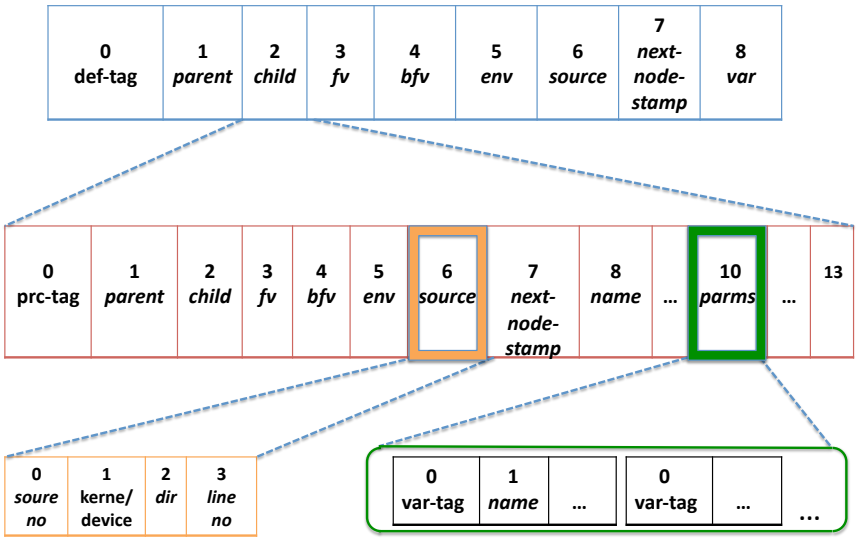


Figure 4.1: Pictorial representation of a parse tree constructed from kernel/device function in Scheme

4.3.1 Recognizing kernel/device symbols

Gambit checks the syntax of Scheme source expressions. If the syntax is correct, then it constructs a list of parse trees from that source. In order to check for a kernel or device function we need to access the parse tree nodes. First, we need to check for the `define` symbol, then we need to check for the `kernel/device` symbol. In Scheme, the keyword`define` is used to define a procedure or a global variable. In this thesis, kernel/device function must be a procedure. Therefore, we also need to check in parse trees for a node constructed from a procedure.

In Figure 4.1 we show parse tree nodes generated by Gambit from a kernel/device function. In Gambit, a Scheme object defined with the keyword `define` makes a parse tree node (usually represented as a vector) with nine fields. It starts with a tag called def-tag in field 0 to represent a node which is defined with `define` in a Scheme source expression. This is shown in Figure4.1 with a blue-bordered table with nine fields. Field

1 of that node parent represents a node if it is a sub-expression of a parent node. In field 2, child represents a list of nodes containing sub-expressions. Field 3, fv, represents a list of free variables contained in this expression. Field 4, bfv, represents a list of free variables with binding used in this expression. Field 5, env, is the environment of this node. In field 6, source represents the source of this node. In field 7, next-node-stamp contains an integer which is used to identify the next node. In field 8, var contains the name of the identifier.

```
;;AST -> boolean
;;returns true if the node indicate a kernel/device-function
(define (is-kernel/device-function node)
  (and (vector? node)
       (eqv? 'def-tag (car (vector-ref node 0)))
       (= 8 (vector-length node))
       (let ([child (car (vector-ref node 2))])
         (and (vector? child)
              (eqv? 'prc-tag (car (vector-ref child 0)))
              (= 13 (vector-length child))
              (let ([source (vector-ref child 6)])
                (and (vector? source)
                     (= 4 (vector-length source))
                     (let ([tag (vector-ref source 1)])
                       (or (eqv? 'kernel tag)
                           (eqv? 'device tag)))))))))
```

Listing 4.2: Scheme code snippet to access kernel/device symbol in the parse tree nodes constructed by Gambit

In order check for a `kernel/device` symbol, we access the child in field 2 of `def-tag` node. This node is represented as a red table with thirteen fields in 4.1. It starts with the tag `prc-tag` in field 0; this represents that this node is constructed from a Scheme source expression, which is a procedure. This node also contains the fields parent, child, fv, bfv, env, source, next-node-stamp, name, and others. Field 6, `source`, is represented by an orange table with four fields and contains the keyword `kernel/device` in field 2. In field 0, source no, contains the command-line argument number for a source file. directory in field 2 is the location of the source file, and line no in field 3 represents the line number of `kernel/device` symbol in the source file. In this way, we can check for a `kernel/device` symbol in a parse tree node constructed from a Scheme expression defining a kernel/device function. In Listing 4.2 we provide Scheme code snippet to access `kernel/device` symbol in the parse tree nodes generated from kernel/device function source expressions.

4.3.2 Extracting kernel/device function name

After checking for a `kernel/device` symbol in a parse tree node starting with a def-tag, our implementation extracts the name of a kernel/device function from that node. In Figure 4.1, we show how to extract the

name of a kernel/device function. First, we access the node with `def-tag`. In this node, field 2 contains the node starting with a `prc-tag`. In that node field 8 contains the name of a kernel/device function. In Listing 4.3 we provide a Scheme code snippet that returns the name of a kernel/device function from parse tree nodes.

```
1 ;;AST->symbol
2 (define (kernel/device-function-name  node)
3   (cond ((is-kernel/device-function node)
4          (let ([child (car (vector-ref node 2))])
5            (vector-ref child 8)))))
```

Listing 4.3: Scheme code snippet to extract the name of a kernel/device function from the parse tree nodes

In Listing 4.1 for the kernel skeleton `vector_addition`, our implementation extracts the name `vector_addition` from the parse tree nodes constructed from the Scheme source expression on lines 2–3.

4.3.3 Extracting parameter list

In order to generate the shims from a kernel skeleton, our implementation also extracts the names of kernel parameters from the parse tree. Figure 4.1 shows where to extract parameters from parse tree nodes generated from a kernel/device function. In the `def-tag` node, child (field 2) contains the `prc-tag` node. Field 10 (parms) of `prc-tag` node contains the list of parameters of a kernel/device function represented by a green rounded rectangle.

```
1  ;;AST -> list
2  (define (get-parameters node)
3    (cond ((is-kernel/device-function node)
4           (let* ([child (car (vector-ref node 2))])
5                 ([parms (vector-ref child 10)])
6             (map (lambda (var)
7                    (cond ((is-var-tag var)
8                           (vector-ref var 1))
9                          (else
10                          (error "This is not a variable."))))
11                parms)))))
12 ;;AST -> boolean
13 (define (is-var-tag node)
14   (and (vector? node)
15        (= 11 (vector-length node))
16        (eqv? 'var-tag (car (vector-ref node 0)))))
```

Listing 4.4: Scheme code snippet to extract the names of kernel/device function parameters

Each parameter contained in a node starts with a tag `var-tag` with eleven fields (only the fields `0` and `1` are shown in Figure 4.1), represented by each black table contained within the green rounded rectangle. In field `1` of a `var-tag` node, the `name` contains the symbol that denotes the variable. In Listing 4.4, we provide a Scheme code snippet that extracts the kernel parameters from the parse tree nodes and returns a list that contains the extracted names.

In Listing 4.1, kernel skeleton `vector_addition` takes two parameters: `u32_constant` and `u32v_src`. Gambit builds two `var-tag` nodes and puts them into field `10` of the `prc-tag` node. Our implementation extracts the first parameter `u32_constant` from field `1` of the first `var-tag` node. Similarly, the second parameter `u32v_src` is extracted from field `1` of the second `var-tag` node. These parameters also contain type information in their symbols.

4.4 Generating Scheme shim

Our implementation checks for a kernel/device function in a list of parse tree nodes. After recognizing a kernel/device function the name of that kernel/device function and the parameters are extracted from the parse tree nodes constructed by Gambit.

In order to generate the Scheme shim, the name of a kernel and its parameters with type information are required. In Listing 4.5 we provide the generated Scheme shim from the kernel skeleton `vector_addition`.

In our implementation, calling a kernel is actually calling the Scheme shim of the interface that links to that kernel. In Scheme shim, the vector-length-calculation helper function calculates the length of a vector which is a kernel argument. Therefore, we need to generate this helper function. The name of this helper function is identical to the kernel skeleton. Our implementation uses the extracted name of a kernel to generate the name of this helper function. This helper function has seven extra parameters for execution configuration. These seven extra parameters are added with the extracted parameters to generate the parameter list. Our implementation generates the helper function, shown on lines 32–37 of Listing 4.5, from the kernel skeleton `vector_addition`. In the parameters list on line 29, `gDx`, `gDy`, `bDx`, `bDy`, `bDz`, and `shared-size` are generated to take the execution configuration parameters. The actual kernel arguments (`u32_constant`, and `u32_src`) are then extracted from the parse tree nodes and appended to the parameter list.

This helper function also calls the `c-lambda` function to convert data types from Scheme to C. This function call passes all helper function's parameters to a `c-lambda` function. It also passes the length for each vector by adding a length calculation expressions for each vector. On lines 32–37, the `c-lambda` function call is generated. The extracted kernel name `vector_addition` is appended with the suffix `_scm_driver` on line 32 in order to generate the name `vector_addition_scm_driver` for the `c-lambda` function.

In order to generate the argument list for the `c-lambda` function call, our implementation generates the seven execution configuration parameters on line 33. Next, it appends the actual extracted kernel parameters to the argument list and adds the appropriate length calculation expression to that list for each vector.

```
;;-------------------------------Scheme shim Start---------------------------
;; A c-declare construct to provide a forward declaration of the CUDA C shim.
(c-declare #<<c-declare-end
        //forward declaration for CUDA C shim
        void vector_addition_cu_driver();

c-declare-end
)
;;the c-lambda function performs data type conversion and calls the CUDA C shim
;;vector_addition_scm_driver :: int * int * int * int * int *
;;                              int * int * uint32_t *
;;                              u32vector * int -> null
(define vector_addition_scm_driver
  (c-lambda (int int int int int int int unsigned-int32 scheme-object int)
            void
#<<c-lambda-end
        //casting to C pointer
        ___U32* host_u32v_src =___CAST(___U32*,___BODY_AS(___arg9,___tSUBTYPED));
        //calling the CUDA-C shim
        vector_addition_cu_driver( ___arg1, ___arg2, ___arg3, ___arg4, ___arg5,
                                   ___arg6, ___arg7, ___arg8,
                                   host_u32v_src, ___arg10);
c-lambda-end
))
;;vector-length-calculation helper function
;;vector_addition :: int * int * int * int * int *
;;                   int * int * uint32_t *
;;                   u32vector -> null
(define (vector_addition gDx gDy gDz bDx bDy bDz shared-size u32_constant u32v_src)
;;calling the c-lambda function vector_addition_scm_driver along  with the
;;vector-length-calculation  as an argument
  (vector_addition_scm_driver
    gDx gDy gDz bDx bDy bDz shared-size
    u32_constant
    u32v_src
    ;;vector-length-calculation function
    (u32vector-length u32v_src)))
;;-----------------------Scheme shim End-------------------------------------
```

Listing 4.5: Generating the Scheme shim from the kernel skeleton `vector_addition`

In order to generate the length calculation expression, our implementation extracts type information from a vector's name. For this example, `u32v_src` is a kernel parameter. It contains the symbol `v` identifying it as a vector. Type information for this vector can also be identified from its name. Here, `u32` specifies that it is a vector of 32-bit unsigned-integers. Therefore, our implementation can generate its corresponding vector length calculation expression `(u32vector-length u32v_src)`, shown on line 37.

Our implementation generates the `c-lambda` function on lines 13–24. Its parameter list containing type information is also generated, as shown on line 14. Our implementation generates default type `int` for the seven execution configuration parameters. Foreign object type for the first kernel parameter `u32_constant` is generated as `unsigned-int32` and foreign object type `scheme-object` is generated for the second kernel parameter vector (`u32v_src`). Finally, parameter `int` is generated as the default type for the length of vector `u32v_src`.

In `c-lambda` function pointer-casting operations are generated to extract the C-pointers for the vectors. On line 18, extracted C-pointer type `___U32*` is generated for kernel parameter `u32v_src` . Our implementation can identify its type from its constant prefix `u32v`. Here, `u32` determines the C-pointer type `___U32*`. The name of C-pointer `host_u32v_src` is generated by adding prefix `host_` to its name `u32v_src` . Then, our implementation generates the `___CAST` macro to extract the C-pointer. Our implementation keeps track of the position of vectors in a parameter list because a Scheme vector in a pointer-casting operation must be referred to with the keyword `___arg` followed by its position in that parameter list. In this case, our implementation generates `___arg9` on line 18 of the `___CAST` macro to refer to type `scheme-object` in the parameter list.

Next, our implementation generates a call to the CUDA-C shim on lines 20–23. The kernel name `vector_addition` is appended with the suffix `_cu_driver` to generate the name of CUDA-C shim `vector_addition_cu_driver`. Then our implementation generates the arguments list. All the parameters of this `c-lambda` function are passed to the CUDA-C shim. Therefore, in the argument list parameters are referred to using the keywords `___arg` followed by their position in the parameter list apart from the vectors. For a vector the casted pointer is passed to the CUDA-C shim. Our implementation generates the casted C-pointer variable name, and passes this pointer variable as an argument to the CUDA-C shim. On line 18, `host_u32v_src` is a C-pointer type variable for the vector `u32_src`. In order to generate it in the argument list, our implementation keeps track of the position of a vector and prefix `host_` is added to its name on line 22.

Our implementation also generates the `c-declare` constructs to provide forward declarations for CUDA-C shims. Lines 3–8 on Listing 4.2 is the forward declaration for the CUDA-C shim `vector_addition_cu_driver`. The return type `void` is generated on line 15 because the CUDA-C shim does not return anything.

Our implementation generates the Scheme shim in a file named *`file-name`*`_gpu-interface_`*`kernel-name`*`.scm`. Here, *`file-name`* specifies the name of a `.scm` file that has the kernel skeleton. In this case, for the kernel skeleton `vector_addition` in Listing 4.5, Scheme shim is generated in file `main_gpu-interface_vector_addition.scm` in Listing 4.2.

Note that our implementation appends the generated Scheme shim to the parse tree constructed from the file containing the kernel skeleton, but it does not change the source file containing the kernel skeleton. In order to do that, our implementation constructs a parse tree node for Scheme, including the expression

(include "*file-name*_gpu-interface_*kernel-name*.scm")

that appends the Scheme shim generated in file *file-name*_gpu-interface_*kernel-name*.scm to the parse tree generated from file *file-name*.scm .

The Gambit compiler compiles this Scheme shim to file `main.c`. Note that Gambit also compiles Scheme source expressions for host programs in the `main.scm` file that also contains the kernel skeleton to file `main.c` . Our implementation does not compile the kernel skeleton, It just constructs the parse tree of it. Therefore, only lines 5–15, representing the host program in Listing 4.1, are compiled to file `main.c`. In general, the file containing Scheme shim is deleted by `gsc`. In order keep the Scheme shim we added an extra flag `-keep-scm-shim` to `gsc` .

Our implementation generates a separate Scheme shim file for each kernel. After generating the Scheme shim it is compiled to the same file containing the compiled host program. If we want to generate all the Scheme shims in a single file then the file would be overwritten again and again by the generated Scheme shim for each kernel skeleton. Therefore, we generate separate files containing Scheme shims for each kernel skeleton.

Note that the commenting and indentation in Listing 4.5 is not auto-generated. We annotate the generated Scheme shim code from our implementation to provide more clarity to the readers.

4.5 Generating CUDA-C shim

In order to generate the CUDA-C shim, our implementation requires the kernel name and the actual kernel arguments. Kernel parameters contain type information, so it is easy to identify C-types for kernel arguments and generate them. For example, kernel parameter `u32v_src` has the constant prefix `u32v` that identifies its C-type as `uint32_t*` . Here, `u32` identifies the C-type `uint32_t` and `v` specifies that it is a C-pointer.

Our implementation generates CUDA-C shim `vector_addition_cu_driver` for the kernel skeleton `vector_addition` on lines 7–32 of Listing 4.6. First, our implementation generates a forward declaration for the kernel `vector_ addition` in line 2. Name of the kernel is extracted from parse tree nodes. Then, the parameter list is generated. In parameter list, C-type for kernel parameter `u32_constant` is `uint_32`, and C-type for `u32v_src` is `uint32_t*`. The arguments that take the length of a vector are also generated. In this example, `u32_src_len` is generated by adding the suffix `_len` with the actual name `u32_src`. Its default type, `int` is also generated.

On line 7, our implementation generates CUDA-C shim enclosed with `extern "C"` because it is linked from a `.c` file containing the compiled host program in C. It generates the name `vector_addition_cu_driver`

by adding suffix `_cu_driver` to kernel name `vector_addition`. Then it generates the parameter list. Seven parameters are generated on lines 8–9 to take the execution configuration values and the size of the dynamic shared memory. Then the actual kernel parameters are generated and for each vector an extra parameter is also generated to take the length of that vector. For a vector `h_` is added at the beginning of its name

```
1  //----------------------------kernel forward declaration ----------------------
2  __global__  void vector_addition (uint32_t u32_constant, uint32_t* u32v_src,
3                                    int u32v_src_len);
4
5  //----------------------------Kernel forward declaration-------------------------
6  //----------------------------CUDA-C shim Start----------------------------------
7  extern "C" {
8   void vector_addition_cu_driver (int gDx, int gDy, int gDz, int bDx,
9                                   int bDy, int bDz, int shared_size,
10                                  uint32_t u32_constant, uint32_t* h_u32v_src,
11                                  int h_u32v_src_len ){
12    //device pointers
13    uint32_t* d_u32v_src;
14    //calculating the size of device memory
15    size_t size_u32v_src = h_u32v_src_len * sizeof(uint32_t);
16    //allocating device memory
17    cudaMalloc((void **) &d_u32v_src, size_u32v_src);
18    //copying host to device
19    cudaMemcpy(d_u32v_src, h_u32v_src, size_u32v_src, cudaMemcpyHostToDevice);
20    //defining Grid configuration
21    dim3 dimGrid(gDx, gDy, gDz);
22    //defining Block configuration
23    dim3 dimBlock(bDx, bDy, bDz);
24    size_t size = shared_size;
25    //Now, at this point it calls the kernel
26    vector_addition <<< dimGrid, dimBlock, size >>> (u32_constant, d_u32v_src,
27                                                     h_32v_src_len);
28    //copying device to host
29    cudaMemcpy(h_u32v_src, d_u32v_src, size_u32v_src, cudaMemcpyDeviceToHost);
30    //deallocation of device memory
31    cudaFree(d_u32v_src);
32   }
33 }
34 //-------------------------------------CUDA-C shim End-----------------------------
```

Listing 4.6: Generating CUDA-C shim

to distinguish from its device memory pointer on line 10. The name of the parameter taking the length of a vector is also generated by adding `h_` at the beginning of a vector's name and `_len` at the end. Vector `u32v_src` is named `h_u32_src_len` on line 11. For the scalar type kernel parameter `u32_constant` on line 11, it remains unchanged in the parameter list of CUDA-C shim. This is because a scalar type CUDA-C does not need another scalar type in device memory in order to pass it to a CUDA-C kernel.

Memory operations on lines 17,19, 29, and 31 are generated by the vector's name, with prefixes added in order to distinguish them from device, host, and size variables. Since `u32v_src` has neither `OUT` nor `IN` in its name, line 19 transfers this vector from host to device memory before the kernel call (lines 26–27) and line 20 transfers it from device to host memory after the kernel call. Note that our implementation will not generate line 19 if `u32v_src` has an `OUT` notation. Similarly, line 29 will not be generated if an `IN` notation is present in `u32v_src`. Our implementation generates the device pointer by adding `d_` at the beginning of a vector name, in line 13. Size calculation in bytes is also generated on line 15 by adding `size` to the name of the vector `u32v_src`.

The generation of grid and thread block configuration variables on lines 21 and 23 are fixed string. A kernel call with execution configuration is generated on lines 26 and 27. The kernel's name is extracted from the parse tree. The execution configuration is also a fixed string

```
<<< dimGrid, dimBlock, size >>>
```

and the arguments are generated from the extracted parameters, but for a vector the device memory pointer is passed to a kernel. Our implementation can distinguish a vector from a scalar by identifying the sub-string `v` in the name of a kernel parameter. In the argument list, the first extracted kernel parameter `u32_constant` is added to the beginning of the argument list without any modification of its name because it is a scalar type. However, the second extracted kernel parameter `u32v_src` is identified as a vector, therefore its corresponding device memory pointer `d_u32v_src` is actually passed to the kernel. The suffix `d_` is added to the vector name and instead of vector `u32v_src`, device memory pointer `d_u32v_src` is added to the argument list. For a vector, its length is also passed to a kernel. Therefore `h_u32_src_len` - representing the length of `u32v_src` - is added to the argument list.

This CUDA-C shim in Listing 4.6 is generated in a file named `maingpu.cu` because the kernel skeleton `vector_addition` is in file `main.scm`.

4.6 Generating the Scheme kernel call

We provide a special construct in Scheme for Gambit to call a kernel. This special construct also has a part that defines an execution configuration. In our implementation, when a kernel is called by this special construct, it actually calls the generated Scheme shim. Our implementation identifies the special construct calling a kernel and then converts it into an ordinary Scheme function call.

4.6.1 Identifying a kernel call constructs

In order to identify a kernel call construct that has an execution configuration, our implementation checks for it in a parse tree node constructed with tag `app-tag` that has eight fields. `app-tag` specifies a node for a function call Scheme expression. A kernel call construct is actually a function call, except it has a part that contains expressions defining an execution configuration. In order to check for an execution configuration in a function call, parse tree nodes constructed from that function call must have the following parts:

1. It must be a function call.

2. Its first argument must be a marker `<<<` that marks the start of an execution configuration. Second and third arguments must be function calls containing expressions defining grid and thread block dimensions. Each of them must not contain more than three parts.

3. The fourth argument must be an expression to define the size of the dynamic shared memory. In this case, the fifth argument is the marker `>>>` that specifies the end of an execution configuration.

4. The last argument must be the marker `>>>` that marks the end of an execution configuration.

Our implementation checks for `app-tag` in parse tree nodes to identify a function call. In order to enable this we must supply a `-cu` flag to `gsc` . In our implementation, apart from the name of a kernel in our kernel call constructs, every thing is treated like normal arguments by Gambit.

In Figure 4.2 we show how the execution configuration expressions are treated as arguments in a node specified with `app-tag` as represented by blue-colored box with eight fields.

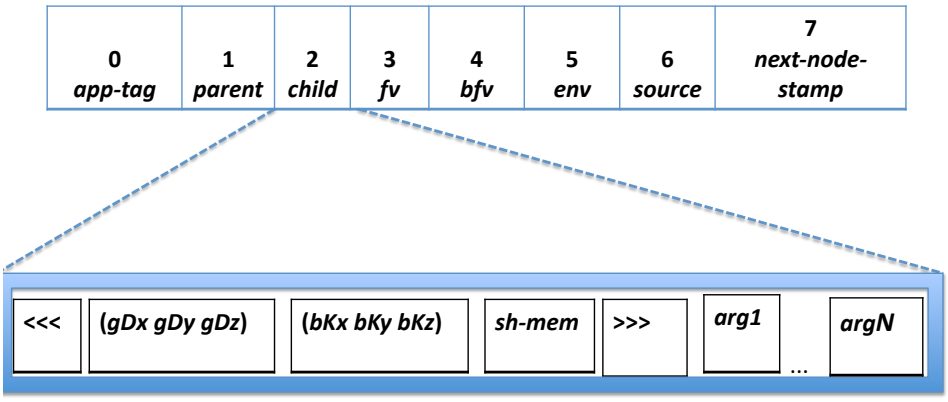


Figure 4.2: Accessing expressions defining an execution configuration in a parse tree

All of the nodes, represented as black-bordered rectangles containing argument symbols , are listed together and represented as a blue-bordered frame contained in field 2 (child) of the app-tag node. The first node contains the marker <<<. The second node contains arguments (*gDx gDy gDz*) that define grid dimensions. The third node contains arguments (*bKx bKy bKz*) that define thread block dimensions. The fourth node — *sh-mem* — contains expressions for defining the size of dynamic shared memory. The fifth node contains the marker >>>, which is followed by the nodes containing actual kernel arguments *arg1, ... argN*. This way our implementation can access and check expressions defining execution configuration and check whether it is a kernel call or not.

Note that the second and third nodes defining grid and thread block dimensions are also constructed as two app-tag nodes. This is because both (*gDx gDy gDz*) and (*bKx bKy bKz*) are treated as function calls by Gambit.

4.6.2 Converting an execution configuration into an argument list

Our implementation checks the parse tree of a function call in order to identify a kernel call. If it finds a kernel call with an execution configuration, it converts the expressions used in execution configuration into an argument list of a Scheme function call. Our kernel call construct is

```
(kernel-name <<< (gDx gDy gDz) (bKx bKy bKz) sh-mem >>> arg ...  )
```

After the conversion into an ordinary function call by our implementation, it becomes

```
(kernel-name gDx gDy gDz bKx bKy bKz sh-mem arg ...  )
```

Our implementation arranges all seven execution configuration expressions into an ordinary argument list.

An execution configuration might not have expressions defining y- or z-dimensions of a grid and thread blocks, or defining the size of dynamic shared memory. In that case, our implementation add 1s in the argument list for the positions reserved for y- or z-dimensions and adds 0 for dynamic shared memory. For example, on lines 12–14 of Listing 4.1, kernel call

```
(vector_addition <<< (nblocks) (blockSize) (* (* 2 N) size-int32) >>>
                     constant
                     src)
```

does not define y- and z-dimension for grid and thread block organization. Our implementation converts it into an ordinary function call and inserts 1s in the argument list in places reserved for y- and z-dimensions:

```
(vector_addition nblocks 1 1 blockSize 1 1 (* (* 2 N) size-int32) constant src)
```

This is shown in Listing 3.1 in Chapter 3. Note that this function call actually calls the vector-length-

calculation helper function `vector_addition` on lines 29–37 of Listing 4.5.

Our implementation extracts the parse tree nodes of execution configuration expressions and actual kernel arguments expressions from the `app-tag` node. Next, it removes the parse tree nodes for the two markers `<<<` and `>>>` , as well as the two `app-tag` nodes for expressions (*gDx gDy gDz*) and (*bKx bKy bKz*) defining grid and thread block organizations. The six `var-tag` nodes for expressions - *gDx* , *gDy* , *gDz* , *bKx* , *bKy* , and *bKz* - are then extracted from their parent `app-tag` nodes constructed from expressions (*gDx gDy gDz*) and (*bKx bKy bKz*). These six `var-tag` nodes are inserted into the `app-tag` node constructed from a kernel call construct in the sequence *gDx*, *gDy*, *gDz*, *bKx*, *bKy*, and *bKz* in order to make them arguments of an ordinary function call.

In order to arrange them in a sequence of arguments, *next-node-stamp* field of nodes for six execution configuration expressions, shared memory, and actual kernel arguments must be sequenced. For example, in order to make the expression *gDx* as the first argument, *next-node-stamp* field of parse tree node sets to `0`. In the same way, *next-node-stamp* field of parse tree node for expressions *gDy* sets to `1`. In this way, *next-node-stamp* field of expression *sh-mem* is set to `6`. For the first kernel argument the *next-node-stamp* fields is set to `7`. Similarly, the rest of the kernel arguments follow the same sequence to set their *next-node-stamp* fields.

In an execution configuration, programmers may define only one or two expressions with x- or y-dimensions. In that case, our implementation constructs a parse tree node that contains a constant `1` for those missing dimensions. It then adds the parse tree node as an argument to the `app-tag` node that calls the kernel. Note that *next-node-stamp* field of that node must be sequenced.

Programmers might not define an expression for dynamic shared memory. In this case, our implementation generates a parse tree node that contains a constant `0` and inserts it as an argument to the `app-tag` node constructed from a kernel call special form. Its *next-node-stamp* field is set to `6` because it is the seventh argument in the argument list.

So far in this chapter we described the generation of both shims from a kernel skeleton in Scheme. We also described how a kernel call construct is converted to an ordinary function call that calls the generated Scheme shim to link a CUDA-C kernel. In the following section we describe the implementation of synchronization special forms in Gambit.

4.7 Implementation of synchronization macro

We provide synchronization macro `sync` in Gambit to synchronize the execution of multiple kernels. `sync` can take multiple kernel calls as its arguments, and after each kernel call this macro calls a generated `c-lambda` function `call-syncthread` shown in Listing 4.7.

From this `c-lambda` function CUDA library function `cudaDeviceSynchronize()` is called, as shown in line 5. This call actually forces the host program running on a CPU to be idle until all the previously-launched

kernels running on GPUs have completed their executions. Note that this macro can also take

```
(define call-syncthread
  (c-lambda ()
            void
#<<c-lambda-end
    cudaDeviceSynchronize();
c-lambda-end
))
```

Listing 4.7: Definition of `c-lambda` function `call-syncthread`

only a single kernel call, where `call-syncthread` will be called only once after that kernel call. `async` specifies non-synchronized kernel calls in a host program and can take multiple kernel calls as its arguments. It organizes the kernel calls sequentially but it does not make a call to `cudaDeviceSynchronize()` library function in between two kernel calls. This macro can also take one kernel call as argument. Note that the execution of multiple kernels is not synchronized without a call to `cudaDeviceSynchronize()` in between two kernel calls.

In order to manage GPUs CUDA provides some library functions. Therefore, we also provide some useful library functions for Gambit Scheme that enables programmers to manage GPUs from Scheme as described in the following section.

4.8 Library functions in Scheme for GPU

In CUDA, there are library functions for querying and managing devices, controlling kernel executions, timing kernel executions, and managing versions of CUDA runtime libraries. These library functions are faceted. Programmers often need to provide multiple library functions to perform a task. Therefore, we provide some useful library functions in Scheme for GPUs for our implementation. These simple library functions reduce the burden of combining multiple library functions. It also gives opportunity for Scheme programmers to use those CUDA-C functionalities in Scheme.

4.8.1 Device management

Sometimes programmers may need some functionality for managing GPU devices. The following functionalities help programmers to query a device and find out an appropriate device for data-parallel computation.

- `(query-device)`

  The `(query-device)` library function displays to the standard output the properties of all the devices. It prints the compute capability, shared memory size per block, total number of registers per block, wrap size, maximum threads per block, maximum grid dimension, clock rate, total constant memory,

and total constant memory texture alignment of all active devices. `query-device` uses CUDA library function

`cudaGetDeviceProperties(struct cudaDeviceProp *`*`prop`*`, int `*`device-id`*`)`

to get the device properties of a particular device. This utility function sets the CUDA structure `cudaDeviceProp` [12] for a valid *`device-id`*. Here, *`device-id`* is an integer index number of an active GPU device. It also uses `cudaGetDeviceCount(int *`*`count`*`)` [12] to get the number of currently active devices.

- `(check-and-set-device `*`device-id`*`)`

  The `(check-and-set-device `*`device-id`*`)` library function checks for a valid *`device-id`* . This means it checks for an actual GPU device not the CUDA device CPU emulation (a CPU emulator that can run CUDA-C kernels on a system with no installed GPU). If the *`device-id`* is less than or equal to the system device count then it is a valid device. Next, it checks for CPU device emulation. If `deviceProp.major` defined in the structure `cudeDeviceProp` is greater than `999` then it is not a GPU device. If the device passes these two tests, it sets the device for the current execution and responses to the standard output device. Here, *`device-id`* is an integer number.

- `(set-device `*`device-id`*`)`

  The `(set-device `*`device-id`*`)` library function does not perform any checking operations, but rather sets the device using the CUDA library function

  `cudaSetDevice(int `*`device-id`*`)`

  If the device does not exist, it raises a `cudaError_t` type exception `cudaErrorInvalidDevice` .

- `(get-device-id)`

  The `(get-device-id)` library function returns an integer index number of an active device in *`device-id`* using the library function [12]

  `cudaGetDevice(int *`*`device-id`*`)`

- `(reset-device `*`device-id`*`)`

  The `(reset-device `*`device-id`*`)` library functions resets all states of an active device. It also cleans up any resources allocations by the current program on the device. This function resets the device immediately. The programmer must ensure that the device is not accessed by any other running host threads when this function is called. Note that this function does not check for other running host

threads that might try to access the same device. We recommend that Scheme programmers use `reset-device` at the beginning or at the end of a host program.

- `(reset-all-devices)`

  The `(reset-all-devices)` library function resets all the devices and destroys all previous resource allocations. We recommend that Scheme programmers also use `(reset-all-devices)` at the beginning or at the end of a host program.

4.8.2 Version management

- `(driver-version)`

  We provide library function `(driver-version)` to know the version number of the installed CUDA driver application programming interface. This library function prints the version number to standard output. It prints `0.0` if no driver is installed. This version number describes the features supported by the installed driver and runtime libraries. Programmers can check whether they need to install a newer device driver than the one currently installed.

- `(runtime-version)`

  The `(runtime-version)` library functions prints the version of the installed CUDA runtime libraries to standard output. Note that CUDA runtime libraries must be compatible with the version number of installed CUDA driver.

4.8.3 Timing function

We implemented a simple timing function `(gpu-time` *kernel call ...*`)` in Scheme to measure execution time for kernels. This timing function reduces the burden of defining, synchronizing, and measuring time lapse between separate CUDA events. It takes into account execution times for kernels, Scheme shim, and CUDA-C shim. This means it also counts vector transfer times between device and host memory, allocation and deallocation operations of device memory pointers in CUDA-C shim.

`gpu-time` displays the kernel execution time in milliseconds to standard output, and it also returns time for later use to a host program. It is similar to Scheme `time` macro. For example:

```
(gpu-time
  (k1 <<< >>> arg ...)
  (k2 <<< >>> arg ...))
```

measures the combined execution time for kernel *k1* and *k2* .

In CUDA-C, programmers need to create two `cudaEvent_t` [12] type variables to capture the kernel start and end time-stamps. `cudaEventCreate`[12] then creates both the events one before the kernel call

and another after the kernel call. `cudaEventSynchronize` [12] then waits to finish the kernel execution and `cudaEventRecord` [12] computes the elapsed time between the two events in milliseconds. In our implementation, the `gpu-time` function only requires *`kernel call`* expressions and is very simple to use.

In order to implement this library function, our implementation generates two files: `cudalib.scm` and `cudalib.cu`. Here, Scheme file `cudalib.scm` contains `c-lambda` and `c-declare` constructs in order to call C-functions in file `cudalib.cu`. For example, library function `(query-device)` calls a `c-lambda` function `query-device` in file `cudalib.scm`, which then `device-query` calls the C-function `deviceQuery` in file `cudalib.cu` with the CUDA-C code for querying a device.

4.8.4 CUDA library functions unavailable in Scheme

We have implemented some useful library functions in Scheme that enable convenient GPU management for Scheme programmers. However, we do not provide all the CUDA library functions in Scheme. The reason is that many of these library functions are not frequently used in CUDA programs. Moreover, a Scheme implementation of these library functions would not be convenient for Scheme programs because many of them depend on many C `struct` values which would need to be constructed in Scheme. In order to use them in Scheme programs, programmers still need to learn and know low-level CUDA detail; and, this is not necessarily useful to Scheme or other functional languages.

Some CUDA library functions unavailable in Scheme are listed below:

- In CUDA, programmers can choose devices using the library function

  `cudaChooseDevice ( int * device, const struct cudaDeviceProp * prop )`

  based on the values set to fifty-one fields of the `cudaDeviceProp` structure type variable *`prop`*. We do not provide this library function in Scheme because programmers need to set all the fifty-one fields of the `cudaDeviceProp` structure type variable *`prop`* which might not be convenient for the Scheme programmers.

- CUDA provides library function

  `cudaPointerGetAttributes (struct cudaPointerAttributes *attributes, const void *ptr)`

  to return the attributes of a pointer *`ptr`* to a `cudaPointerAttributes` structure type variable *`attributes`* . This function can identify whether a pointer points to device or host memory. In our implementation we do not need to provide a separate device memory vector in Scheme. Our implementation hides a device memory pointer for a Scheme vector generated in CUDA-C shim to provide an abstraction to reduce hands-on memory management. In our implementation we assume

that a Scheme vector defined in a host program is in host memory, but when it is passed to a kernel skeleton it is in device memory. Therefore, we do not need to provide this library function in Scheme.

- CUDA provides library function

```
cudaMallocHost(void ** ptr, size_t size)
```

that allocates *size* bytes of page-locked host memory to a pointer *ptr* which is accessible from a device. Allocation of an excessive amount of page-locked memory may degrade system performance due to a lack of available memory for paging. Accessing host memory from a kernel is an order of magnitude slower than accessing device memory. It also has both very high latency and very limited throughput. It is also recommended this library function to be used more economically in terms of host memory consumption. Apart from that, this library function is usually used to allocate host memory to manage streams of operations on the GPU. This is not covered in this thesis, therefore we do not provide this library function in Scheme.

- A device may have the capability to directly access the device memory of a peer device. CUDA provides library function

```
cudaDeviceCanAccessPeer (int * canAccessPeer, int device, int peerDevice)
```

to verify whether device *device* can access the device memory of another device *peerDevice*. This library function returns 1 in *canAccessPeer* if *device* has access to the device memory of device *peerDevice* . If not, this library function returns 0 in *canAccessPeer*. A device can enable the accessibility of another device to its memory by calling a library function `cudaDeviceEnablePeer Access()`. Note that access to peer device is not available to 32-bit CUDA applications. We do not provide this library function in Scheme because in this thesis we only focus on the interactions between host and device. Moreover, peer access is not a common practice in most of the CUDA applications [1].

- CUDA provides some stack-based library functions in order to launch a kernel. First, library function

```
cudaConfigureCall( dim3 gridDim , dim3 blockDim , size_t sharedMem = 0,
                   cudaStream_t stream = 0)
```

pushes execution configuration parameters to the top of stack. The arguments for a kernel can be pushed to the top of stack using the library function

[1] Only the *Simple Peer-to-Peer Transfers with Multi-GPU* CUDA SDK example uses this library function. It just demonstrates how to access device memory of a peer device.

```
cudaSetupArgument( const void *  arg , size_t  size )
```

Here, argument *arg* of `size` bytes is pushed to the stack. Note that `cudaSetupArgument()` must be preceded by the library function `cudaConfigureCall()` . Kernel name *entry* is then launched using the library function

```
cudaLaunch( const char *  entry )
```

which is preceded by calls to `cudaSetupArgument()` . Note that all three library functions are a part of C API. That means they can be compiled by a C-compiler. These library functions are used as an alternative of CUDA execution configuration syntax to launch a kernel. CUDA execution configuration syntax can only be compiled through `nvcc` compiler. In our implementation, we provide a special form in Scheme to call a kernel, therefore, we do not need to provide these stack-based library functions in Scheme to launch a kernel.

4.9 Template `makefile`

We provide a template `makefile` in Listing 4.8 that can manage the generation of Scheme shim and CUDA-C shim from a Scheme file, and compiles to a GPU executable from the provided source files. First, `gsc` generates the Scheme shim and CUDA-C shim from a Scheme file and compiles the Scheme file to a C file. `gsc` then generates a link file that contains linking information gathered from the Scheme file. Next, `gsc` generates object files from the compiled C file, linking file, CUDA-C shim file, and file containing the actual definition of the CUDA-C kernel. Finally, `nvcc` compiler generates a GPU executable from those object files.

In this `makefile`, programmers must provide the names of Scheme file containing the kernel skeleton , the CUDA-C file containing the kernel definition and discard the `.scm` and `.cu` extensions. On line 11, variable `SCM_FILE` must be assigned with the name of a Scheme file, and variable `KERNEL_FILE`, on line 14, must be assigned with the name of a CUDA-C file. For example, in Listing 4.1, file `main.scm` contains the kernel skeleton `vector_addition` on lines 2–3. The definition for kernel `vector_addition` is in file `kernel.cu`. Therefore, programmers should assign `main` to variable `SCM_FILE` and `kernel` to variable `KERNEL_FILE`. Programmers must provide the Gambit installation directory in variable `GAMBIT_DIR` on line 19, and the CUDA installation directory in variable `CUDA_DIR` on line 21.

First, `gsc` (assigned to variable `$(GSC)`) generates the Scheme shim and CUDA-C shim `$(SCM_FILE)gpu.cu` from Scheme file `$(SCM_FILE).scm` on line 25 and compiles `$(SCM_FILE).scm` to a C file `$(SCM_ FILE).c` on line 82. Command-line option `-cu`, in line 66, is provided in order to generate the Scheme shim and CUDA-C shim. For example, `gsc` compiles file `main.scm` to file `main.c`. At the same time, `main_gpu-interface_vect`

or_addition.scm — containing the Scheme shim — and maingpu.cu — containing the CUDA-C shim — are generated. Note that our implementation inserts the Scheme shim into the parse tree generated from the Scheme file main.scm containing the kernel skeleton , and compiles the Scheme shim along with the host program to a C file. Here, generated Scheme shim in file main_gpu-interface_vector_ addition.scm is appended with the parse tree generated from main.scm. gsc then compiles that combined parse tree to main.c. Note that a -keep-scm-shim option is provided to gsc in order to keep the Scheme shim file.

```
#
# template makefile for build CUDA-scheme executables.
# (c) 2013 Rashed Chowdhury ....
#
# to use this, fill in the SCM_FILE (without the .scm extension!)
# ...

#################################################################

# no .scm extension
SCM_FILE =

# no .cu extension -- often the same as $(SCM_FILE)
KERNEL_FILE =

#################################################################
# if your installation directories change, fix these macros
# main gambit install
GAMBIT_DIR =/grad/arc552/research/Gambit-Install-Dir
# main CUDA install
CUDA_DIR =/usr/local/cuda


#you shouldn't need to change these
GSC =  $(GAMBIT_DIR)/bin/gsc
NVCC = $(CUDA_DIR)/bin/nvcc
TEMP = cudalib.cu cudalib.scm
#################################################################


VERBOSE = -verbose
ARCH = -arch=sm_20
CUDA_LIB = -lcuda -lcudart
```

```
LIB = $(CUDA_LIB) -ldl -lgambc -lm -lutil
RM = -rm

#################################################################
all:            $(SCM_FILE).exe
#################################################################


#####################################################################
#DO NOT CHANGE ANYTHING BELOW THIS LINE

#step 4: building a GPU executable
$(SCM_FILE).exe: $(SCM_FILE).o $(KERNEL_FILE).o $(SCM_FILE)gpu.o  $(SCM_FILE)_.o
        -@echo "---------------------------------------"
        -@echo "generating GPU executable:- "$(SCM_FILE)".exe"
        -@echo "---------------------------------------"
        $(NVCC) -v -L  "$(GAMBIT_DIR)/lib" -L "$(CUDA_DIR)/lib64" -D___SINGLE_HOST\
                 $(LIB) -o $(SCM_FILE).exe  *.o
        -@echo "--------removing object-files and temporary library files-------"
        $(RM) -f *.o   $(TEMP)
        $(RM) -f $(SCM_FILE)gpu.cu
        $(RM) -f $(SCM_FILE)_.c
        $(RM) -f $(SCM_FILE).c


#step 3: generating objects file
# also generates $(SCM_FILE)gpu.o
$(SCM_FILE).o $(SCM_FILE)gpu.o: $(SCM_FILE).c $(KERNEL_FILE).cu $(SCM_FILE)gpu.cu \
                                $(SCM_FILE)_.c
        -@echo "--------------generating object files-------------"
        $(GSC)  -cc-options #-------------------------------------------------#
                       "-I $(CUDA_DIR)/include -I $(GAMBIT_DIR)/include  $(ARCH)" \
                -ld-options-prelude #-----------------------------------------#
                       "-L $(CUDA_DIR)/lib64 $(CUDA_LIB) -L  $(GAMBIT_DIR)/lib"   \
                -obj $(VERBOSE) $(SCM_FILE)gpu.cu $(KERNEL_FILE).cu $(SCM_FILE).c \
                                $(SCM_FILE)_.c


#step 2:- build link file
```

```
73 $(SCM_FILE)_.c: $(SCM_FILE).c
74         -@echo "------------generating link file-----------------------"
75         $(GSC) -link $(VERBOSE) $(SCM_FILE).c
76
77
78 #step 1:- build the shim
79 # also generates CUDA-C shim file $(SCM_FILE)gpu.cu
80 $(SCM_FILE).c $(SCM_FILE)gpu.cu: $(SCM_FILE).scm
81         -@echo "-------Building Scheme shim and CUDA shim--------------"
82         $(GSC)  $(VERBOSE)  -keep-scm-shim  -bare-time -cu $(SCM_FILE).scm
83
84 #utility targets
85 clean:
86         $(RM) -f $(SCM_FILE).exe *.o $(SCM_FILE)_.c  $(SCM_FILE).c                \
87                  *_gpu-interface_*.scm
```

Listing 4.8: Template `makefile` for compiling kernel skeleton

Next, the link file is generated from the compiled C file `$(SCM_FILE).c` on line 75. In case of `main.c`, `main_.c` is generated as the link file. Objects files are then generated by invoking `gsc` on line 64–69 from `$(SCM_FILE)gpu.cu` (CUDA-C shim), `$(KERNEL_FILE).cu` (definition of CUDA-C kernel), `$(SCM_FILE).c` (compiled C file that also contains Scheme shim), and `$(SCM_FILE)_.c` (linking information) files. Therefore, `maingpu.cu`, `kernel.cu`, `main.c`, and `main_.c` are compiled to object files by invoking `gsc`. Note that in this stage `gsc` compiles kernel file `$(KERNEL_FILE).cu` to the object file `$(KERNEL_FILE).o` by implicitly invoking `nvcc`.

Last, `nvcc` is directly invoked to generate GPU executable `$(SCM_FILE).exe` from the object files on lines 56–57. Therefore, GPU executable `main.exe` is generated from object files `maingpu.o`, `kernel.o`, `main.o`, and `main_.o`.

This template `makefile` deletes the CUDA-C shim `$(SCM_FILE)gpu.cu` on line 54. It also deletes the `cudalib.scm` and `cudalib.cu` files assigned to variable `$(TEMP)` on line 53. Link file `$(SCM_FILE)_.c` and C file `(SCM_FILE).c` are also deleted on lines 55 and 56. In order to keep them, programmers need to comment out these commands. In order to delete the Scheme shim programmers can remove the command-line option `-keep-scm-shim` on line 82. `gsc` then automatically deletes the Scheme shim during the time of compilation.

We also introduce to Gambit another command-line option on line 82: `-bare-time`. This displays two execution times in millisecond to a standard-output device in milliseconds. Here, the first execution time is for a CUDA-C kernel and the second one is the combined execution time for the CUDA-C shim of that kernel and the kernel itself. These two timings help to analyze performance for the different parts of the generated shims. Programmers can avoid these two timings by removing this command-line option from the template `makefile`.

4.10 Summary

In this chapter, we showed Scheme special forms for GPUs in order to generate the foreign-function interface. These special constructs carry special symbols that are necessary to generate the interface. Gambit constructs parse tree nodes from these constructs and extracts necessary information from the parse tree nodes. The parameters of those special constructs contain type information in their names. This type information can be used to identify types, and generate the appropriate types and generate the appropriate types in different parts of the interface.

We also showed how the special kernel call constructs can be converted to an ordinary Scheme function call to call the generated Scheme shim and supply necessary execution configuration parameters to define the execution configuration in CUDA-C shim. We then we described some useful library functions from Scheme to ease the GPU management.

We also described a template `makefile` that manages the generation of the foreign-function interface and builds a GPU executable from files containing kernel skeletons in Scheme and actual CUDA-C kernel definitions. The steps involved in the compilation process are demonstrated by this `makefile`.

We now need to evaluate our implementation in Gambit. In order to do that we measure the runtime performances of some example test cases implemented both in CUDA-C and Scheme. This will be described in the next chapter.

Chapter 5

Testing and Evaluation

We provided six test cases in order to evaluate the performance of our implementation in Gambit. We implemented our test cases both in Scheme and CUDA-C. We then analyzed performance of these test cases implemented in Scheme by comparing then with their implementation performance in CUDA-C. Some of the CUDA-C implementations are taken from various open-source projects. For each test case Scheme implementation a Scheme program was linked to CUDA-C kernels. Note that we also used the same CUDA-C kernel in the CUDA-C implementations for that test case.

Our testing platform consisted of a host machine connected to with two GPU devices: `GeForce GTX 560 Ti` [10] and `Quadro 600` [13]. Although the machine was equipped with an on-board `Quadro 600`; that processor is very weak and was dedicated to system display purposes. Therefore, we only tested GPU functions on the higher performance `GeForce GTX 560 Ti`. Table 5.1 we provide some specifications of `GeForce GTX 560 Ti`.

Table 5.1: Specifications of `GeForce GTX 560 Ti`

Specifications	`GeForce GTX 560 Ti`
Compute capability	2.1
Global memory	1GB
Shared memory per thread block	48KB
Constant memory	64KB
Maximum threads per thread block	1024
Maximum thread blocks per grid	65535
No. of multiprocessor	8
Bus type	GDDR5
Bus interface	PCIe 2.0 x16

In order to evaluate the runtime performance of the generated code by our implementation in Gambit, we needed to compare execution times for a test case implemented in Scheme with its CUDA-C implementation. Our host machine had a model 44 `Intel (R) Xeon (R) CPU E5607 @ 2.27 GHz` processor, was manufactured by Intel Corporation, and had 6 GB of memory. In our host machine, a `GNU/Linux` operating system with a `Linux` kernel was installed with version `#1 SMP Wed May 15 10:48:38 EDT 2013`. Note that our host machine did not drive a display and did not perform other tasks, so there was only a small overhead.

The version of Gambit compiler we used for our implementations was `4.6.2` [34]. The `nvcc` compiler driver

V0.2.1221, release 5.0 [7] was installed on our host machine, as well as CUDA driver 5.0 [7] and gcc version 4.4.7 [1]. For each experiment for our test cases, we ran each program 100 times and measured the arithmetic mean time and standard deviations for execution times. We also measured execution times against different vector sizes and different numbers of participating thread blocks. We then calculated curve-fitting functions and drew trend lines for both implementations of a test case to observe scaling behavior.

5.1 Measuring Execution Time

We evaluated our implementation in Gambit by comparing execution times for six test cases implemented in Scheme with CUDA-C. In this section we describe how we measured times both in Scheme and CUDA-C implementations for single and multiple kernel executions.

5.1.1 Single Kernel

- Measuring time in Scheme

  We used macro gpu-time to measure execution times for the Scheme implementations of our test cases. For example, in the following Scheme code gpu-time measures execution time for kernel kernel1 that takes a constant constant and src:

```
(gpu-time
     (kernel <<< (nblocks) (blockSize) >>> constant src))
```

  Here, gpu-time takes the kernel call kernel1 as an argument. In our implementation, gpu-time recorded the start time stamp for a kernel before calling both shims. This macro actually counted the execution times for the generated vector-length-calculation helper function and c-lambda function in Scheme shim, CUDA-C shim, and a supplied CUDA-C kernel. gpu-time also recorded the stop time stamp after the execution of CUDA-C shim.

  For a Scheme implementation, we also measured execution times for different parts of a generated foreign-function interface. In order to do that, we measured combined execution time for a generated CUDA-C shim and a supplied CUDA-C kernel. We also measured execution time for a supplied CUDA-C kernel only. In Scheme implementation for a test case, execution time is the combined execution time for a generated Scheme shim, CUDA-C shim, and a supplied CUDA-C kernel. Therefore, we can measure the execution time for a generated Scheme shim by comparing it with the combined execution time for a CUDA-C shim and a kernel. Execution time of a CUDA-C shim can be measured by comparing with the execution time for a kernel only.

  In order to measure execution times for a generated CUDA-C shim and a supplied CUDA-C kernel in Scheme implementation, we provided compiler flag -bare-time to generate CUDA code for measuring

times by Gambit compiler that involved two `cudaEvent_t` type variables. These two variables computed elapsed time to measure combined execution time for a generated CUDA-C shim and a supplied CUDA-C kernel and a s supplied CUDA-C kernel, as described in Appendix G. We also measured execution time for the generated code and subtracted it from execution times to measure actual execution times.

- Measuring time in CUDA-C

  In order to measure execution times for the CUDA-C implementations of our test cases, we used CUDA library functions. In Listing 5.1 we show a code snippet from an example host program in CUDA-C that calls the CUDA-C kernel `kernel1` and measures execution times for this example using CUDA library functions involving `cudaEvent_t` types. On line 1, four `cudaEvent_t` types variables (`start`, `stop`, `start_k` and `stop_k`) are declared. Here, `start` and `stop` time stamps are used to measure execution time for CUDA-C implementation. This execution time includes vector transfer operations between host and device memory and allocation/deallocation operations in device memory. The other two `cudaEvent_t` type variables (`start_k` and `stop_k`) are used to compute the elapsed time and to measure execution time for the kernel only.

  On line 2, a `float` type variable `elapsed_time_ms` is declared and initialized to store an elapsed time between two time stamps. On lines 3–6, `start`, `stop`, `start_k`, and `stop_k` are initialized using CUDA library functions `cudaEventCreate()`. Then the start time stamp for CUDA-C implementation is recorded on line 8:

  ```
  cudaEventRecord(start, 0)
  ```

  Next, device memory is allocated on line 10. Next, the vector is transferred to device from host memory on line 11 and the kernel is called on lines 15–16. Here, start time stamp only for the kernel execution is recorded just before the kernel call on line 13:

  ```
  cudaEventRecord(start_k, 0)
  ```

  The stop time stamp `stop_k` is recorded on line 16:

  ```
  cudaEventRecord(stop_k, 0)
  ```

  just after the kernel call. After the kernel execution, the vector is transferred back to host memory on line 19. Next, device memory is deallocated on line 20. The stop time stamp is recorded on line 22:

  ```
  cudaEventRecord(stop, 0)
  ```

```
1         cudaEvent_t start, stop, start_k, stop_k;
2         float elapsed_time_ms = 0.0f;
3         cudaEventCreate(&start);
4         cudaEventCreate(&stop);
5         cudaEventCreate(&start_k);
6         cudaEventCreate(&stop_k);
7         //taking stop time stamp for CUDA-C implementation
8         cudaEventRecord(start, 0);
9         size_t size_u32v_src = h_u32v_src_len * sizeof(uint32_t);
10        cudaMalloc((void **) &d_u32v_src, size_u32v_src);
11        cudaMemcpy(d_u32v_src,h_u32v_src,size_u32v_src,cudaMemcpyHostToDevice);
12        //taking start time stamp only for kernel execution
13        cudaEventRecord(start_k, 0);
14        //calling kernel
15        kernel1 <<<blockSize,nBlocks>>> (u32_constant, d_u32v_src,
16                                         h_u32v_src_len);
17        //taking stop time stamp only for kernel execution
18        cudaEventRecord(stop_k, 0);
19        cudaMemcpy(h_u32v_src,d_u32v_src,size_u32v_src,cudaMemcpyDeviceToHost);
20        cudaFree(d_u32v_src);
21        //taking stop time stamp for CUDA-C implementation
22        cudaEventRecord(stop, 0);
23        cudaEventSynchronize(stop);
24        cudaEventElapsedTime(&elapsed_time_ms, start, stop );
25        printf("%f,",elapsed_time_ms);
26        cudaEventElapsedTime(&elapsed_time_ms, start_k, stop_k );
27        printf("%f,",elapsed_time_ms);
28        cudaEventDestroy(start);
29        cudaEventDestroy(stop);
30        cudaEventDestroy(start_k);
31        cudaEventDestroy(stop_k);
```

Listing 5.1: Measuring execution times in CUDA-C implementation using `cudaEvent_t` type variables.

Next, the `stop` time stamp is synchronized on line 23 with the most recent call to `cudaEventRecord(stop,0)` on line 22. The elapsed time between `start` and `stop` time stamps is measured in milliseconds on line 24 and displayed to standard output on line 25. Execution time for the kernel only is measured by computing the elapsed time between time stamps `start_k` and `stop_k` on line 26 and displayed to standard output on line 27. Finally all four time stamps are destroyed on lines 28–31. Note that `start`

and `stop` time stamps also count the execution of time stamps `start_k` and `stop_k` on lines 18 and 22, but their execution times are so small that we can easily neglect them.

5.1.2 Multiple Synchronized Kernels

- Measuring time in Scheme

  Sometimes multiple kernel calls need to be synchronized because a kernel may depend on the results of other kernels. In order to synchronize multiple kernel executions we used our kernel synchronization macro, `sync`, shown on lines 4–6 of Listing 5.2. Below we describe how we measured time for the test cases involving multiple synchronized kernel executions using a sample Scheme code snippet. Here, the host program calls two synchronized kernels `kernel1` and `kernel2`.

```
1 (let ([constant 786]
2       [src (make-u64vector N 100)])
3   (gpu-time
4     (sync
5       (kernel1 <<< (nblocks) (blockSize) >>> constant src)
6       (kernel2 <<< (nblocks)(blockSize) >>> constant src))))
```

Listing 5.2: Measuring execution time for two synchronized kernels using `sync`.

  In this example, `gpu-time`, records the start time stamp before calling the first kernel (`kernel1`) on line 5. This calling actually calls the kernel's generated foreign-function interface which includes the vector-length-calculation helper function, `c-lambda` function, and CUDA-C shim. The second kernel (`kernel2`) is then called along with its generated foreign-function interface which includes the vector-length-calculation helper function, `c-lambda` function, and CUDA-C shim. Note that when the first kernel is finished, the vector returns back to host memory from device memory before the code involving kernel synchronization is called. Before calling to the second kernel, the vector is again transferred to device memory and the resultant vector is returned back to host memory after the second kernel is finished. Control is then returned back to the host program in Scheme. Finally, `gpu-time` records the stop time stamp. Note that in CUDA-C shim there is one allocation and one deallocation operation for each vector.

  Sometimes multiple kernels do not need to be synchronized. In that case, multiple kernel calls do not need `sync`, as on line 4.

- Measuring time in CUDA-C

  In Listing 5.3 we provide a sample CUDA-C code snippet that describes how we measured execution times for the CUDA-C implementation of our test cases involving multiple synchronized kernels. Here, `start` time stamp is recorded on line 6 and device memory is allocated on line 8. Next, the vector is

transferred to device memory on line 9 and the first kernel (`kernel1`) is called on lines 11–12. The vector is transferred back to host memory on line 13. On line 15, `cudaDeviceSynchronize()` holds all operations from lines 16–28 until (`kernel1`) is finished. This is the synchronization point. The vector is again transferred back to device memory before calling the second kernel (`kernel2`) on lines 18–19. When the second kernel is finished, the vectors is again transferred back to host memory from device memory on line 20.

```
1         cudaEvent_t start, stop;
2         float elapsed_time_ms = 0.0f;
3         cudaEventCreate(&start);
4         cudaEventCreate(&stop);
5         //taking start time stamp
6         cudaEventRecord(start, 0);
7         size_t size_u64v_src = h_u64v_srclen * sizeof(uint64_t);
8         cudaMalloc((void **) &d_u64v_src, size_u64v_src);
9         cudaMemcpy(d_u64v_src,h_u64v_src,size_u64v_src,cudaMemcpyHostToDevice);
10        //calling first kernel
11        kernel1 <<< blockSize, nBlocks>>> (u32_constant, d_u64v_src,
12                                           h_u64v_srclen);
13        cudaMemcpy(h_u64v_src,d_u64v_src,size_u64v_src,cudaMemcpyDeviceToHost);
14        //synchronizing kernel
15        cudaDeviceSynchronize();
16        cudaMemcpy(d_u64v_src,h_u64v_src,size_u64v_src,cudaMemcpyHostToDevice);
17        //calling second kernel
18        kernel2 <<<blockSize, nBlocks>>> (u32_constant, d_u64v_src,
19                                           h_u64v_srclen);
20        cudaMemcpy(h_u64v_src,d_u64v_src,size_u64v_src,cudaMemcpyDeviceToHost);
21        cudaFree(d_u64v_src);
22        //taking stop time stamp
23        cudaEventRecord(stop, 0);
24        cudaEventSynchronize(stop);
25        cudaEventElapsedTime(&elapsed_time_ms, start, stop);
26        printf("%f ", elapsed_time_ms);
27        cudaEventDestroy(start);
28        cudaEventDestroy(stop);
```

Listing 5.3: Measuring execution time for two synchronized kernels in CUDA-C implementation..

The device memory is deallocated on line 21 and the `stop` time stamp is recorded on line 23. In this CUDA-C implementation, the vector is allocated (line 8) and deallocated (line 21) only once. However, in the Scheme implementation in Listing 5.2, allocation and deallocation happen twice for the vector

`src`. The reason is that in this example, vector `src` is passed to both kernels. At first the vector is allocated and deallocated in the CUDA-C shim of the first kernel, `kernel1`, and then again in the CUDA-C shim of the second kernel, `kernel2`. Therefore, if a vector is passed to multiple kernels there will be separate allocation and deallocation operations in each CUDA-C shim for each kernel.

Note that a CUDA-C implementation does not need `cudaDeviceSynchronize()` as on line 15, if multiple kernels do not need to be synchronized.

So far in this chapter, we have described how time both in Scheme and CUDA-C for our test cases was measured. In the following sections we describe our test cases and analyze their performance in Scheme with CUDA-C. We provide charts to analyze performance. In those charts, we use different shapes and colors to represent different execution times:

- Blue diamonds represent execution time for a Scheme implementation that includes execution times for Scheme Shim, CUDA-C shim, and a supplied CUDA-C kernel. For multiple synchronized kernels, execution time for `sync` is also included.
- Red squares represent execution time for a CUDA-C implementation that includes execution times for vector transfer operations, allocation/deallocation operations, and a supplied CUDA-C kernel. For multiple synchronized kernel calls, it also includes execution times for `cudaDeviceSynchronize()`.
- Kiwi triangles represent combined execution time for a CUDA-C shim and a supplied kernel, in short cuShim+kernel.
- Blue circles represent execution time for a supplied kernel called from Scheme code.
- Red circles represent execution time for a supplied kernel called from CUDA-C code.

Apart from that we also provide trend lines in the charts to observe scaling behavior for different execution times. We use different colors to distinguish them:

- Blue trend lines show the scaling behavior for execution times for Scheme implementations as well as supplied CUDA-C kernels.
- Red trend lines show the scaling behavior for CUDA-C implementations as well as supplied CUDA-C kernels.
- Kiwi trend lines show the scaling behavior for cuShim+kernels.

We also provide data labels for different execution times. Blue and red boxes represent execution time in Scheme and CUDA-C, respectively. Kiwi boxes represents combined execution time for cuShim+kernels. In our charts, we also provide the mathematical functions for those trend lines. Blue and red square boxes represents functions for Scheme and CUDA-C, respectively. Kiwi boxes represent functions for cuShim+kernels.

5.2 Vector Addition

This is the simplest test case for our implementation. This example implemented element-by-element vector addition in the supplied CUDA-C kernel. In this test case, the host program passed two Scheme floating-point vectors and an integer constant to a kernel that performed parallel vector addition. Here, the first vector was an input vector and the second vector was an output vector. All participating GPU threads running the same kernel added the constant to an element of the input vector and stored the additions to the output vector. When the kernel is finished, the vector is returned back to the host program. We provided IN notation in the name of input vector and `OUT` notation in the name of output vector to avoid unnecessary memory transfer operations in the generated CUDA-C shim.

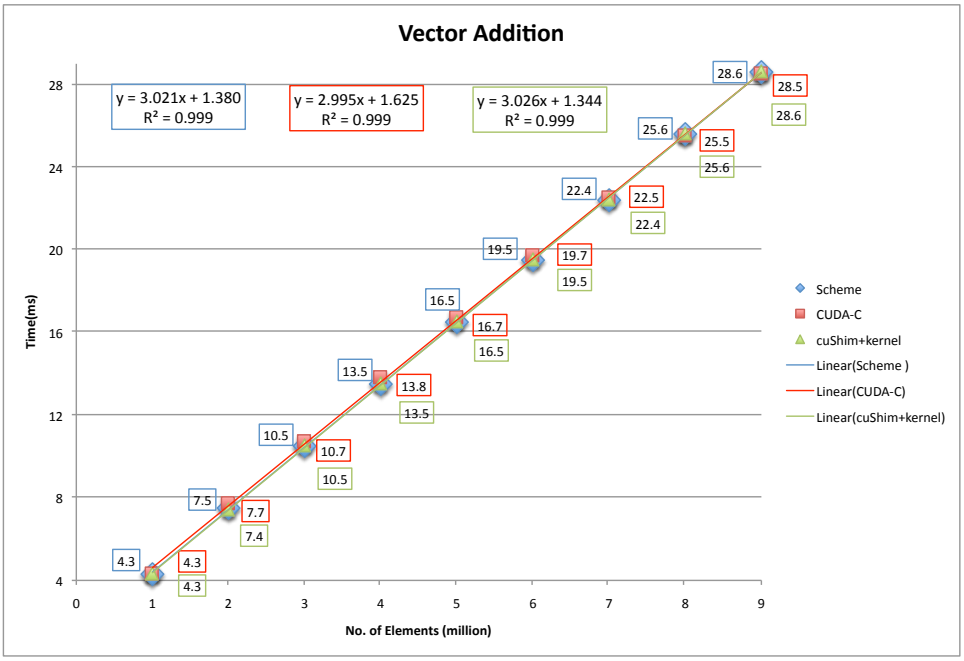


Figure 5.1: Performance comparisons of vector addition between Scheme and CUDA-C (block size fixed to 1024 threads)

The chart in Figure 5.1 compares execution times for this test case implemented in Scheme with CUDA-C. Here, Y-axis represents execution times in milliseconds and the X-axis represents vector sizes in millions. In execution configuration, we kept the thread block size to 1024 threads and the grid size changed according to the vector sizes on the X-axis. Both Scheme and CUDA-C showed liner trends $y = 3.021x + 1.380$ and $y = 2.995x + 1.625$, respectively. The execution time difference between Scheme and CUDA-C was $\Delta y = 0.026x - 0.245$, which is the execution time for the Scheme shim. Note that the R Square values were 0.999 for both trend lines, which means that the both lines fit the data perfectly. We observed no overhead in Scheme compared to CUDA-C. Execution times for cuShim+kernel also showed a liner trend $y = 3.026x + 1.344$. The R Square value for cuShim+kernel was 0.999, which means that the line fit the data

perfectly. This trend line was almost the same as the trend for Scheme. Therefore, most of the execution time in the Scheme implementation for this test case was occupied by the generated CUDA-C shim and the supplied CUDA-C kernel. Therefore, execution time for the generated Scheme shim was negligible.

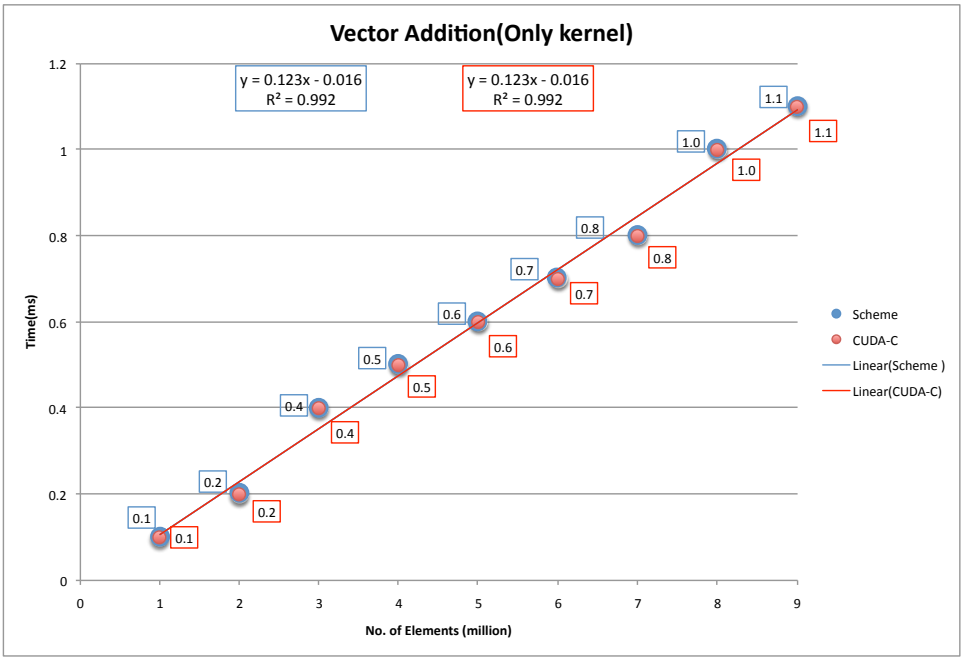


Figure 5.2: Performance comparisons for the supplied kernel only

We provide a chart in Figure 5.2 that compares execution time for the supplied CUDA-C kernel only for this test case called from both Scheme and CUDA-C code. We found that both Scheme and CUDA-C followed exactly the same liner trends of $y = 0.123x + 0.016$ and $y = 0.123x + 0.016$, respectively. There was no difference for the supplied CUDA-C kernel in this test case, weather it was called from Scheme or CUDA-C code. Note that the R Square values were 0.999 for both trend lines, which means that both lines fit the data perfectly.

In Figure 5.3 we compared execution times for this test case implemented both in Scheme and CUDA-C. We kept the grid size fixed to 65535 thread blocks as this was the maximum grid size in our testing GPU. Therefore, thread block size increased with increasing vector sizes along the X-axis. Here, we found that both Scheme and CUDA-C shim showed liner trends $y = 2.923x + 2.05$ and $y = 2.911x + 2.186$, respectively, and there was no overhead in Scheme compared to CUDA-C. The execution time difference between Scheme and CUDA-C was $\Delta y = 0.012xx - 0.136$; this is the execution time for the Scheme shim only. cuShim+kernel also followed liner trend $y = 2.931x + 1.975$; which is similar to the Scheme trend. We also found that most of the execution times in our Scheme implementation were occupied by the cuShim+kernel. Therefore, we found that the generated Scheme shim was negligible in this experiment. Note that the R Square values were 0.999 for all three cases, which means that all lines fit the data perfectly.

Figure 5.4 shows a comparisons of execution times for the supplied kernel in this experiment. We found

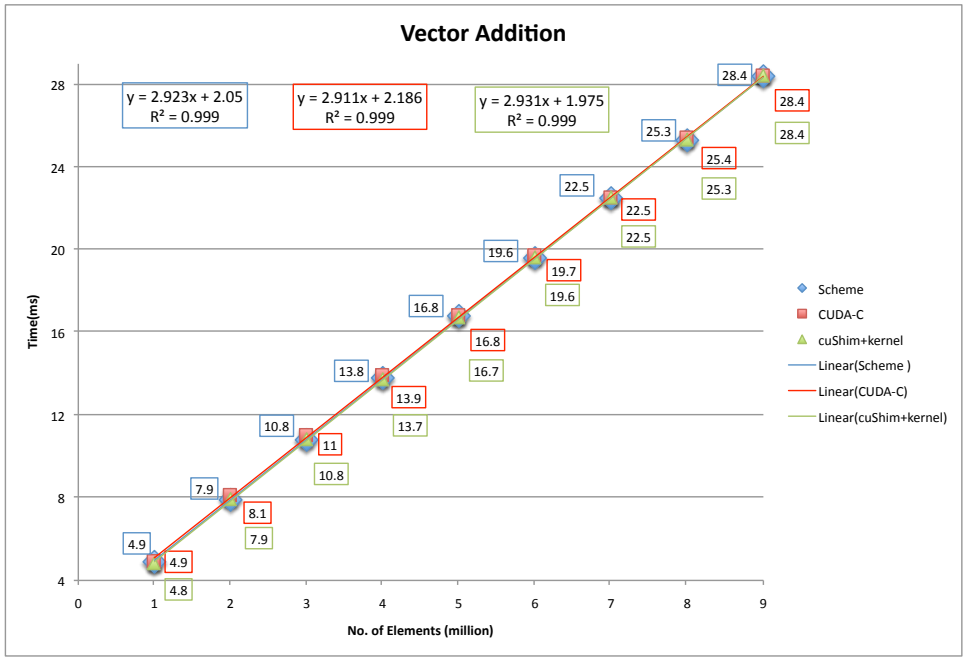


Figure 5.3: Performance comparisons of vector addition between Scheme and CUDA-C (grid size fixed to 65535 thread blocks)

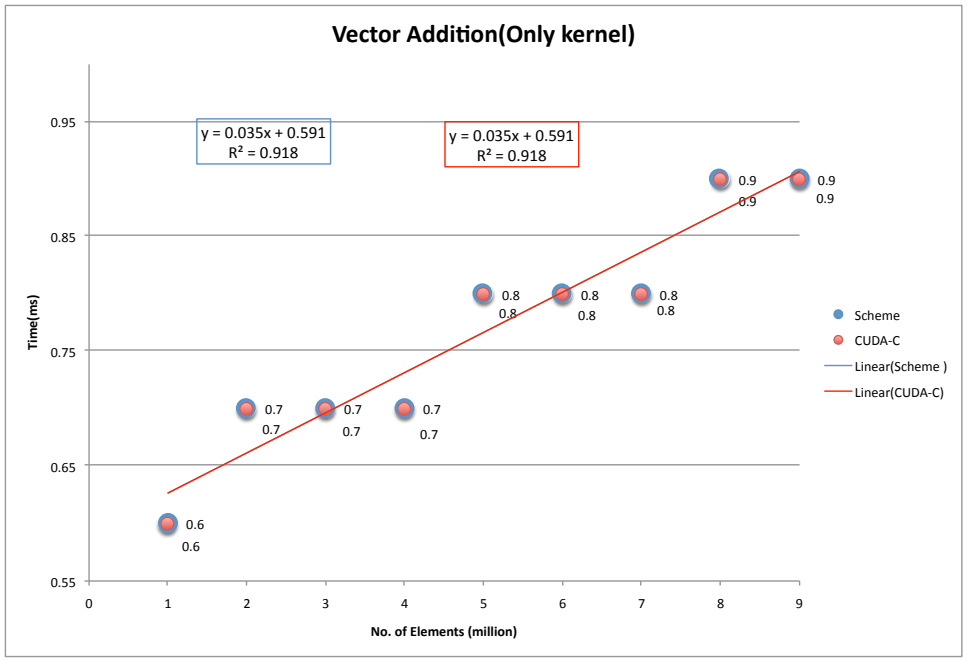


Figure 5.4: Performance comparisons for the supplied kernel only

that execution times for the CUDA-C kernel were the same in both cases. Both execution times for the kernel called from Scheme and CUDA-C followed liner trends $y = 0.035x + 0.591$ and $y = 0.035x + 0.591$, respectively. Execution times in both cases were the same for different vector sizes. Note that the R Square values were 0.918 for both Scheme and CUDA-C, which is a relatively good fit of the lines to the data sets.

In this test case we found that there was almost no overhead in Scheme compared to CUDA-C in both experiments. We observed that execution time for the Scheme shim was negligible in both experiments and most of the execution time for the Scheme implementations was occupied by the generated CUDA-C shim. We also observed that execution time for the supplied kernel was the same in both cases, weather it was called from Scheme or CUDA-C code. Note that the Scheme implementation of this test case is provided in Appendix A.

5.3 Multiple Synchronized Kernels

In this test case, a host program called two synchronized kernels, where the second kernel will not start until the first kernel has finished execution in device. The first kernel takes two vectors containing 32-bit unsigned-integers and an integer constant. The first vector is the input vector and contains initial data. The second vector is the output vector and contains the result. Each participating thread adds the supplied constant to an element of the input vector in parallel and stores the result to the output vector, just like the example described in Section 5.2. The output vector is then returned back to the host program. The second kernel would then be called called with two vectors also containing 32-bit integers and the same constant. Here, the first vector is also an input vector and the second vector is an output vector containing the result. In this test case, the output vector for the first kernel was the input vector of the second kernel. Therefore, the second kernel could not start execution until the first kernel was finished. In the second kernel, each participating thread subtracts the constant from the input vector and stores the result to the output vector. The resultant vector is then returned back to device memory, which means the input vector to the first kernel is now exactly the same as the resultant vector of the second kernel.

In this test case, we measured overhead for two synchronized kernel executions implemented in our system. Both kernels were synchronized using `sync`. The chart in Figure 5.5 compares execution times for this example implemented in Scheme and CUDA-C. Here, the X-axis represents vector sizes in millions and the Y-axis represents execution times in milliseconds. For results in Figure 5.5, we kept the block size fixed to 1024 threads, as this was the maximum block size in our testing GPU, and the grid sizes for this experiment were changed according to the vector sizes along the X-axis.

We found that both Scheme and CUDA-C showed liner trends $y = 4.231x + 1.675$ and $y = 5.398x + 2.741$, respectively. The time difference between Scheme and CUDA-C was $\Delta y = -1.167x - 1.066$; this included `sync` macro, two generated Scheme shims, and four allocation/deallocation operations. It suggests that there was no overhead in Scheme for this test case compared to CUDA-C. Note that the R Square value was 0.990

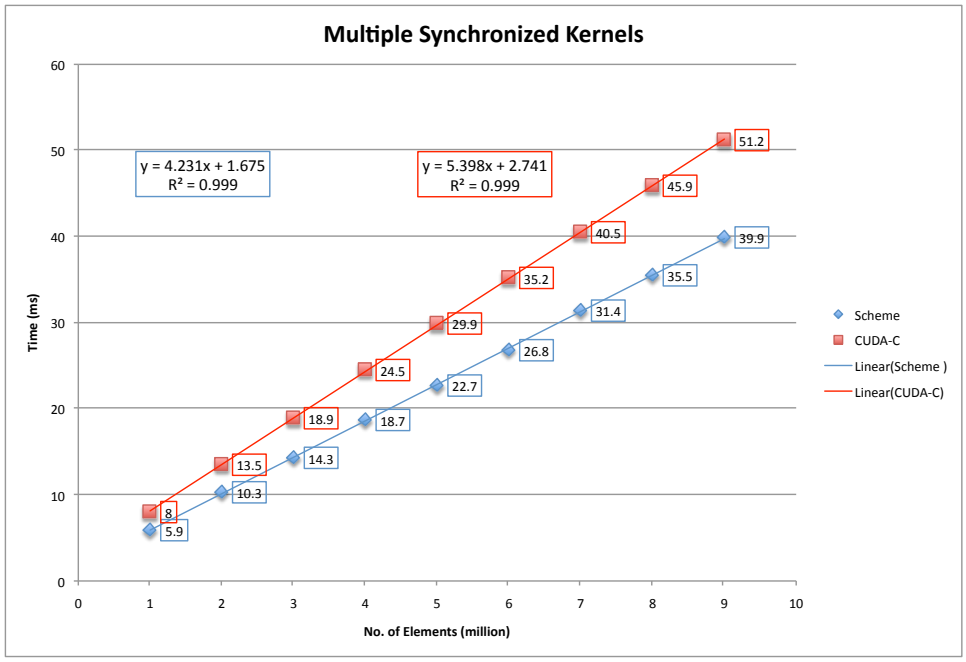


Figure 5.5: Performance comparisons of two synchronized kernels

for both implementation, which means that both lines fit the data perfectly.

We ran the same kernel with different execution configurations in this experiment and kept the grid size fixed to 65535 thread blocks while the block size increased with increasing vector size. The chart in Figure 5.6 compares execution times in Scheme with CUDA-C. Here, we observed that both Scheme and CUDA-C showed liner trends $y = 4.066x + 2.9$ and $y = 5.236x + 3.894$, respectively. The time difference between Scheme and CUDA-C was $\Delta y = -1.17x - 0.994$. We also observed no overhead in Scheme compared to CUDA-C. Note that the R Squared value was 0.990 for both implementations, which means that the both lines fit the data perfectly. Note that the Scheme implementation of this test case is provided in Appendix B.

5.4 Reduction

This test case performed a parallel sum reduction [44] on a vector of 32-bit unsigned-integers to calculate a single sum. Parallel reduction is a common technique that used to compute a result in a set of data. Reduction can be used when a vector of values needs to be reduced to a single value using any binary associative operator such as maximum, minimum, average or product from a set of numbers.

This example used multiple thread blocks to process very large vectors. Each thread block performed reduction to compute a partial sum for a segment of that array. Finally, a thread block performed reduction on the partial sums to compute the overall sum. Therefore, we used two synchronized kernels to compute the overall sum. The first kernel was invoked twice to compute a partial sum for all the participating thread

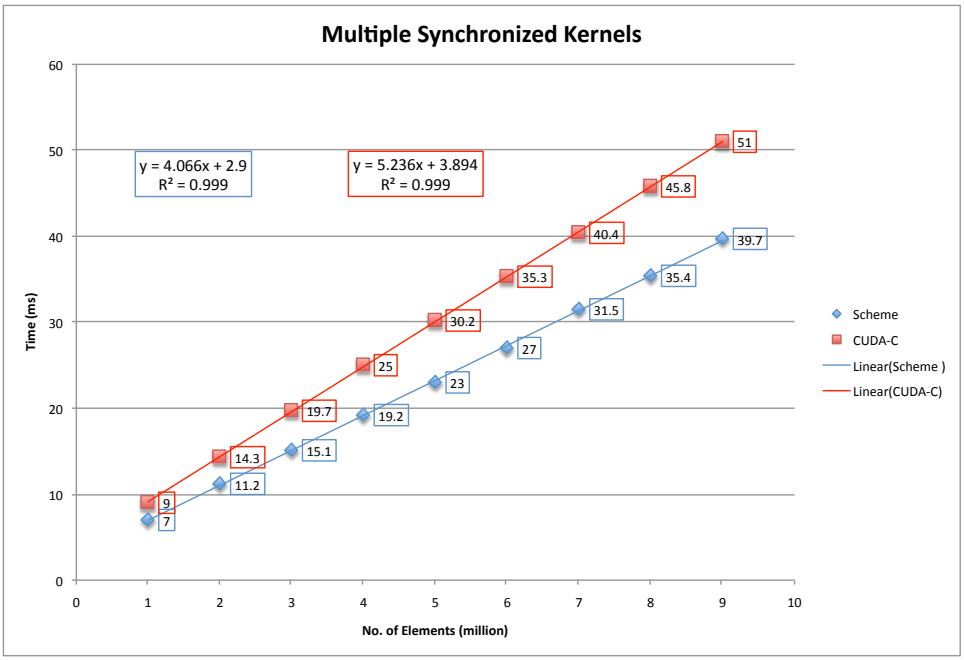


Figure 5.6: Performance comparisons of two synchronized kernels

blocks. Here, 32 threads block were organized into a one-dimensional grid where each thread block containing 512 one-dimensional threads ran this kernel in parallel. Another kernel was then invoked to compute, the overall sum from the partial sums computed by the first kernel. This kernel was run by a single, one-dimensional thread block containing 32 threads. These two kernels must be synchronized to compute the final sum. Therefore, the second kernel must not start until the first kernel is finished. In this test case, both kernels were optimized to improve overall performance.

The chart in Figure 5.7 compares sum reduction implemented in Scheme and CUDA-C. In this chart, the X-axis represents sizes for input vectors in 1024 or 1K units and is in logarithmic scale. The Y-axis represents execution times in milliseconds. Execution times for Scheme and CUDA-C are represented with blue diamonds and red squares, respectively.

In this example, both Scheme and CUDA-C showed polynomial trends $y = -2E-09x^2 + .0004x + 2.231$ and $y = -2E-09x^2 + .0004x + 1.560$, respectively. The trend line for Scheme never met the trend line for CUDA-C. The difference in execution time between Scheme and CUDA-C for this example was $\Delta y = 0.671$; this included three Scheme shims and six allocation/deallocation operations. Note that the R Square value was 1.0 for both implementations, which means that the both lines fit the data almost perfectly. We found that overhead in Scheme decreased gradually with increasing vector size. Initially overhead was 35% for vector size 128K and gradually decreased to less than 1% for vector size 32768K.

In this test case, the fist kernel was called twice and second kernel was called only once. Both kernels take two vectors. Therefore, twelve memory allocation/deallocation operations for the vectors were executed in the CUDA-C shims for these two kernels. However, the CUDA-C implementation had six extra

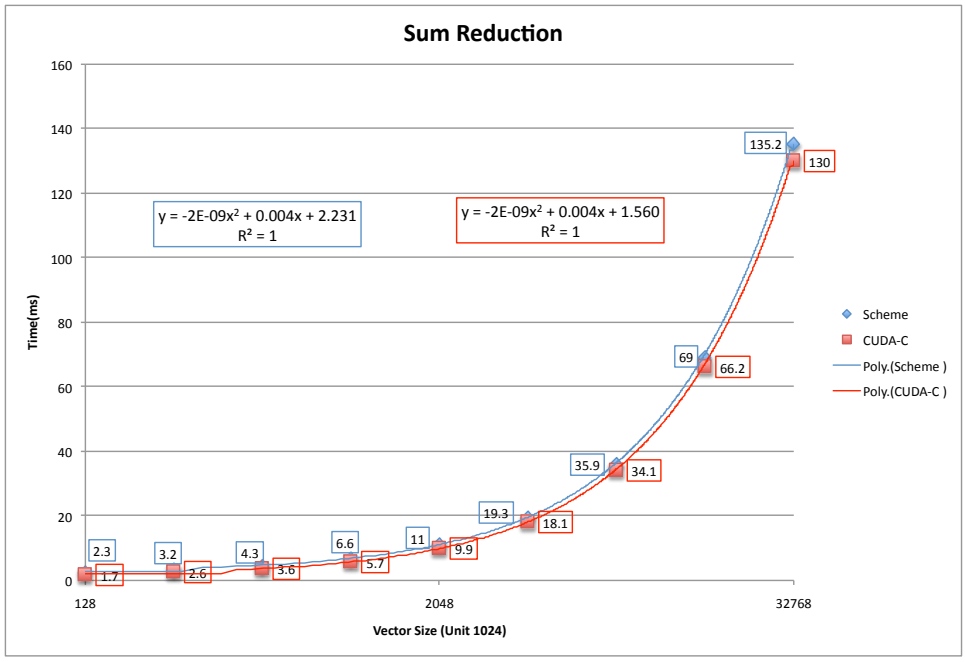


Figure 5.7: Performance comparisons of sum reduction

allocation/deallocation operations, which contributed to overhead in the Scheme implementation.

For this test case, we also compared performance in Scheme with CUDA-C for an increasing number of threads, as in Figures 5.8–5.11. In these charts, the X-axis represents the number of participating thread blocks in a grid. In this experiment, we ran kernels for vector sizes 4M, 8M, 16M and 32M.

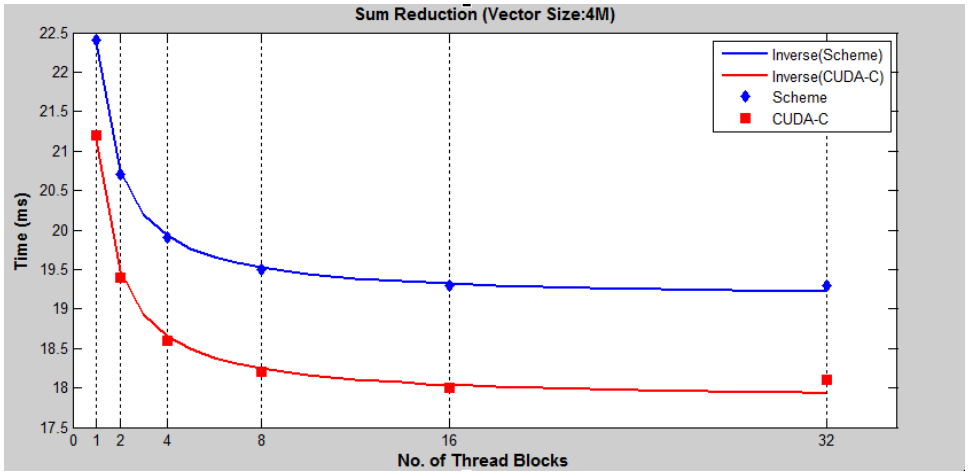


Figure 5.8: Performance comparison of sum reduction

As shown in Figure 5.8, we ran the kernel for vector size 4M. We found that both Scheme and CUDA-C showed inverse trends $y = 19.115 + 3.225/x$ and $y = 17.831 + 3.309/x$, respectively. The R Square values for Scheme and CUDA-C were 0.998 and 0.994, respectively, which means that both lines fit the data almost perfectly. From this trend line we observed that execution times for both Scheme and CUDA-C decreased

gradually but implementation execution times remained consistent after grid size 4. We found a consistently small overhead in Scheme of between 5–7% for the grid sizes on the X-axis. For this experiment, we found that $\Delta y = 1.284 - 0.054/x$; this included three Scheme shims and six allocation/deallocation operations.

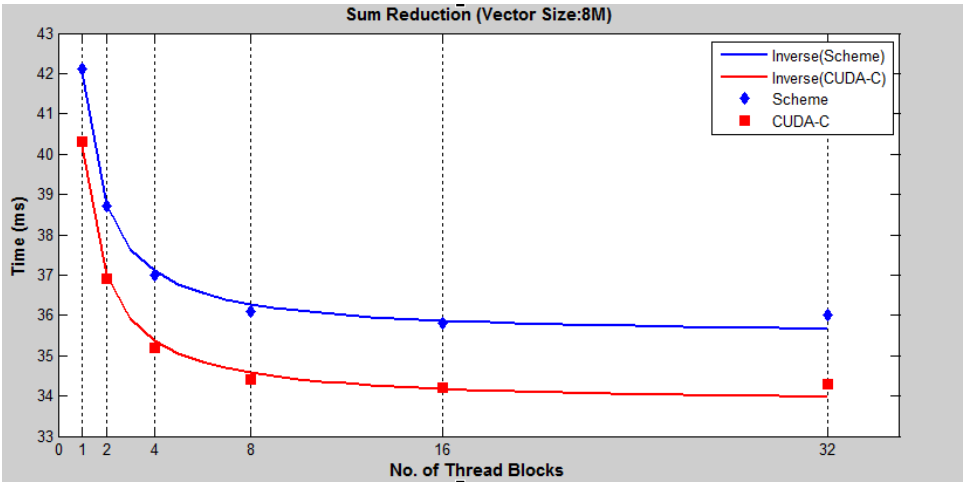


Figure 5.9: Performance comparison of sum reduction

In Figure 5.9, the kernel was called with vector size 8M. Here, we also found that both kernels showed inverse trends $y = 35.459 + 6.575/x$ and $y = 33.775 + 6.425/x$, respectively. The R Square values for Scheme and CUDA-C were 0.994 and 0.993, respectively, which means that both lines fit the data almost perfectly. Here we also observed that execution times for both Scheme and CUDA-C decreased gradually but implementation execution times remained consistent after grid size 4. Here, we also found consistently small overhead in Scheme of between 4–5% for the grid sizes along the X-axis. For this experiment, we found that $\Delta y = 1.684 + 0.15/x$; this also included three Scheme shims and six allocation/deallocation operations.

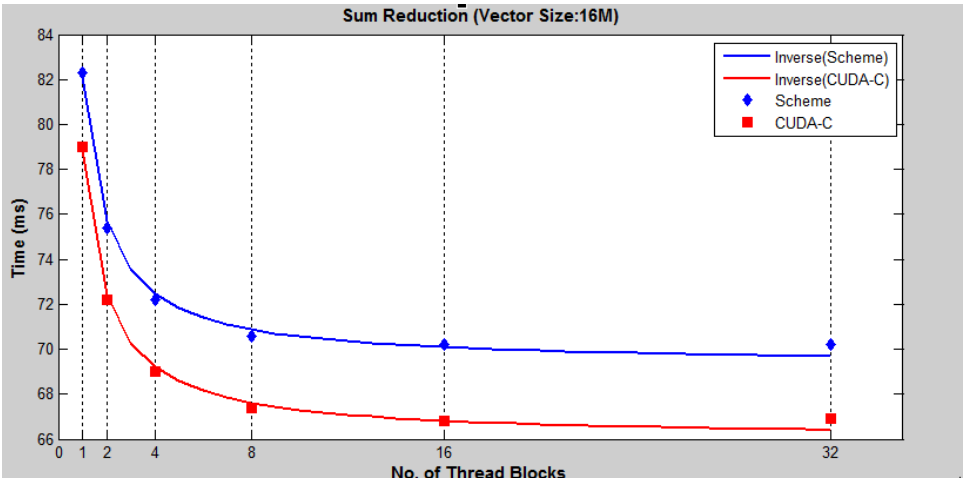


Figure 5.10: Performance comparison of sum reduction

In Figure 5.10, the kernel was called with vector size 16M. Here we found that both kernels showed inverse trends $y = 69.287 + 12.788/x$ and $y = 66.013 + 12.811/x$, respectively. The R Square values for Scheme and

CUDA-C were 0.995 and 0.996, respectively, which means that the both lines fit the data perfectly. We also observed that execution times for both Scheme and CUDA-C decreased gradually but that implementation execution times remained consistent after grid size 4. We also found a consistently small overhead in Scheme of between 4–5% for the grid sizes along the X-axis. For this experiment, we found that $\Delta y = 3.274 - 0.022/x$; this included three Scheme shims and six allocation/deallocation operations.

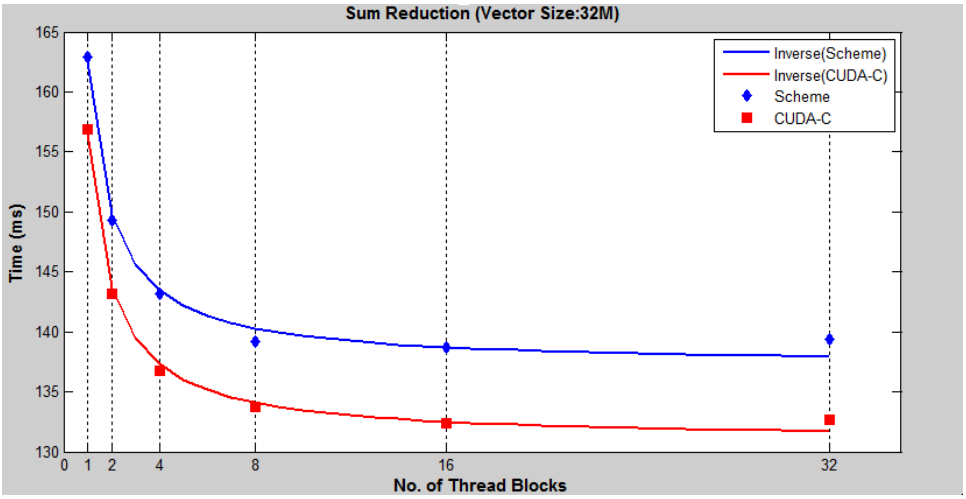


Figure 5.11: Performance comparison of sum reduction

In Figure 5.11, the kernel was called with vector size 32M. Here we found that both kernels show inverse trends $y = 137.130 + 25.357/x$ and $y = 130.899 + 25.605/x$, respectively. The R Square values for Scheme and CUDA-C were 0.991 and 0.995, respectively, which means that the both lines fit the data perfectly. We also found consistently small overhead in Scheme of between 3–5% for the grid sizes along the X-axis. For this experiment, we found that $\Delta y = 6.231 - 0.248/x$; this included three Scheme shims and six extra allocation/deallocation operations.

From these four experiments we observed that both execution times reduced with an increasing number of thread blocks. We also observed that after a certain number of thread block there was no improvement in performance for both implementations. We found a consistently small overhead in Scheme compared to CUDA-C across an increasing number of thread blocks and an increasing number of vector sizes. The reason for this consistent overhead was the six extra allocation/deallocation operations in the generated CUDA-C shim since execution times for the generated Scheme shim, since execution times for the generated Scheme shims were negligible. Note that the Scheme implementation for this test case is provided in Appendix C.

5.5 Matrix Multiplication

In this test case, the supplied CUDA-C kernel [16] performed parallel matrix multiplication on two square matrices of 32-bit floating-point numbers and stored results in another square matrix. Here, all three matrices were identical in size. In this test case, participating threads were organized into square thread blocks, and

thread blocks were organized into a two-dimensional square grid. Each thread block performed matrix multiplication on one-dimensional segments of the two matrices. Therefore, each participating thread in a thread block looped through a row of a segment of the first matrix and a column of a segment of the second matrix based on its two-dimensional thread ID. Each thread then multiplied an element from a row with the corresponding element in a column and summed up the total for all multiplications. Next, it stored the summation in the third matrix in a position determined based on its two-dimensional index. In this example, the first two matrices have `IN` notations and the third matrix has an `OUT` notation in their names in order to avoid unnecessary memory transfer operations.

Table 5.2: Execution configurations used for different vector sizes seen on the X-axis of Figure 5.12

Size of vector	Block Dimension	Grid Dimension
288 × 288	8 × 8	9 × 9
576 × 576	16 × 16	9 × 9
1152 × 1152	16 × 16	18 × 18
2304 × 2304	32 × 32	18 × 18
4608 × 4608	32 × 32	36 × 36
9216 × 9216	32 × 32	72 × 72

We ran the kernel with different vector sizes in this text case. We also used different execution configurations for different vector sizes, as shown in Table 5.2. In this test case we used a two-dimensional grid and two-dimensional thread blocks to run this kernel.

In Figure 5.12 we provide a chart that compares performance of matrix multiplications implemented in Scheme with CUDA-C. In this chart, the X-axis represents the widths of input and output vectors and the Y-axis represents execution times in milliseconds. Note that Y-axis is in logarithmic scale for this test case. In this example, we doubled the widths of the matrices for scaling. Both dimensions of the matrices are doubled since the matrices are square matrices. Therefore, vector sizes are quadrupled because of this scaling behavior.

In this test case, both Scheme and CUDA-C implementations followed power trends $y = 2E-07x^{2.817}$ and $y = 4E-07x^{2.730}$, respectively. The execution time difference between Scheme and CUDA-C was $\Delta y = -2E-07x^{0.087}$; this included the generated Scheme shim. In this test case, we observed no overhead in Scheme and the trend line in Scheme never crossed the trend line in CUDA-C. However, it met the CUDA-C trend line after matrix width 4608. cuShim+kernel also showed the power trend $y = 2E-07x^{2.819}$ which is almost the same as with Scheme. It suggests that most of the execution time was consumed by the CUDA-C shim and the kernel. In this chart, the trend line for Scheme is overlapped by the trend line for cuShim+kernel [1]. Therefore, execution time for the generated Scheme shim was also negligible in this test case. Note that the R Square values for these three timings are 0.994, 0.993, and 0.994; this means the lines

[1] We can only see the kiwi trend line for cuShim+kernel in Figure 5.12

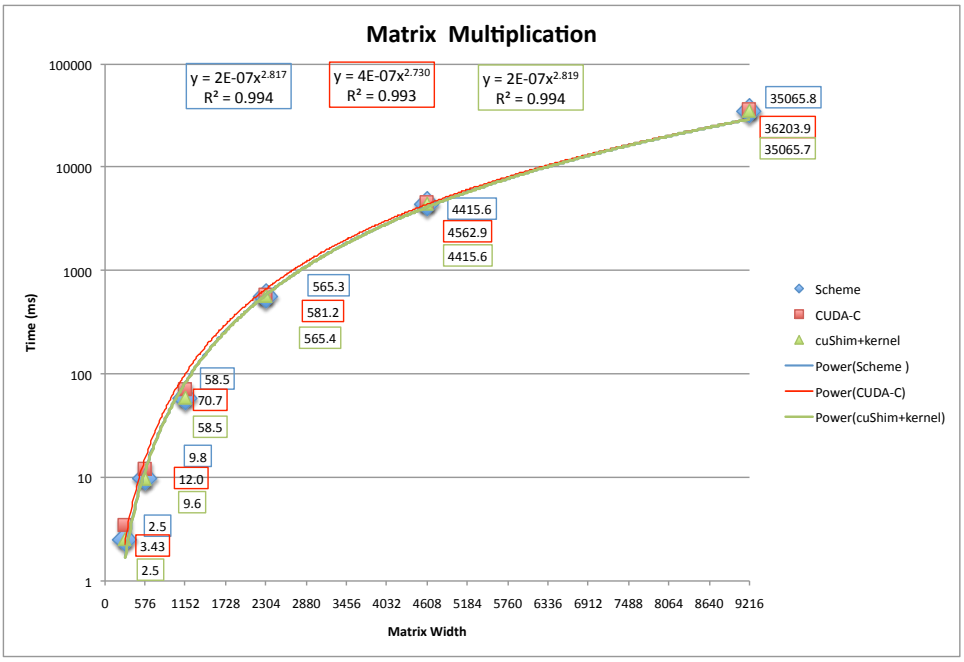


Figure 5.12: Performance comparison of parallel matrix multiplication

fits the data almost perfectly.

For this test case, we also measured execution time for the supplied kernel only in both Scheme and CUDA-C. The chart in Figure 5.13 compares execution times for the kernel in Scheme with CUDA-C. Both Scheme and CUDA-C showed power trends $y = 8E-08x^{2.925}$ and $y = 2E-07x^{2.801}$, respectively. The difference in kernel execution time between Scheme and CUDA-C was $\Delta y_k = -1.2E-07x^{0.124}$. Therefore, execution time for the supplied kernel was less when called from Scheme code compared to CUDA-C. In this example, the initially Scheme trend line maintained a distance from the CUDA-C trend line before it met this line smoothly smoothly after matrix width 4608 on the X-axis.

For this test case, we also ran the same kernel without `IN` and `OUT` notations in the names of the kernel parameters to observe its performance without `IN/OUT` notations. We found that there was no overhead in Scheme compared to CUDA-C, as shown in the chart in Figure 5.14. Both Scheme and CUDA-C showed power trends $y = 3E-07x^{2.754}$ and $y = 4E-07x^{2.730}$, respectively. The execution time difference between Scheme and CUDA-C was $\Delta y = -1E-07x^{-0.007}$; this included Scheme shim and three extra memory transfer operations. Here, extra execution time without IN/OUT notation was $\Delta y_{IN/OUT} = 1E-7x^{-0.273}$ and included three extra memory transfer operations. The generated cuShim+kernel also showed power trend $y = 3E-07x^{2.761}$; this is similar to Scheme, so the execution time for generated Scheme shim was also negligible here. We found that execution times in Scheme decreased by around 0.7% –16.4% in this experiment compared to execution times with `IN` and `OUT` notations in Figure 5.12. The reason is without `IN` and `OUT` notations our implementation generated three extra memory transfer operations in the CUDA-C shim. Note that the Scheme implementation of this test case is provided in Appendix D.

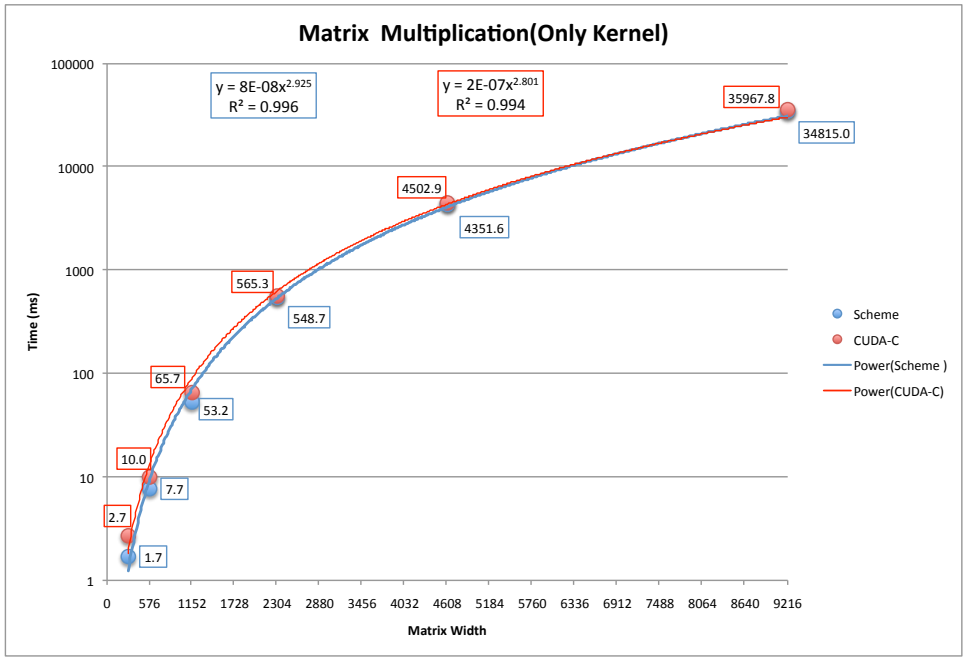


Figure 5.13: Performance comparison of matrix multiplication for CUDA-C kernel

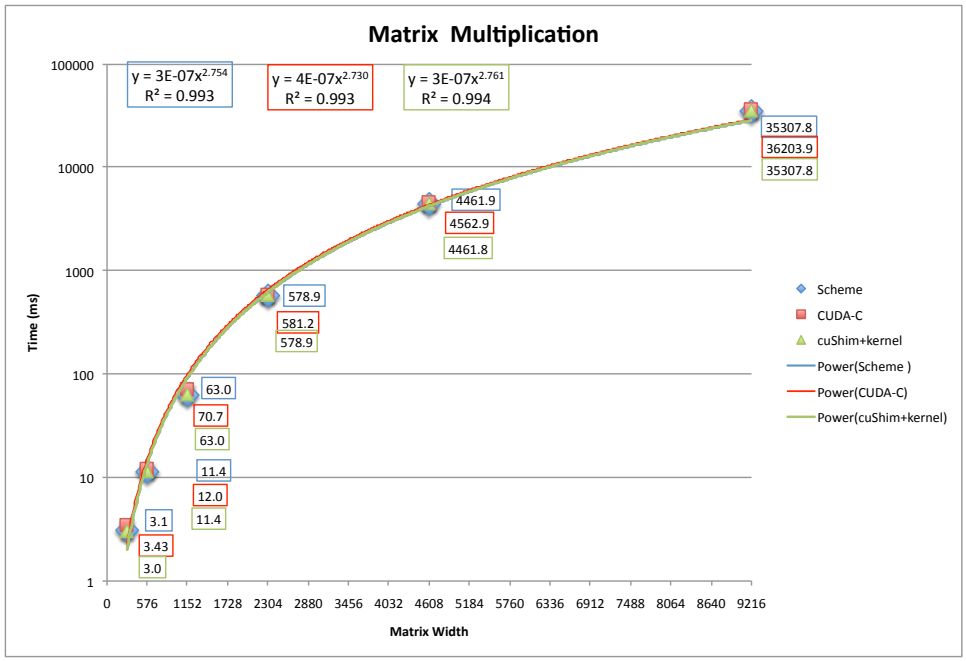


Figure 5.14: Performance comparison of parallel matrix multiplication without `IN/OUT` notations

5.6 3D Finite Difference computing

In this test case, we linked a kernel that performed parallel 3D finite difference computations [57] from Scheme. We chose this example to evaluate runtime performance of our implementation in complex and real world applications. This example illustrated a 3D stencil computation [50] over a uniform grid [55]. Stencil-only computation is a common computational technique used in finite difference codes. This technique can also be used for finite difference discretization of wave equation, which is a major building block for *Reverse Time Migration (RTM)* [25], used in seismic imaging.

In this example, the kernel processed two-dimensional thread blocks along the slowest varying dimension of the three-dimensional data set in order to maximize data reuse from shared memory. In this test case, 32 $\times$ 32 thread blocks are organized into a two-dimensional grid. Each two-dimensional thread block contains 16 $\times$ 16 threads. Two vectors - each 703MB in size - contained 32-bit floating-point numbers were passed to the kernel. Here, one vector contained initial data and other vector contained the results.

Table 5.3: Assessment of runtime (ms) performance of parallel 3DFD

Programs	Mean(ms)	σ (ms)
CUDA-C Implementation	214.1	0.9
Shims+Kernel	276.8	1.6
Shim(CUDA-C)+Kernel	276.8	1.6
Kernel(Scheme)	32.9	0.2
Kernel(CUDA-C)	30.9	0.4

In this test case, Scheme implementation (Shims+Kernel) created 29% or 62.7 ms overhead compared to its CUDA-C implementation, as seen in Table 5.3. Here, the Scheme implementation of this example included execution times for generated Scheme shim, CUDA-C shim and the supplied CUDA-C kernel from Scheme. The combined execution time for the CUDA-C shim and the CUDA-C kernel in Scheme implementation was 276.8 ms. Therefore, overhead contributed by the Scheme shim was negligible in this test case. We also found that when the CUDA-C kernel was linked from Scheme its execution time was 2.00 ms (32.9 - 30.9 = 2.0) longer on average compared to its execution times in the CUDA-C implementation. This extra time created about 3% of overall overhead. Therefore, almost 97% (62.7 - 2.0 = 60.7) of overall overhead in this test case was caused by the CUDA-C shim, including memory transfer operations, allocation and deallocation operations in device memory, and a call to the CUDA-C kernel.

In this example, the resultant vector did not need to be copied from host to device memory before kernel execution. Similarly, the vector containing initial data did not need to be copied back to host memory after execution of the kernel. In order to do this we provided an `IN` notation in the name of the vector containing initial data. We also provided an `OUT` notation in the name of the vector containing results to avoid two extra memory transfer operations. We found that without these two notations in the names of kernel parameters, execution times in Scheme increased to 433.7 ms. This is almost double compared

to the CUDA-C implementation (214.1 ms). Therefore, `IN` and `OUT` special notations are needed to avoid extra memory-transfer operations that are cause huge overhead for this test case. Note that the Scheme implementation for this test case is provided in Appendix E.

5.7 Scalar Product

In this test case, we linked Scheme code to a CUDA-C kernel that performed parallel scalar product on GPU. This kernel was taken from NVIDIA's CUDA SDK [9]. In this example, a kernel takes an input vector pair containing floating-point numbers and an output floating-point vector. Both input vectors were divided into equal number of segments. Here, the kernel calculated a scalar product for each segment pair from both input vectors and stored resultant scalar products on the output vector. Therefore, the length of the output vector was equal to the number of segments of input vectors. In this example, each input vector was divided into 2560 segments. Therefore, CUDA-C kernel calculated 2560 scalar products.

In this test case, we measured execution times against different numbers of participating thread blocks. We changed the grid size in execution configuration for a particular size of input vector pairs. In this test case we measured execution time for an increasing number of threads with a fixed vector size. We also increased vector sizes linearly to observe the scaling behavior in the separated charts.

The charts in Figures 5.15 –5.18 show performance comparisons of parallel scalar product implemented in Scheme and CUDA-C for four different vector sizes. In these charts, the X-axis represents number of thread blocks per grid and the Y-axis represents execution time in milliseconds. In the execution configuration, each one-dimensional thread block had 256 threads. Therefore, the number of participating threads for a particular execution time can be found by multiplying grid size with the fixed block size.

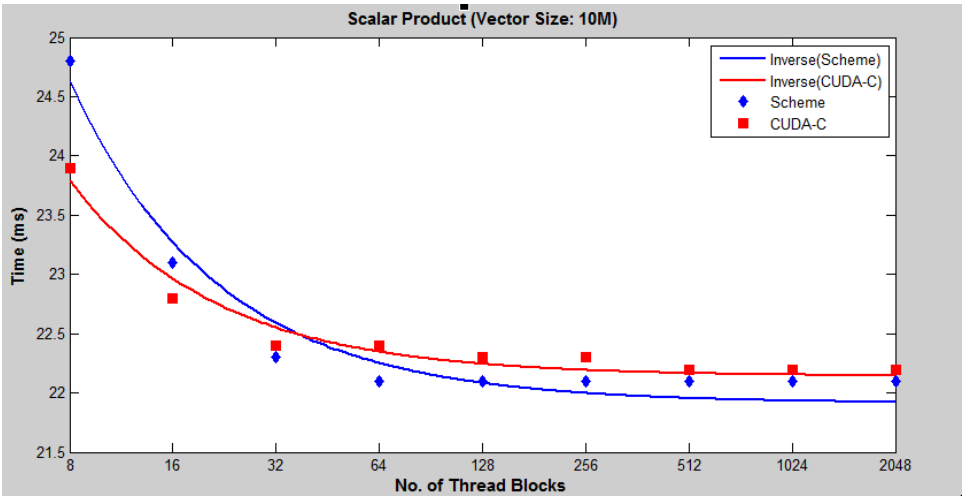


Figure 5.15: Performance comparison of parallel scalar product

First, the kernel was called with a vector size 10M for both input vectors. In Figure 5.15 both execution times for Scheme and CUDA-C showed inverse trends $y = 21.912 + 21.744/x$ and $y = 22.140 + 13.208/x$,

respectively. The R Square values in Scheme and CUDA-C were 0.964 and 0.969, respectively, this means that both lines fit the data almost perfectly. By seeing both trend lines we understand that both execution times reduced with an increasing number of thread blocks. Initially, Scheme took longer to execute than CUDA-C. With an increasing number of thread blocks along the X-axis, the distance between these two trend lines decreased. This means the difference between the two execution times is also reduced. Finally, the Scheme trend line crossed the CUDA-C trend line after 32 thread blocks on the X-axis. Scheme then takes less time compared to CUDA-C for the rest of the values on the X-axis. For this experiment, we found that $\Delta y = -0.228 + 21.744/x$ included the generated Scheme shim. We observed that initially there was a 1–4% overhead in Scheme for grid sizes 8 to 16, but that there was no overhead in Scheme from grid size 32 on. Both Scheme and CUDA-C then showed consistent execution times.

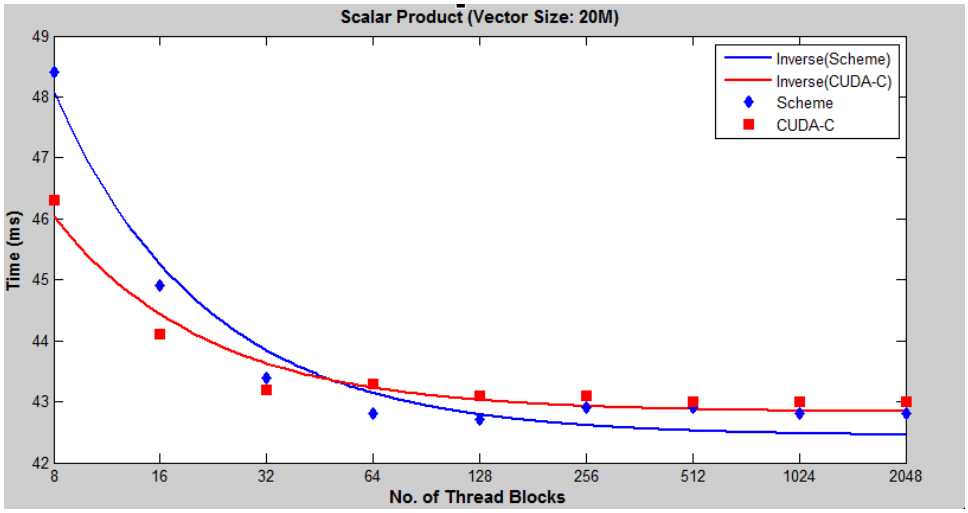


Figure 5.16: Performance comparison of parallel scalar product

In Figure 5.16, we ran the same programs for both implementations but we doubled the vector size to 20M. We also found that both implementations showed inverse trends, with $y = 42.435 + 45.134/x$ in Scheme and $y = 42.825 + 25.752/x$ in CUDA-C. The R Square values for Scheme and CUDA-C were 0.970 and 0.954, respectively, which is a good fit of both lines to the data. We observed that the execution times for both implementations were double compared to the execution times in Figure 5.15. This is because vector size doubled in this experiment. This chart also shows the same behavior as the previous chart. Initially, Scheme implementation took longer than CUDA-C. As the grid size increased, distance between two execution times reduced until the Scheme trend line crossed the CUDA-C trend line after the thread block 32 on the X-axis. Scheme then took less time compared to CUDA-C for the rest of the values along the in X-axis. For this experiment, we found that $\Delta y = -0.39 + 19.382/x$; this included the Scheme shim. We also found that 1–5% overhead in Scheme compared to CUDA-C for grid sizes 8 to 16. At grid size 32. the overhead became almost 0% in Scheme and then execution times for both implementations were almost consistent across the grid sizes on the X-axis.

In Figure 5.17, the kernel was called with vector size 30M for two input vectors. This tripled the vector size compares in Figure 5.15. Here, we also found both Scheme and CUDA-C show inverse trends $y = 63.198 + 65.123/x$ and $y = 63.123 + 38.832/x$, respectively. The R Square values for both Scheme and CUDA-C were 0.969 and 0.953, respectively. This is a relatively good fit of both lines to the data.

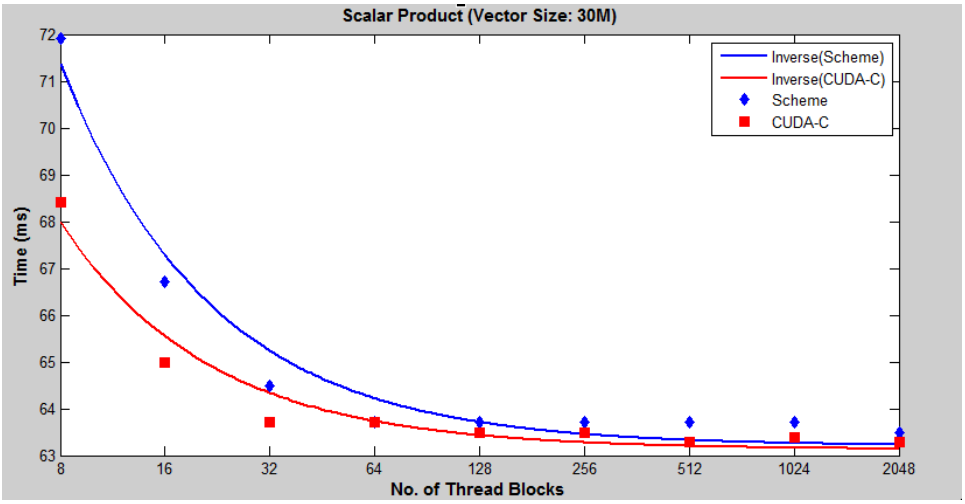


Figure 5.17: Performance comparison of parallel scalar product

Here, we also found that initially Scheme had longer execution times compared to CUDA-C and gradually the difference between Scheme and CUDA-C reduced on the Y-axis. After grid size 32 both trend lines kept consistent distance across the values on the X-axis. In this experiment, the Scheme trend line never met the CUDA-C trend line. For this experiment we found that $\Delta y = 0.074 + 26.58/x$ and contained only the Scheme shim. We also found an initial 1–5% overhead in Scheme for grid sizes 8 to 32 and almost 0% overhead after grid size 64. In this experiment, we observed that 0% overhead in Scheme drifted to the right on the X-axis compared to Figures 5.15 and 5.16. For these experiments, we found 0% over head at grid size 32.

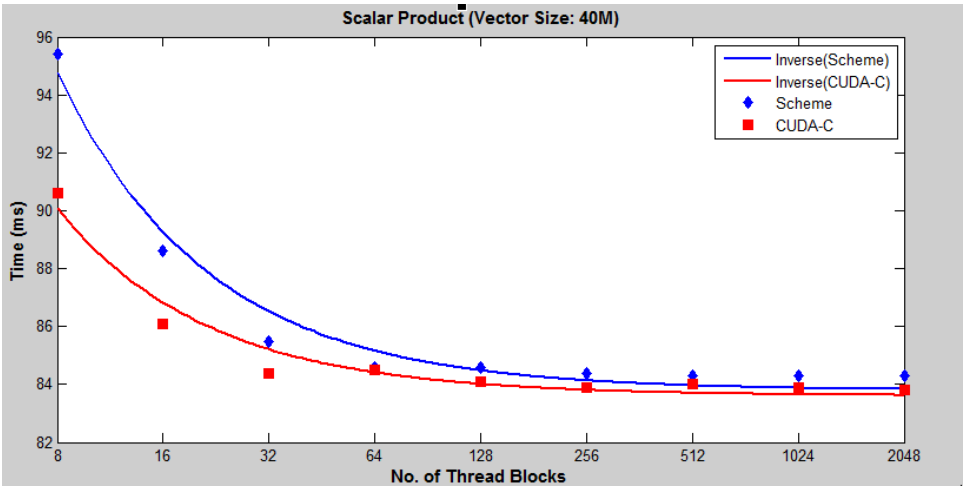


Figure 5.18: Performance comparison of parallel scalar product

In Figure 5.18, the kernel was called with the vector size 40M for two input vectors. The vector size is now four times bigger compared to the vector size in Figure 5.15. Here, we observed that both Scheme and CUDA-C showed inverse trends $y = 83.789 + 87.781/x$ and $y = 83.601 + 51.662/x$, respectively. The R Square values for Scheme and CUDA-C were 0.974 and 0.956. This is a good fit of both lines to the data. We also observed that initially Scheme had longer execution times compared to CUDA-C. Gradually, the difference between these two execution times reduced. After grid size 32, both trend lines kept consistent distance across the X-axis values. In this experiment, the Scheme trend line never met the CUDA-C trend line and we found that $\Delta y = 0.18 + 36.119/x$. We also found a 1–6% overhead in Scheme implementation for grid sizes 8 to 32. After grid size 64, we observed almost 0% overhead in Scheme, as seen in Figure 5.18.

In this example we observed a 0–5% overhead in Scheme. We also found that overhead in Scheme reduced with increased grid sizes on the X-axis and after a certain grid size, overhead became almost 0%. We also found that increasing the grid size did not always reflect better performance for both implementations. After a certain grid size we repeatedly observed similar execution times. We also found that increasing vector size required more thread participation to achieve optimal performance in Scheme.

5.8 Related Work

5.8.1 GPGPU-Based Systems

The Lua programming language has an extension to access data-parallel OpenCL code known as *LuaGPU*[19]. It allows programmers to write host programs in Lua. LuaGPU saves programmers from a lot of error-prone error-checking and pointer-operations in OpenCL. Unlike our implementation, a host program in LuaGPU takes a data-parallel OpenCL kernel source code as a Lua string. Programmers also need to write their own datatype mapping operations to pass kernel arguments from a Lua program to an OpenCL kernel, whereas our system automatically generates these mapping operations in the Scheme shim. In addition, LuaGPU requires programmers to write out the memory transfer operations which are also automatically generated in the CUDA-C shim in our system.

Moreover, data and kernel are launched from a queue-like data structure defined in Lua, whereas our system does not require any additional data structure to call a kernel. A kernel call is like an ordinary function call in Scheme. It just requires an additional construct to define an execution configuration. Therefore, LuaGPU still leaves a lot of Lua coding for programmers.

The Python programming language also has an extension to access data-parallel CUDA-C code called *PyCUDA* [51]. This extension allows programmers to write host programs in Python that control and issue data-parallel programs written in CUDA-C. Unlike our system, PyCUDA requires programmers to define memory transfer operations, allocation/deallocation operations, and it also takes data-parallel CUDA-C code as Python strings. PyCUDA provides special arrays for GPUs, whereas our system does not require any additional special GPU arrays in Scheme. An ordinary Scheme vector can be passed to a kernel.

Our system does not provide any error-reporting facility, whereas in PyCUDA, errors generated from GPU computations are detected and reported automatically. In contrast to our system, every feature of the CUDA runtime system is accessible from Python via PyCUDA, including textures, pinned-host memory, OpenGL interaction, zero-copy host memory mapping, etc. PyCUDA also provides some library functionality such as element-wise arithmetic-operations, map-reduce, and parallel scan that allows a restricted subset of Python code to be automatically farmed out to the GPU. In addition, PyCUDA has a just-in-time compiler that generates NVIDIA's low-level *PTX* abstract-machine code [6] which allows automated tuning of device code to improve runtime performance, whereas our system does not generate code at runtime.

PyCUDA has been used successfully in many research projects. Tomasz Rybak at Bialystok Technical University uses PyCUDA for generating recurrence diagrams for time-series analysis. He was able to achieve an 85-fold speedup compared to CPU computations. Romain Brette and Dan Goodman are also using PyCUDA to simulate spiking neural networks with their simulator *Brian* [39]. Brian relies on PyCUDA to generate runtime GPU code for the integration of differential equations provided by the users in a Python script. GPU performance was up to 60 times faster than a comparable CPU implementation for some models. There are some image processing applications developed with PyCUDA that implement k-means clustering routines [27]. For those applications, PyCUDA was about 10x slower than the CUDA-C implementations but these were still probably an improvement over CPU computations. In contrast to PyCUDA, overhead in our system for the test case 3DFD as a real-world example, discussed in Section 5.6, was 1.3x slower than the CUDA-C implementation.

Accelerate [28] is a domain-specific high-level skeleton-based language for GPGPU computing in the Haskell programming language. In Accelerate, both host and kernel programs are written in high-level Haskell, whereas our system allows only host programs in Scheme as high-level language. Accelerate provides abstractions for the programmers both for device and host programs that eases GPU programming. Unlike our system, Accelerate has a dynamic code generator that instantiates CUDA implementations as PTX at runtime. This code generator exploits runtime information to optimize GPU code. However, compiling kernels at runtime is an overhead at execution time. In order to reduce this overhead, Accelerate memoizes compiled kernels. Therefore, kernels that are invoked multiple times are only generated and compiled once.

In [28], Manuel M.T. Chakravarty et al. mention three test cases in order to evaluate performance of Accelerate. For parallel dot product, Accelerate takes almost precisely twice as long as CUDA-C, whereas overhead for this test case in our system was only 0–4% compared to CUDA-C. Another test case, the Black-Scoles option pricing algorithm shows that the overhead for Accelerate reduces with increasing vector sizes compared to CUDA-C. Similarly, sparse-matrix vector multiplication also shows similar performance behavior to Black-Scholes as overhead reducing compared to CUDA-C with increasing vector sizes. In our system, we also observed similar performance behavior, diminishing overhead rapidly with the increasing vector sizes for the test case parallel sum reduction, discussed in Section 5.4.

Firepile [56] is a library for GPU computing in Scala programming language which has both functional and

object-oriented features. Firepile allows programmers to write both host and kernel code in Scala, whereas our system allows only host programs in Scheme as high-level language. Like our system, the Firepile library hides details of GPU programming by managing devices and memory operations automatically. However, Firepile is a library to manage devices and memory operations, whereas our system generates code for the memory operations in the shims and GPUs are managed by the library functions in Scheme provided to our system.

In order to compile from Scala to OpenCL, Scala compiler first compiles both host and kernel into Java bytecode. Then the Java Virtual Machine executes the bytecode. Next, the Firepile library identifies the bytecode for the kernel and invokes its internal compiler to convert kernel bytecode to native OpenCL code. Next, Firepile copies data from host to device memory and then invokes the kernel. Finally, it copies back results from device to host memory.

In [56], Nathaniel Nystrom et al. mention five test cases: reduction, Black-Scholes, matrix multiplication, the discrete cosine transform (DCT8x8), and matrix transpose. The Firepile version of parallel reduction performed as well as CUDA-C. In our system, parallel reduction showed 35% overhead for small vector sizes. However, this overhead diminished rapidly with increasing vector sizes. For matrix multiplication, Firepile version was 15% faster than the NVIDIA CUDA-C version. In our system, matrix multiplication was also faster than the CUDA-C version. Firepile versions of discrete cosine transform (DCT8x8), Black-Scholes and matrix transpose were consistently slower than the CUDA-C versions. In contrast to Firepile, test cases for our system initially showed overhead for smaller vector sizes. However, we observed that overhead diminished rapidly with increasing vector sizes or number of thread blocks.

5.8.2 Concurrent and Distributed Systems

Erlang [20] is a functional programming language designed for programming concurrent, and distributed systems. It is developed by Ericsson to write programs to provide a flexible and safe support for telecommunication equipment. Like Scheme, Erlang is a dynamically typed language and has support for higher-order functions. It also provides C/C++ interfaces to link C code.

In Erlang, processes are the executing elements that run on the Erlang virtual machine. Processes are created and managed by the Erlang runtime system, not by the underlying operating system. Concurrent Erlang processes are totally separate and share nothing (*Multiple Instructions Multiple Data*). But, they are not embarrassingly parallel because they interact with each other only through message passing. Data is also immutable in an Erlang process. Multiple processes can be synchronized only through the message-passing mechanism; and, this is inefficient since messages must be copied among the processes. Erlang also includes sophisticated error-handling, code-replacement mechanisms, and a large set of libraries.

In contrast to Erlang, our system compiles programs for data-parallel computations on GPUs with a large degree of SIMD parallelism. In GPGPU computing, a kernel actually runs on a GPU rather on a virtual machine. Moreover, there is no message-passing mechanism in GPGPU computing to synchronize

multiple kernel executions. Multiple GPU threads can access shared data in parallel. Therefore, our system works for a completely different paradigm than the Erlang; and, it is also true for others GPGPU based systems, discussed in Section 5.8.1. GPGPU computing fundamentally works differently from concurrent and distributed computing. Currently, Erlang has no support for high-performance GPGPU computing.

Gambit Scheme also has a support for concurrent and distributed computing, known as *Termite Scheme* [38]. Termite Scheme is mainly inspired by Erlang. Therefore, Termite processes also communicate through message-passing mechanism; and, it is impossible for a process to directly access the memory space of another process. Termite is flexible enough that the programmers can easily build, and experiment with libraries providing higher-level distribution primitives and frameworks [38].

Moreover, the semantic properties of Termite Scheme allow programmers to write simple yet robust code to build higher layers of abstractions that are themselves clean, maintainable, and reliable [38]. Unlike Erlang, Termite provides macros and continuations for distributed computing; this enables task migration and dynamic code updates. In contrast to Termite, our system extends Gambit to write data-parallel programs only for the NVIDIA's GPUs.

OpenMP [26] is an API to develop multithreaded applications available for C, C++, and FORTRAN. It supports most processor architectures and operating systems. OpenMP consists of a set of compiler directives and library functions that enables *Symmetric Multiprocessing* in C, C++, and FORTRAN. Unlike our system, OpenMP uses a fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional number of threads. This thread is called a *master thread*. The other threads of the team are called *slave threads* of the team.

There are two types of synchronization available in OpenMP: implicit and explicit. Implicit synchronization points exists at the beginning and the end of a parallel construct. When a thread finishes its work, it waits at the implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, then the threads can leave the barrier. This implicit barrier works similar to CUDA synchronization primitives.

Like CUDA, OpenMP also allows all threads to access the same global shared memory. Moreover, every individual thread has it's own private memory. Explicit synchronization is also available in OpenMP through shared data residing in global shared memory allowing interprocess communication. Although there are some similarities between CUDA and OpenMP. However, OpenMP only works on CPUs, whereas our system is designed only to work on NVIDIA GPUs.

5.9 Summary

We provided six test cases implemented both in Scheme and CUDA-C in order to evaluate performance of our implementation in Gambit. This showed 0-35% overhead on average for our test cases case. Our test cases covered various constructs of our implementation in Gambit. We evaluated the performance ranging

from single kernel execution to multiple synchronized kernel executions. We also evaluated performance of simple kernels with a few statements to synchronized kernels with nested loops and conditional statements. We also tested the performance of kernels executed by one-dimensional thread blocks to multi-dimensional thread blocks, and from one-dimensional grid to multi-dimensional grid.

We found that the execution time for a generated Scheme shim was negligible. Most of the execution time in Scheme was occupied by CUDA-C shim and the supplied kernel. We observed that the single kernel executions described in Section 5.6 create high overhead. However, the single kernel executions described in Sections 5.2 and 5.5 did not create any overhead. In the case of multiple synchronized kernels, we found that overhead was caused by extra memory allocation/deallocation operations in CUDA-C shim. We also found that without `IN` and `OUT` notations in the name of kernel parameter performance decreased for the Scheme implementations described in Section 5.5. This was because of extra memory transfer operations in CUDA-C shim. These extra memory transfer operations cause a huge overhead of more than 100% for test case 3DFD in Section 5.6.

We also evaluated performance of our implementation against an increasing number of thread in Sections 5.4 and 5.7. We found a small, consistent overhead of 0–5% in Scheme compared to CUDA-C. For the single kernel execution in Section 5.7, we observed that overhead in Scheme reduces with an increasing number of participating thread blocks. After a certain number of thread blocks we found that performance in Scheme surpassed CUDA-C. However, with increased vector sizes Scheme can not surpass CUDA-C but consistently showed almost 0% overhead. In case of multiple synchronized kernel executions, we also found consistent overhead in Scheme implementations because of extra allocation/deallocation operations in device memory. Although execution times for Scheme implementations reduced with an increasing number of thread blocks, execution times for CUDA-C implementations also reduced. Therefore, we observed consistently small overheads for vector size across the X-axis.

Our test cases showed that overhead rapidly reduced to almost 0% with increasing vector sizes as well as increasing number of thread blocks as described in Sections 5.4 and 5.7. Therefore, overhead in days/weeks-long real-world applications can also be reduced reasonably by increasing vector sizes or number of thread blocks.

Chapter 6

Conclusion

This chapter summarizes our work that linking Scheme code to data-parallel CUDA-C kernels in Gambit. We begin with a brief review of our thesis, followed by outlining the contributions of our work, and identifying some future research from our implementation.

6.1 Summary

Data-parallel computation in GPUs for scientific and engineering applications is becoming popular. NVIDIA Corporation provides CUDA-C programming language to develop data-parallel programs - also known as kernels - for GPUs. The execution of these kernels is managed and issued by a host program that runs on a CPU. Therefore, programmers must manage memories both for GPUs and a CPU.

Scheme is a mostly functional programming language. It is also an expressive language. Therefore, programs developed in Scheme are easy to maintain and understand. In this thesis, we showed how a Scheme program can be linked to a data-parallel CUDA-C kernel. In our work, the Gambit Scheme compiler generates an interface (Scheme and CUDA-C shims) from a kernel skeleton defined in Scheme. This kernel skeleton acts as a representative in Scheme for a data-parallel CUDA-C kernel supplied by programmers and the generated shims manage the memory operations. Therefore, our work reduces hands-on memory management and enables developing expressive host programs in Scheme for managing and issuing data-parallel programs running on GPUs.

Our implementation in Gambit generates a foreign-function interface from a kernel skeleton defined in Scheme in order to link Scheme code to a data-parallel CUDA-C kernel. In Chapter 3, we described what parts are required for an interface that links a CUDA-C kernel from Scheme. From our investigation we found four parts to be necessary:

1. A vector-length-calculation helper function for calculating length of a vector
2. A `c-lambda` function for converting kernel arguments from Scheme to C types
3. A forward declaration for a CUDA-C shim using a `c-declare` construct
4. A C-function for managing memory operations and calling a CUDA-C kernel

We named helper function, `c-lambda` function, and forward declaration together as Scheme shim and the C-function that calls a CUDA-C kernel as CUDA-C shim. These two shims are separated into two different files: Scheme shim are in a file with a `.scm` extension and CUDA-C shim are in a file with a `.cu` extension. CUDA-C shim calls a supplied CUDA-C kernel to link with Scheme code. In this thesis we used constant prefixes to name kernel parameters and supply type information for Gambit.

We discussed the implementation of our work in Gambit in Chapter 4. We provided special constructs in Scheme to define a kernel, call a kernel with an execution configuration, and synchronize multiple kernels. Gambit extracts necessary information to generate both shims from parse tree nodes generated from a kernel skeleton defined in Scheme. Type information for kernel arguments is extracted from the constant prefixes mentioned in their names, so programmers must follow a strict naming convention in our implementation when naming kernel parameters. In Chapter we also discussed some implemented library functions in Scheme for GPUs.

In Chapter 5, we evaluated our implementation by running some test cases. These test cases cover various language constructs of our implementation in Gambit and CUDA-C. We found that some test cases implemented in Scheme created a reasonably low overhead compared to their CUDA-C implementations. We also found that some test cases do not create any overhead. We implement two extra notations - `IN` and `OUT` - to avoid unnecessary memory transfer operations for vectors. We found that, without these two extra notations, some test cases created overhead ranging from more than 100% in Scheme.

Our work enables Scheme programmers to develop expressive programs that control and issue data-parallel programs running on GPUs, while also reducing hands-on memory management.

6.2 Contributions

The specific contributions of this work are:

1. An extended Gambit Scheme compiler with some special constructs for GPU computation
2. A linkage of Scheme programs to CUDA-C kernels that involves necessary type conversions for data types.
3. Parse tree nodes generated by Gambit to extract kernel parameters and names.
4. Useful library functions in Scheme for Gambit to manage GPUs.

6.3 Future Work

Our implementation in Gambit creates some scope for future works. These future works are directly related to the limitations of our work.

6.3.1 Type inference for kernel parameters

In our implementation, we use constant prefixes to extract type information for kernel parameters. Therefore, programmers must follow a strict naming conventions to name kernel parameters in our implementation. Using these constant prefixes might be irritating to Scheme programmers. It would be easy for programmers to name kernel parameters without constant prefixes. However, in order to link Scheme code to CUDA-C kernel type information, kernel parameters are required. This is because Scheme does not provide any type information for data types. However, CUDA-C requires type information for data types, whereas We can extract type information for kernel parameters using *type inference* [37] technique by implementing it in Gambit. Type inference technique allows a compiler to deduce the type for a data type during the time of compilation.

6.3.2 Minimization of unnecessary memory transfer operations

In Chapter 5, we described how our implementation eliminates extra unnecessary memory transfer operations by providing `IN` and `OUT` annotations in kernel parameters' names. Specifically, when a vector is passed to multiple kernels our implementation requires two extra memory transfer operations for each vector in CUDA-C shim. These extra memory transfer operations contribute to overall overhead. Therefore, we need to eliminate these extra memory transfer operations in order to minimize overhead in our implementation.

6.3.3 Compiling body of a kernel

For now, our implementation links Scheme code to CUDA-C kernels. In order to do that, our implementation generates Scheme and CUDA-C shims from a kernel skeleton defined in Scheme. We extend Gambit so that it accepts this special constructs for kernel skeleton and generates both shims. Our implementation extracts the parse tree nodes constructed from the body of a kernel skeleton. Theses parse tree nodes can be used to compile the body of a kernel to CUDA-C code that can run on a GPU. Compilation of a kernel's body also enables us to link device functions from a kernel because device functions are only callable from kernels. Note that kernels in CUDA-C do not support recursion and they cannot declare static variables inside their body.

References

[1] —. *GCC, the GNU compiler Collection*. Free Software Foundation, Inc., November 2012. URL `http://gcc.gnu.org/gcc-4.4/`. Last accessed: July 2013.

[2] —. *IBM system/360 Operating System Linkage Editor Loader IBM Form No C28-6538-8*. IBM Corporation, October 1966. IBM Programming Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602.

[3] —. *IBM System/360 Operating System Assembler Language IBM Form No C28-6514-5*. IBM Corporation, December 1967. IBM Programming Publications, Department 232, San Jose, California 95114.

[4] —. *IBM system/360 Operating System Linkage Editor Program Logic Manual IBM Form No Y28-6667-0*. IBM Corporation, January 1968. IBM Programming Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602.

[5] —. *OpenCL - The open standard for parallel programming of heterogeneous systems*. The Khronos OpenCL Working Group, 2013. URL `http://www.khronos.org/opencl/`. Last accessed: December 2013.

[6] —. *CUDA C Best Practices Guide*. NVIDIA Corporation, July 2013. URL `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide`. Last accessed: June 2013.

[7] —. *CUDA Downloads. Developer Zone*. NVIDIA Corporation, 2013. URL `https://developer.nvidia.com/cuda-downloads`. Last accessed: July 2013.

[8] —. *CUDA GPUs. Developer Zone*. NVIDIA Corporation, 2013. URL `http://developer.nvidia.com/cuda-gpus`. Last accessed: May 2013.

[9] —. *CUDA Toolkit Documentation*. NVIDIA Corporation, July 2013. URL `http://docs.nvidia.com/cuda/cuda-samples/index.html#scalar-product`. Last accessed: August 2013.

[10] —. *GeForce GTX 560 Ti, Specifications*. NVIDIA Corporation, 2013. URL `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/specifications`. Last accessed: July 2013.

[11] —. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide*. NVIDIA Corporation, April 2007.

[12] —. *NVIDIA CUDA Library Documentation 4.2*. NVIDIA Corporation, May 2012. URL `http://graphics.im.ntu.edu.tw/~bossliaw/nvCuda_doxygen/html/index.html`. Last accessed: June 2013.

[13] —. *Quadro 600*. NVIDIA Corporation, 2013. URL `http://www.nvidia.com/object/product-quadro-600-us.html`. Last accessed: July 2013.

[14] —. *What is GPU Computing?* NVIDIA Corporation, 2013. URL `http://www.nvidia.com/object/what-is-gpu-computing.html`. Last accessed: April 2013.

[15] —. *The Java Tutorial. Lesson: Concurrency*. Oracle Corporation, 2013. URL `http://docs.oracle.com/javase/tutorial/essential/concurrency/`. Last accessed: January 2014.

[16] —. *Test the result of CUDA Matrix multiplication using shared memory and global memory.* Stackoverflow, 2013. URL `http://stackoverflow.com/questions/12526062/test-the-result-of-cuda-matrix-multiplication-using-shared-memory-and-global-mem?rq=1`. Last accessed: June 2013.

[17] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[18] Peter Achten, John Van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In *1992 Glasgow Workshop on Functional Programming*, pages 1–17, Ayr, Scotland, July 1993. Springer-Verlag. doi: 10.1007/978-1-4471-3215-8_1.

[19] William A Adam. Unchaining the GPU with Lua and OpenCL. URL `http://williamaadams.wordpress.com/2012/04/23/unchaining-the-gpu-with-lua-and-opencl`. Last Accessed: December 2013.

[20] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007. ISBN 193435600X.

[21] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, September 2010. ACM. ISSN 0001-0782. doi: 10.1145/1810891.1810910.

[22] D.H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R.A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, New York, NY, USA, November 1991. ACM. doi: 10.1145/125826.125925.

[23] D W. Barrons. *Assembler and Loader.* American Elsevier, New York, 1969.

[24] Françoise Baude, Fabrice Belloncle, Denis Caromel, Nathalie Furmento, Yves Roudier, Philippe Mussi, and Günther Siegel. Parallel object-oriented programming for parallel simulations. *Information Sciences*, 93(1–2):35 – 64, August 1996. Elsevier Science Inc. ISSN 0020-0255. doi: 10.1016/0020-0255(96)00060-6.

[25] Edip Baysal, Dan D. Kosloff, and John W.C Sherwood. Reverse time migration. In *Geophysics.* 48(11): 1514–1524. Society of Exploration Geophysicists, November 1983.

[26] OpenMP Architecture Review Board. *The OpenMP API.* OpenMP ARB Corporation, 2013. URL `http://openmp.org/openmp-faq.html#Content.OMPAPI`. Last accessed: January 2014.

[27] Bryan Catanzaro, Bor yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, and Kurt Keutzer. Efficient, high-quality image contour detection. In *In IEEE International Conference on Computer Vision*, pages 2381 – 2388, Kyoto, Japan, October 2009, IEEE.

[28] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, January 2011. ACM. ISBN 978-1-4503-0486-3. doi: 10.1145/1926354.1926358.

[29] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Standford University, March 1992.

[30] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, August 2012. Elsevier Science Publishers B. V. ISSN 0167-8191. doi: 10.1016/j.parco.2011.10.002.

[31] R. Kent Dybvig. *The Scheme Programming Language.* MIT Press, fourth edition, 2009.

[32] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, ICPP '11, pages 216–225, Taipei City, Taiwan, September 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.45.

[33] Marc Feeley. *A Tour of Scheme in Gambit*, January 2009. URL `http://dynamo.iro.umontreal.ca/wiki/index.php/A_Tour_of_Scheme_in_Gambit`. Last accessed: June 2013.

[34] Marc Feeley. *Distributions*, December 2013. URL `http://dynamo.iro.umontreal.ca/wiki/index.php/Distributions`. Last accessed: July 2013.

[35] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *In Lisp and Functional Programming*, pages 119–130, Nice, France, June 1990. ACM Press.

[36] Matthias Felleisen. On the expressive power of programming languages. *Sci. of Computer Programming*, 17(1-3):35–75, December 1991. Elsevier North-Holland, Inc. ISSN 0167-6423. doi: 10.1016/0167-6423(91)90036-W.

[37] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3rd edition, 2008. ISBN 0262062798.

[38] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in Termite Scheme. In *Scheme and Functional Programming Workshop 2006*, pages 125–135, Portland, Oregon, USA, September 2006. ACM Press. doi: 10.1145/1159789.1159795.

[39] Dan F M Goodman and Romain Brette. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2(5):1–10, November 2008. doi: 10.3389/neuro.11.005.2008.

[40] Andrew S. Grimshaw. Object-oriented parallel processing with Mentat. *Inf. Sci.*, 93(1):9–34, August 1996. Elsevier Science Inc. doi: 10.1016/0020-0255(96)00059-X.

[41] Jürg Gutknecht. Oberon, Gadgets, and some archetypal aspects of persistent objects. *Inf. Sci.*, 93(1):65–86, August 1996. Elsevier Science Inc. doi: 10.1016/0020-0255(96)00061-8.

[42] Kevin Hammond. Why parallel functional programming matters: Panel statement. In Alexander Romanovsky and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 201–205. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21337-3. doi: 10.1007/978-3-642-21338-0_17.

[43] Kevin Hammond. Why parallel functional programming matters: Panel statement. In Alexander Romanovsky and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 201–205. Edinburgh, UK, June 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21337-3. doi: 10.1007/978-3-642-21338-0_17.

[44] Mark Harris. Optimizing Parallel Reduction in CUDA. Technical report, NVIDIA Corporation, 2008. URL `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

[45] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989. ACM. ISSN 0360-0300. doi: 10.1145/72551.72554.

[46] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989. Oxford University Press. doi: 10.1093/comjnl/32.2.98.

[47] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.org, PUC-Rio., 2012. URL `http://www.lua.org/manual/5.1/manual.html`. Last accessed: May 2013.

[48] Ravi Prasad K. Jagannath and Phaneendra K. Yalavarthy. Efficient gradient-free simplex method for estimation of optical properties in image-guided diffuse optical tomography. *Journal of Biomedical Optics*, 18(3):030503, March 2013. SPIE. doi: 10.1117/1.JBO.18.3.030503.

[49] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002. URL `http://haskell.org/definition/haskell98-report.pdf`.

[50] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 51–60, New York, NY, USA, October 2006. ACM. ISBN 1-59593-578-9. doi: 10.1145/1178597.1178605.

[51] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, March 2012, Elsevier Science Publishers B. V. ISSN 0167-8191. doi: 10.1016/j.parco.2011.09.001.

[52] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of OpenCL programs, June 2010.

[53] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *ACM Haskell symposium*, Haskell '10, pages 67–78, New York, NY, USA, September 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863533.

[54] Tatsuru Matsushita. Expressive power of declarative programming languages, PhD thesis, Department of Computer Science, University of York. October 1998.

[55] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In David R. Kaeli and Miriam Leeser, editors, *GPGPU*, volume 383 of *ACM International Conference Proceeding Series*, pages 79–84. ACM, March 2009. ISBN 978-1-60558-517-8. doi: 10.1145/1513895.1513905.

[56] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for GPUs in Scala. In Ewen Denney and Ulrik Pagh Schultz, editors, *GPCE*, pages 107–116, New York, NY, USA, October 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047883.

[57] Phil Perillat. *3DFD*. Arecibo Observatory, 2009. URL `http://www.naic.edu/~phil/hardware/nvidia/doc/src/3DFD/`. Last accessed: April 2013.

[58] Guotao Quan, Hui Gong, Yong Deng, Jianwei Fu, and Qingming Luo. Monte Carlo–based fluorescence molecular tomography reconstruction method accelerated by a cluster of graphic processing units. *Journal of Biomedical Optics*, 16(2):026018, February 2011. SPIE - the international society for optics and photonics. doi: 10.1117/1.3544548.

[59] Atanas Radenski. Object-oriented programming and parallelism: Introduction. *Inf. Sci.*, 93(1):1–7, August 1996. Elsevier Science Inc. doi: 10.1016/0020-0255(96)00058-8.

[60] John D. Ramsdell. Scheme: the next generation. *ACM Lisp Pointers*, 7(4):13–14, December 1994.

[61] Paul Roe. Parallel programming using functional languages. PhD thesis, Department of Computing Science, University of Glasgow. February 1991.

[62] Xavier Saint-Mleux, Marc Feeley, and Jean pierre David. Shard: a Scheme to hardware compiler. In *Scheme and Functional Programming Workshop 2006*, pages 39–49, Portland, Oregon, USA, September 2006. ACM Press.

[63] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997. ISBN 0201895390.

[64] Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065–2074, May 2010. Elsevier B.V. ISSN 18770509. doi: 10.1016/j.procs.2010.04.231.

[65] T. Topa, A. Karwowski, and A. Noga. Using GPU With CUDA to Accelerate MoM-Based Electromagnetic Simulation of Wire-Grid Models. *Antennas and Wireless Propagation Letters, IEEE*, 10:342–345, December 2011. ISSN 1536-1225. doi: 10.1109/lawp.2011.2144557.

[66] Mark van Heeswijk, Yoan Miche, Erkki Oja, and Amaury Lendasse. GPU-accelerated and parallelized ELM ensembles for large-scale regression. *Neurocomputing*, 74(16):2430 – 2437, September 2011. Elsevier B.V. ISSN 0925-2312. doi: 10.1016/j.neucom.2010.11.034.

[67] David A. Watt. *Programming language processors - compilers and interpreters.* Prentice Hall International Series in Computer Science. Prentice Hall, 1993. ISBN 978-0-13-720129-7.

[68] Rick Weber, Akila Gothandaraman, Robert J. Hinde, and Gregory D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, January 2011. IEEE Computer Society. ISSN 1045-9219. doi: 10.1109/TPDS.2010.125.

[69] Yasuhiko Yokote and Mario Tokoro. The design and implementation of concurrent Smalltalk. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 331–340, New York, NY, USA, January 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28730.

[70] Bo Zhang, Xiang Yang, Fei Yang, Xin Yang, Chenghu Qin, Dong Han, Xibo Ma, Kai Liu, and Jie Tian. The CUBLAS and CULA based GPU acceleration of adaptive finite element framework for bioluminescence tomography. *Optics Express*, 18(19):20201–20214, Sep 2010. The Optical Society. doi: 10.1364/OE.18.020201.

Appendix A

Vector addition

This appendix shows code for test case parallel vector addition, described in section 5.2. We provide the host program implemented in Scheme.

A.1 Host program in Scheme

In Listing A.1 we provide host program implemented in Scheme calls CUDA-C kernel `VecAdd` on line 48. The kernel skeleton is defined on lines 2–3.

```
1  ;;--------------------kernel skeleton----------------------
2  (define VecAdd
3    (kernel (f32v_IN_D_A f32v_D_B u32_constant u32_N)))
4  ;;-----------------------------------------------------------
5  (c-declare #<<c-declare-end
6    void RandomInit(float*, int);
7    void checkResult(float*, float*, int, int);
8  c-declare-end
9  )
10
11 (define check-result
12   (c-lambda (scheme-object scheme-object int int)
13             void
14 #<<c-lambda-end
15     ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
16     ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));
17     checkResult(host_vec1, host_vec2, ___arg3, ___arg4);
18 c-lambda-end
19 ))
20
21 (define random-init
22     (c-lambda (scheme-object  int)
23               void
24 #<<c-lambda-end
25          ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
26          RandomInit(host_vec1,  ___arg2);
27 c-lambda-end
28 ))
29
30 (define args (command-line))
31 (define N (string->number   (car (cdr args))))
32 (define check-result? (string->number (car  (cdr  (cdr args)))))
33 ;;for fixed grid size
34 (define constant 786)
35 (define grid-size 65535)
36 (define block-size (+ 1 (floor (/ N grid-size))))
37 ;;for fixed block size
38 ;(define block-size 1024)
39 ;(define grid-size (floor (/ (- (+ N block-size) 1) block-size)))
40
41 (define D_A (##still-copy(make-f32vector N)))
42 (define D_B (##still-copy(make-f32vector N)))
43
44 (define (main)
```

```
45    (random-init D_A N)
46    (random-init D_B N)
47    (gpu-time
48      (VecAdd <<< (grid-size) (block-size) >>> D_A D_B constant N))
49    (if (= check-result? 1)
50       (check-result D_A D_B constant N)))
51
52  (main)
```

Listing A.1: Host program implemented in Scheme calling CUDA-C kernel `VecAdd`

Appendix B

Synchronized kernels

In this appendix we provide the Scheme implementation for the test case multiple synchronized kernels, described in Section 5.3. In this test case host program calls two synchronized kernels.

B.1 Host program in Scheme

In Listing B.1 we provide host program implemented in Scheme calls two synchronized kernels `VecAdd` and `VecSub` on lines 54–55. Two kernel skeletons are defined on lines 2–6.

```
1  ;;------------------kernel skeleton---------------------------------
2  (define VecAdd
3    (kernel (u32v_IN_A u32v_OUT_B u32_constant u32_N)))
4
5  (define VecSub
6    (kernel (u32v_IN_B u32v_OUT_C u32_constant u32_N)))
7  ;;-----------------------------------------------------------------
8  (c-declare #<<c-declare-end
9    #include <stdint.h>
10   void RandomInit(uint32_t*, int);
11   void checkResult(uint32_t*, uint32_t*, int, int);
12 c-declare-end
13 )
14
15 (define check-result
16   (c-lambda (scheme-object scheme-object int int)
17             void
18 #<<c-lambda-end
19     ___U32* host_vec1 = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
20     ___U32* host_vec2 = ___CAST(___U32*, ___BODY_AS(___arg2, ___tSUBTYPED));
21     checkResult(host_vec1, host_vec2, ___arg3, ___arg4);
22 c-lambda-end
23 ))
24
25 (define random-init
26   (c-lambda (scheme-object  int)
27               void
28 #<<c-lambda-end
29          ___U32* host_vec1 = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
30          RandomInit(host_vec1,  ___arg2);
31 c-lambda-end
32 ))
33
34 (define args (command-line))
35 (define N (string->number   (car (cdr args))))
36 (define check-result? (string->number (car  (cdr  (cdr args)))))
37
38 (define constant 786)
39 ;;for fixed grid size
40 (define grid-size 65535)
41 (define block-size (+ 1 (floor (/ N grid-size))))
42 ;;for fixed block size
43 ;(define block-size 1024)
44 ;(define grid-size (floor (/ (- (+ N block-size) 1) block-size)))
```

```
45
46 (define D_A (##still-copy(make-u32vector N)))
47 (define D_B (##still-copy(make-u32vector N)))
48 (define D_C (##still-copy(make-u32vector N)))
49
50 (define (main)
51   (random-init D_A N)
52   (gpu-time
53     (sync
54       (VecAdd <<< (grid-size) (block-size) >>> D_A D_B constant N)
55       (VecSub <<< (grid-size) (block-size) >>> D_B D_C constant N)))
56   (if (= check-result? 1)
57      (check-result D_A D_C constant N)))
58
59 (main)
```

Listing B.1: Host program in Scheme calls two synchronized kernels `VecAdd` and `VecSub`

Appendix C

Reduction

This appendix shows host program implemented in Scheme for the test case parallel sum reduction described in Section 5.3.

C.1 Host program in Scheme

In Listing C.1 we provide the host program implemented in Scheme calls two synchronized kernels performing sum reduction. Here, two kernel skeletons are defined on lines 2–6. Kernel `reduce5` is called on lines 57–60 and kernel `reduce6` is called on lines 62–66. In this test case `sync` macro takes the function calls to `reduce` as arguments on lines 76–81 to synchronized executions of `reduce6` and `reduce5`.

```
1  ;;------------------kernel skeleton------------------------------
2  (define reduce6
3    (kernel (u32v_idata u32v_odata u32_size u32_threads)))
4
5  (define reduce5
6    (kernel (u32v_INidata u32v_OUTodata u32_threads)))
7  ;;---------------------------------------------------------------
8  (c-declare #<<c-declare-end
9    void random_data();
10   int CPU_result();
11 c-declare-end
12 )
13
14 (define random-data
15   (c-lambda (scheme-object scheme-object int int)
16             void
17 #<<c-lambda-end
18     ___U32* host_vec1 = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
19     ___U32* host_vec2 = ___CAST(___U32*, ___BODY_AS(___arg2, ___tSUBTYPED));
20     random_data(host_vec1, host_vec2, ___arg3, ___arg4);
21 c-lambda-end
22 ))
23
24 (define CPU-result
25   (c-lambda (scheme-object int)
26             int
27 #<<c-lambda-end
28     ___U32* host_vec1 = ___CAST(___U32*, ___BODY_AS(___arg1, ___tSUBTYPED));
29     ___result = CPU_result(host_vec1,  ___arg2);
30 c-lambda-end
31 ))
32
33 (define args (command-line))
34 (define SCALING (string->number   (car (cdr args))))
35 (define check-result? (string->number (car  (cdr  (cdr args)))))
36 (define maxBlocks (string->number (car  (cdr (cdr  (cdr args))))))
37
38 (define int-size 4)
39 (define size (arithmetic-shift 1 SCALING))
40 (define maxThreads 512)
41
```

```
(define (block-size n  maxThreads)
  (if (= n 1)
      1
      (if (< n (* maxThreads 2))
          (/ n 2)
          maxThreads)))

(define (num-blocks n whichKernel threads maxBlocks)
  (let ((blocks (/ n (* threads 2))))
    (if (= whichKernel 6)
        (min maxBlocks blocks)
        blocks)))

(define (reduce size threads blocks whichKernel d_idata d_odata)
  (cond ((= whichKernel 5)
         (reduce5 <<<(blocks)(threads)(* threads int-size)>>>
                  d_idata
                  d_odata
                  threads))
        ((= whichKernel 6)
         (reduce6 <<<(blocks)(threads)(* threads int-size)>>>
                  d_idata
                  d_odata
                  size
                  threads))))

(let* ((numThreads  (block-size size maxThreads))
       (numBlocks   (num-blocks size 6 numThreads maxBlocks))
       (idata       (make-u32vector size))
       (odata       (make-u32vector numBlocks)))
   (begin
     (random-data idata odata size numBlocks)
     (let ((cpu-result (CPU-result idata size)))
       (gpu-time
         (sync
           (reduce size numThreads numBlocks 6 idata odata)
           (reduce size numThreads numBlocks 6 idata odata)
           (let* ((threads  (block-size numBlocks maxThreads))
                  (blocks (num-blocks numBlocks 5 threads maxBlocks)))
             (reduce numBlocks threads 1  5 odata odata))))
      (if (= check-result? 1)
          (if (= cpu-result (u32vector-ref odata 0))
              (display "TestPassed")
              (display "TestFailed")))))))
```

Listing C.1: Host program in Scheme calls two synchronized kernels `reduce6` and `reduce5`.

Appendix D

Matrix Multiplication

This appendix shows code for test case parallel matrix multiplication, described in Section 5.4. We provide the host program implemented in Scheme. The CUDA-C implementation for this text case is available at `http://stackove rflow.com/questions/12526062/test-the-result-of-cuda-matrix-multiplication-using-shared-memory-and-global-mem?rq=1`.

D.1 Host program in Scheme

In Listing D.1 we provide the host program in Scheme that calls CUDA-C kernel `kernel_global` that performs parallel matrix multiplication on two squares matrices and returns results in another square matrix. The kernel skeleton is defined on lines 2–4 and the kernel is called on lines 36–41.

```
1  ;;------------------------------kernel skeleton----------------------------------
2  (define kernel_global
3    (kernel(f32v_INinput1 f32v_INinput2 f32v_OUToutput
4            u32_width u32_divide u32_tileWidth)))
5
6  ;;-------------------------------------------------------------------------------
7
8  ;;------------------------------helper-functions---------------------------------
9  (define (matrix-size)
10   (* matrix-width matrix-width))
11
12 (define (get-gridDim)
13   (/ (/ matrix-width matrix-divide) tile-width))
14 ;;-------------------------------------------------------------------------------
15
16 (define (test-result output)
17   (let loop ((counter 0))
18     (if (< counter (matrix-size))
19         (if ( = (f32vector-ref  output counter) matrix-width)
20             (begin
21               ;;(display (f32vector-ref  output counter))
22               (loop (+ counter 1)))
23             (display "Test Failed"))
24         (display "Test Passed"))))
25
26 (define matrix-width 9216)
27 (define tile-width 32)
28 (define matrix-divide 4)
29
30 (let ((input1 (make-f32vector (matrix-size) 1.00))
31       (input2 (make-f32vector (matrix-size) 1.00))
32       (output (make-f32vector (matrix-size) 0.0)))
33   (display "calling-kernel")(newline)
34   (gpu-time
35                     ;;   gridDim-x   gridDim-y     blockDim-x blockDim-y
36     (kernel_global <<<((get-gridDim)(get-gridDim))(tile-width tile-width) >>>
37                     input1
38                     input2
39                     output
40                     matrix-width
```

```
41                      matrix-divide tile-width))
42   (display "testing results")(newline)
43   (test-result output))
```

Listing D.1: Host program in Scheme that calls CUDA-C kernel `kernel_global` that performs parallel matrix multiplication

Appendix E

3DFD

This appendix shows code for test case parallel 3D finite difference computation, discussed in Section 5.6. We provide the host program implemented in Scheme. CUDA-C implementation for this test case is available at `http://www.naic.edu/~phil/hardware/nvidia/doc/src/3DFD`.

E.1 Host program in Scheme

In Listing E.1 we provide the host program in Scheme that calls CUDA-C kernel `stencil_3D_16x16_order8`. We also provide the wrapper functions that links C functions for CPU computation on lines 35–90. The kernel skeleton is defined on lines 2–3 and the kernel is called on lines 98–104.

```
1  ;;-----------kernel skeleton-----------------------------------
2  (define stencil_3D_16x16_order8
3    (kernel (f32v_OUToutput  f32v_ININput u32_dimx u32_dimy u32_dimz)))
4
5  ;;-------------------------------------------------------------
6
7  (define BLOCK_DIMX 16)
8  (define BLOCK_DIMY 16)
9  (define RADIOUS 4)
10 (define dimx 480)
11 (define dimy 480)
12 (define dimz 400)
13 (define nreps 1)
14 (define float-size 4)
15
16 (define len (* dimz  (* dimx dimy)))
17 (define d_input (create-still-f32vector (* float-size len)))
18 (define d_output (create-still-f32vector (* float-size len)))
19 (define h_data (create-still-f32vector (* float-size len )))
20 (define h_reference (create-still-f32vector (* float-size len)))
21
22 ;;----------------helper function------------------------------
23 (define create-still-f32vector
24    (c-lambda (int) scheme-object
25 #<<c-lambda-end
26    ___result = ___alloc_scmobj(___sF32VECTOR, ___arg1, ___PERM);
27    ___EXT(___release_scmobj) (___result);
28 c-lambda-end
29 ))
30 ;;-------------------------------------------------------------
31
32 ;;-------------------------------wrapper-functions--------------
33 ;; interface to C-functions for CPU computation
34
35 (c-declare #<<c-declare-end
36 #include <stdlib.h>
37 #include <stdio.h>
38 #include <stdbool.h>
39
40 //forward-declarations for CPU functions
41
```

```
void random_data(float* h_data1, float* h_data2,
                 int dimx, int dimy, int dimz, int one, int five);
void reference_3D(float* h_reference, float* h_data,
                  int dimx, int dimy, int dimz, int radious );
bool within_epsilon(float* h_data,float* h_reference,
                    int dimx, int dimy, int dimz, int zadjust, float delta);

c-declare-end
)

(define random-data
  (c-lambda (scheme-object scheme-object int int int int int)
            void
#<<c-lambda-end
        ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
        ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));


        random_data(host_vec1, host_vec2, ___arg3,
                    ___arg4, ___arg5, ___arg6, ___arg7);

c-lambda-end
))

(define reference-3D
  (c-lambda (scheme-object scheme-object int int int int)
            void
#<<c-lambda-end
        ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
        ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));

        reference_3D(host_vec1, host_vec2, ___arg3,
                     ___arg4, ___arg5, ___arg6);

c-lambda-end
))

(define within-epsilon
  (c-lambda (scheme-object scheme-object int int int int float32)
            bool
#<<c-lambda-end
        ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
        ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));

        ___result = within_epsilon(host_vec1, host_vec2, ___arg3,
                                   ___arg4, ___arg5, ___arg6, ___arg7);

c-lambda-end
))
;;-------------------------------------------------------------------------------

(define (main)
  (begin
    (random-data h_data d_input dimx dimy dimz 1 5)
    (display "\n calling kernel for device-computation \n")
    (gpu-time
      (stencil_3D_16x16_order8<<<(/ dimx BLOCK_DIMX)(/ dimy BLOCK_DIMY))
                                   (16 16) >>>
        d_output
```

```
        d_input
        dimx
        dimy
        dimz))
    (display "\n calling  functions for CPU-computation \n")
    (reference-3D h_reference h_data  dimx dimy dimz  RADIOUS)
    (display "\n comparing results \n")
    (if (within-epsilon
          d_output
          h_reference
          dimx
          dimy
          dimz
          (* RADIOUS nreps)
          0.000100)
        (begin
          (display "\n  Result within epsilon\n")
          (display "\n TEST PASSED \n"))
          (display "\n TEST FAILED \n")))))

(main)
```

Listing E.1: Host program in Scheme calling CUDA-C kernel `stencil_3D_16x16_order8` that performs parallel 3DFD

Appendix F

Scalar Product

This appendix shows code for test case parallel scalar product discussed in Section 5.6. We provide the host program implemented in Scheme. The CUDA-C kernel is taken from NVIDIA's CUDA SDK which is available at `http://www.naic.edu/~phil/hardware/nvidia/doc/src/scalarProd/`.

F.1 Host program in Scheme

In Listing F.1 we provide the host program implemented in Scheme that calls CUDA-C kernel `scalarProdGPU`. Here, kernel skeleton for CUDA-C kernel `scalarProdGPU` is defined on lines 2–3. We provide some `c-lambda` functions on lines 28–57 to link some helper functions implemented in C. The kernel is called on lines 66–71.

```
1  ;;------------------kernel skeleton ----------------------------------
2  (define scalarProdGPU
3    (kernel (f32v_OUTh_C_GPU  f32v_INh_A f32v_INh_B u32_vectorN u32_elementN)))
4  ;;-----------------------------------------------------------------------
5  (define args (command-line))
6  (define SCALING (string->number   (car (cdr args))))
7  (define check-result? (string->number (car  (cdr  (cdr args)))))
8  (define GRID-SIZE (string->number (car  (cdr (cdr  (cdr args))))))
9
10 (define BLOCK-SIZE 128)
11 (define VECTOR_N (* 256 SCALING))
12 (define ELEMENT_N  4096)
13 (define DATA_N (* VECTOR_N ELEMENT_N))
14 (define float-size 4)
15
16 (define h_A (##still-copy(make-f32vector  DATA_N)))
17 (define h_B (##still-copy (make-f32vector  DATA_N)))
18 (define h_C_GPU (##still-copy (make-f32vector  VECTOR_N)))
19 (define h_C_CPU (##still-copy (make-f32vector VECTOR_N)))
20
21 (c-declare #<<c-declare-end
22   void random_data(float*, float*, int);
23   void scalarProdCPU(float*, float*, float*, int, int);
24   void compareResult(float*, float*, int);
25 c-declare-end
26 )
27 ;;---------------interface to C code----------------------------------------
28 (define random-data
29   (c-lambda (scheme-object scheme-object int)
30             void
31 #<<c-lambda-end
32     ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
33     ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));
34     random_data(host_vec1, host_vec2, ___arg3);
35 c-lambda-end
36 ))
37
38 (define scalarProd-CPU
39   (c-lambda (scheme-object scheme-object scheme-object int int)
40             void
41 #<<c-lambda-end
```

```
42      ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
43      ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));
44      ___F32* host_vec3 = ___CAST(___F32*, ___BODY_AS(___arg3, ___tSUBTYPED));
45      scalarProdCPU(host_vec1, host_vec2, host_vec3, ___arg4, ___arg5);
46  c-lambda-end
47  ))
48
49  (define compare-result
50    (c-lambda (scheme-object scheme-object int)
51              void
52  #<<c-lambda-end
53      ___F32* host_vec1 = ___CAST(___F32*, ___BODY_AS(___arg1, ___tSUBTYPED));
54      ___F32* host_vec2 = ___CAST(___F32*, ___BODY_AS(___arg2, ___tSUBTYPED));
55      compareResult(host_vec1, host_vec2, ___arg3);
56  c-lambda-end
57  ))
58  ;;-----------------------------------------------------------------------------
59  (define (host)
60    (display "\ncalling for random data...\n")
61    (random-data h_A h_B DATA_N)
62    (display "calling for CPU result...\n")
63    (scalarProd-CPU h_C_CPU h_A h_B VECTOR_N ELEMENT_N)
64    (display "Calling kernel...\n")
65    (gpu-time
66      (scalarProdGPU <<<(GRID-SIZE)(BLOCK-SIZE)>>>
67        h_C_GPU
68        h_A
69        h_B
70        VECTOR_N
71        ELEMENT_N))
72    (display "Compare results...\n")
73    (if (= check-result? 1)
74        (compare-result h_C_GPU h_C_CPU VECTOR_N)))
75  (host)
```

Listing F.1: Host program in Scheme calling CUDA-C kernel `scalarProdGPU` performing parallel scalar product on a GPU

Appendix G

Shims with time stamps code

In this appendix, we present a generated CUDA-C shim and a Scheme shim with the generated time stamps in CUDA-C by Gambit. These generated time stamps measure execution time for a supplied CUDA-C kernel only and the combined execution time for a CUDA-C shim and a supplied kernel. We provide the command-line option `-bare-time` to generate these time stamps in both shims. We also present some code snippets from our implementation in Gambit that inject time stamps CUDA-C code in both shims.

G.1 CUDA-C shim generated with time stamps

In Listing F.1, we provide a generated CUDA-C shim with the time stamps to measure execution time only for a CUDA-C kernel. Here, lines 32–37 and 42–47 are generated to measure execution time for a CUDA-C kernel only which is called on lines 39–40.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <cuda.h>
4  #include <math.h>
5  #include <stdint.h>
6  #include "cuda.h"
7  #include "cudalib.cu"
8  #include "cuda_runtime_api.h"
9
10
11 __global__  void vector_addition (  uint32_t u32_constant ,  uint32_t* u32v_src ,
12                                     int u32v_src_len )  ;
13
14
15
16
17 extern "C" {
18
19 void vector_addition_cu_driver (  int gDx, int gDy, int gDz, int bDx, int bDy,
20                                   int bDz, int shared_size, uint32_t u32_constant,
21                                   uint32_t* h_u32v_src, int h_u32v_src_len )  {
22         uint32_t* d_u32v_src;
23         size_t size_u32v_src = h_u32v_src_len * sizeof(uint32_t);
24         cudaMalloc((void **) &d_u32v_src, size_u32v_src);
25         cudaMemcpy(d_u32v_src, h_u32v_src, size_u32v_src, cudaMemcpyHostToDevice);
26
27         dim3 dimGrid(gDx, gDy, gDz);
28         dim3 dimBlock(bDx, bDy, bDz);
29
30         size_t size = shared_size;
31
32         cudaEvent_t start, stop;
33         float elapsed_time_ms = 0.0f;
34         cudaEventCreate( &start );
35         cudaEventCreate( &stop);
36         //taking start time-stamp
37         cudaEventRecord( start, 0);
38         //calling kernel
39         vector_addition <<< dimGrid, dimBlock, size >>> (u32_constant, d_u32v_src,
```

```
                                                        h_u32v_src_len );
        //taking start time-stamp
        cudaEventRecord( stop, 0);
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &elapsed_time_ms, start, stop );
        printf(" %f ",elapsed_time_ms);
        cudaEventDestroy(start);
        cudaEventDestroy(stop);
        cudaMemcpy(h_u32v_src, d_u32v_src, size_u32v_src, cudaMemcpyDeviceToHost);
        cudaFree(d_u32v_src);
}
}
```

Listing G.1: A CUDA-C shim with the generated time stamps code to measure execution time for a supplied CUDA-C kernel only

G.2 Scheme shim generated with time stamps

In Listing F.2, we provide a generated Scheme shim with time stamps to measure combined execution time for a CUDA-C shim and a supplied CUDA-C kernel. Here, lines 12–17 and 22–27 are generated to measure a combined execution time. On n lines 19–20, the CUDA-C shim is called and the supplied CUDA-C kernel is called on 39–40 of Listing F.1.

```
1  (c-declare #<<c-declare-end
2
3  void vector_addition_cu_driver();
4  c-declare-end
5  )
6
7  (define vector_addition_scm_driver
8  (c-lambda ( int int int int int int unsigned-int32 scheme-object int)
9  void
10 #<<c-lambda-end
11 ___U32* host_u32v_src = ___CAST(___U32*, ___BODY_AS(___arg9, ___tSUBTYPED));
12 cudaEvent_t start, stop;
13     float elapsed_time_ms = 0.0f;
14     cudaEventCreate( &start );
15     cudaEventCreate( &stop);
16     //taking start time-stamp
17     cudaEventRecord( start, 0);
18     //calling CUDA-C shim
19     vector_addition_cu_driver( ___arg1,___arg2,___arg3,___arg4,___arg5,___arg6,
20                                ___arg7,___arg8, host_u32v_src, ___arg10);
21     //taking stop time-stamp
22     cudaEventRecord( stop, 0);
23     cudaEventSynchronize( stop );
24     cudaEventElapsedTime( &elapsed_time_ms, start, stop );
25     printf(" %f ",elapsed_time_ms);
26     cudaEventDestroy(start);
27     cudaEventDestroy(stop);
28
29 c-lambda-end
30 ))
31
32 (define (vector_addition gDx gDy gDz bDx bDy bDz shared-size u32_constant u32v_src)
33 (vector_addition_scm_driver
34  gDx gDy gDz bDx bDy bDz shared-size
35 u32_constant
```

```
u32v_src
(u32vector-length u32v_src))
)
```

Listing G.2: Generated Scheme shim with time stamps to measure combined execution time for a generated CUDA-C shim and a supplied CUDA-C kernel

Note that commenting and indentation in Listings F.1 and F.2 are not auto generated. We annotate the generated CUDA-C shim and Scheme shim to provide more clarity to readers.

G.3 Implementation in Gambit

Here, we present code snippets from our implementation in Gambit that generates two time stamps CUDA-C code before and after a call to a CUDA-C shim or a supplied CUDA-C kernel. In Listing F.3 two Scheme functions generates time stamps in CUDA-C. Here, `dump-event-bf` on lines 1348–1356 generates time stamps in CUDA-C before a call and `dump-event-af` generates time stamps in CUDA-C after a call.

```
(define (dump-event-bf file-port)
 (display
   "cudaEvent_t start, stop;
    float elapsed_time_ms = 0.0f;
    cudaEventCreate( &start );
    cudaEventCreate( &stop);
    cudaEventRecord( start, 0);\n\n"
   file-port)
 (force-output file-port))


(define (dump-event-af file-port)
  (display
    "cudaEventRecord( stop, 0);
     cudaEventSynchronize( stop );
     cudaEventElapsedTime( &elapsed_time_ms, start, stop );
     printf(\" %f \",elapsed_time_ms);
     cudaEventDestroy(start);
     cudaEventDestroy(stop);\n\n"
    file-port)
  (force-output file-port))
```

Listing G.3: Code snippet in Gambit that generates time stamps in CUDA

In Listing F.4 on lines 1795–1796 calls `dump-event-bf` to generate time stamp in CUDA-C before generating a call to a CUDA-C kernel. Lines 1808–1809 calls `dump-event-af` after generating a call to a CUDA-C kernel in a CUDA-C shim.

```
(define (gen-kernel-call parms c-type-table name)
      (display "dim3 dimGrid(gDx, gDy, gDz);\n" cu-file-port)
      (display "dim3 dimBlock(bDx, bDy, bDz);\n\n" cu-file-port)
      (display "size_t size = shared_size;\n\n " cu-file-port)
      (cond ((memq 'bare-time  opts)
             (dump-event-bf cu-file-port)))
      (display name cu-file-port)
      (generate " <<< ")
      (display "dimGrid" cu-file-port)
      (generate ", ")
      (display "dimBlock" cu-file-port)
      (generate ", ")
      (display "size" cu-file-port)
      (generate " >>> ")
      (generate "( ")
```

```
      (gen-kernel-call-arg-list parms c-type-table)
      (generate " );\n\n")
      (cond ((memq 'bare-time  opts)
             (dump-event-af cu-file-port)))
    )
```

Listing G.4: Code snippet in Gambit that generates time stamps to measure execution time only for a CUDA-C kernel

Code in Listing F.5 on lines 2017–2018 calls to `dump-event-bf` to generate a time stamp before generating a call to a CUDA-C shim and lines 2020–2021 calls `dump-event-af` to generate after generating a call to a CUDA-C shim in a Scheme shim to measure a combined execution time for a CUDA-C shim and a supplied CUDA-C kernel.

```
      (dump-code <<-sym)
      (dump-code c-lambda-end-sym)
      (dump-code "\n")
      (if (not (eq? parms '()))
          (gen-c-ptr parms c-type-table))
      (cond ((memq 'bare-time  opts)
             (dump-event-bf scm-file-port)))
      (gen-cu-driver-call kernel-name parms c-type-table)
      (cond ((memq 'bare-time  opts)
             (dump-event-af scm-file-port)))
      (dump-code c-lambda-end-sym)
      (dump-code "\n))\n\n")
```

Listing G.5: Code snippet in Gambit that generates time stamps to measure a combined execution time for a CUDA-C shim and a CUDA-C kernel

Note that code snippets in Listings F.3, F.4 and F.5 are in file `_ptree1.scm` under `gsc` directory of our extended Gambit development source `4.6.2`.

Appendix H

Code snippets from Gambit

Here, we present some code snippets from our implementation in Gambit. Code snippets in Listings G.1, G.2, G.3, G.4 and G.5 are in file `_ptree1.scm` under `gsc` directory of our extended Gambit development source `4.6.2`.

Code snippet in Listing G.1 recognizes a `kernel/device` symbol.

```
(define (gpu-kernel-construct? ptree)
  (if (def? ptree)
      (prc-tag?  ptree)
      #f))



(define (prc-tag? t1)
  (let ((t2 (car (vector-ref t1 2))))
    (if (prc? t2)
        (kernel-or-device-symbol? (vector-ref t2 6))
        #f)))



(define (kernel-or-device-symbol? t2)
  (let ((symbol (vector-ref (car (vector-ref t2 1)) 1)))
    (if (or (eq? kernel-sym symbol)
            (eq? device-sym symbol))
        symbol
        #f)))
```

Listing H.1: Code snippet in Gambit that checks kernel/device function

This code snippet in Listing G.2 extracts name of a kernel and parameters from a parse tree node.

```
(define (get-name prc-tag)
    (vector-ref prc-tag 8))

(define (get-parms prc-tag)
  (let* ((var-tag (vector-ref prc-tag 10))
         (n-parms (length var-tag)))
    (let loop ((var-tag var-tag)
               (parms '()))
      (if (eq? (cdr var-tag) '())
          (append parms(list(symbol->string (vector-ref (car var-tag) 1))))
          (loop
            (cdr var-tag)
            (append parms(list(symbol->string (vector-ref (car var-tag) 1)))))))))
```

Listing H.2: Code snippet in Gambit that extracts a kernel/device function name and parameters

This code snippet in Listing G.3 recognizes a kernel call special construct and converts it into an ordinary Scheme function call.

```
(define gd-max-dim 3)
(define bk-max-dim 3)


(define (kernel-call-exp? args)
  (if (list? args)
      (if (<= 5 (length args))
          (let ((arg1-pt (car args))
                (arg2-pt (car (cdr args)))
                (arg3-pt (car (cdr (cdr args))))
                (arg4-pt (car (cdr (cdr (cdr args))))))
             (if (and (check-exe-config-sym <<<-sym arg1-pt)
                      (check-exe-config    arg2-pt  gd-max-dim)
                      (check-exe-config    arg3-pt  bk-max-dim)
                      (if (check-exe-config-sym >>>-sym arg4-pt)
                          #t
                          (check-exe-config-sym >>>-sym (get-arg5-pt args))))
                 #t
                 #f))
          #f)
      #f))

(define (show-then-return obj)
  (display "showing the object")
  (display obj)
  obj)

(define (get-arg5-pt args)
  (car (cdr (cdr (cdr (cdr args))))))


(define (get-new-arg-list  args source-template env >>>-sym-pos)
  (let ((gd-pt          (car (cdr args)))
        (bk-pt          (car (cdr (cdr args))))
        (rest-args-pt   (get-rest-args-pt args source-template env >>>-sym-pos)))
    (make-arg-list
      (exe-config->arglist  gd-pt bk-pt source-template env)
      rest-args-pt)))

;;;just show and return the pt
;(define (show lst)
;  (search-ptree lst 0)
;  lst)


(define (get-rest-args-pt args source-template env >>>-sym-pos)
  (cond ((= >>>-sym-pos 4)
         (append
           (get-fake-shared-mem-pt source-template env)
           (get-only-kernel-args-pt args)))
        ((= >>>-sym-pos 5)
         (append
           (get-shared-mem-size-pt args)
           (get-kernel-args-pt args)))))

(define fake-shared-mem-size 0)
```

```
(define fake-shared-mem-pos-src-file 262147)

(define (get-fake-shared-mem-pt source-template env)
  (let ((fake-source (gen-fake-source
                      source-template
                      fake-shared-mem-size
                      fake-shared-mem-pos-src-file)))
    (list (pt fake-source env 'true))))


(define (get-only-kernel-args-pt args)
  (cdr (cdr (cdr (cdr args)))))

(define (get-kernel-args-pt args)
  (cdr (cdr (cdr (cdr (cdr args))))))

(define (get-shared-mem-size-pt args)
  (list (car (cdr (cdr (cdr args))))))

(define (make-arg-list exe-config-args-pt  kernel-args-pt)
  (change-next-node-stamp* kernel-args-pt)
  (append exe-config-args-pt kernel-args-pt))

(define kernel-args-init-next-node-stamp 8)

(define (change-next-node-stamp*  pt)
  (let loop ((pt pt)
             (i kernel-args-init-next-node-stamp))
    (node-stamp-set! (car pt) i)
    (if (not (eq? (cdr pt) '()))
        (loop
          (cdr pt)
          (+ i 1)))))


(define (check-exe-config-sym symbol arg-pt)
  (if (ref? arg-pt)
      (let ((tag  (vector-ref arg-pt 8)))
         (if (var? tag)
             (if (eq? symbol (vector-ref  tag 1))
                 #t
                 #f)))
      #f))


(define (check-exe-config pt max-dim)
  (if (app? pt)
      (let ((dim-pt (vector-ref pt 2)))
        (if (list? dim-pt)
            (if (<= (length dim-pt) max-dim)
                #t
                #f)))
      #f))

(define (exe-config->arglist  gd-pt bk-pt kernel-name-source env)
  (let* ((arg-list1 (config->arglist
                       (append-fake-arg kernel-name-source (vector-ref gd-pt 2) env)
                       1))
         (arg-list2 (config->arglist
```

```
                        (append-fake-arg kernel-name-source (vector-ref bk-pt 2) env)
                        4)))
    ;(display "\nthis is exe-config->arglist\n")
    (append (reverse arg-list1) (reverse arg-list2))))

(define (append-fake-arg kernel-name-source lst env)
  (if (< (length lst) 3)
      (append
        lst
        (get-fake-pt
          (gen-fake-source kernel-name-source 1 262147)
          (length lst)
          env))
      lst))


(define (get-fake-pt fake-src  lst-len env)
  (let ((ptree (list (pt fake-src env 'true))))
    (cond ((= lst-len 1) (append ptree ptree))
          ((= lst-len 2) ptree))))

(define (gen-fake-source kernel-name-source value pos-in-src)
  (vector-set! kernel-name-source 1 value)
  (vector-set! kernel-name-source 3 pos-in-src))

(define (config->arglist child-nodes init-stamp)
  (let loop ((nodes child-nodes)
             (modified-nodes '())
             (i init-stamp))
    (let ((node (car nodes)))
      (remove-parent node)
      (node-stamp-set! node i)
      (if (not (eq? (cdr nodes) '()))
          (loop
            (cdr nodes)
            (cons node modified-nodes)
            (+ 1 i))
          (cons node modified-nodes)))))

(define (remove-parent node)
  (vector-set! node 1 #f))
```

Listing H.3: Code snippet in Gambit that recognizes a kernel call expression and converts that kernel call into an ordinary function call

Code snippet in Listing G.4 calls code generation routines after recognizing a kernel/device function. It also initiates output file ports to generate both shims.

```
((**define-expr? source env)
 (let* ((var-source (definition-name source env))
        (var (source-code var-source))
        (v (env-lookup-var env var var-source)))
   (if *ptree-port*
       (begin
         (display "  " *ptree-port*)
         (write (var-name v) *ptree-port*)
         (newline *ptree-port*)))
```

```
  (let ((node (pt (definition-value source) env 'true)))
       (set-prc-names! (list v) (list node))
       (parse-prog
         (cdr program)
          env
          (let ((temp (new-def source env v node)))
            (if cu-file-port
               (let ((symbol-found? (gpu-kernel-construct? temp)))
                 (if symbol-found?
                     (begin
                       (let ((fun-name (get-kernel-name temp)))
                         (if (eq? symbol-found? kernel-sym)
                             (set!
                               gpu-interface-scm-file-port
                               (open-output-file (##string-append
                                                   (##path-strip-extension output)
                                                   "_gpu-interface_"
                                                   fun-name
                                                   gpu-interface-scm-file-extension))))
                         (generate-gpu-code
                           cu-file-port
                           gpu-interface-scm-file-port
                           temp
                           symbol-found?
                           opts)
                         (if (eq? symbol-found? kernel-sym)
                             (let*((scm-file  (string-append
                                               (##path-strip-directory
                                               (##path-strip-extension output))
                                               "_gpu-interface_"
                                               fun-name
                                               gpu-interface-scm-file-extension))
                                   (expr (generate-include-source
                                           (vector-ref source 0)
                                           (vector-ref source 2)
                                            scm-file)))
                               (if (**include-expr? expr)
                                   (begin
                                     (set! gpu-interface-source(include-expr->source expr *ptree-port*))
                                     (set! gpu-interface-source? #t)))
                               (if (not (##memq 'keep-scm-shim  opts))
                                   (delete-file scm-file)))))
                         lst)
                       (cons temp lst)))
               (cons temp lst)))
         proc))))
```

Listing H.4: Code snippet in Gambit that initiates code generation after recognizing a kernel

Code snippet in Listing G.5 generates CUDA-C functions for the implemented library functions in Scheme for GPUs.

```
(define (dump-cuda-library-function cudalib-file-port)
  (display
     " extern \"C\" {
        #include <stdio.h>
        void  devCheckAndSet(int gpudevice)
       {
           int device_count=0;
           int device;


           cudaGetDeviceCount( &device_count);
           if (gpudevice > device_count)
           {
               printf(\"gpudevice >=  device_count ... exiting\\n\");
               exit(1);
           }
           cudaError_t cudareturn;
           cudaDeviceProp deviceProp;


           cudaGetDeviceProperties(&deviceProp,  gpudevice);
```

```
1090            printf(\"[deviceProp.major.deviceProp.minor] = [%d.%d]\\n\",
1091               deviceProp.major, deviceProp.minor);
1092
1093            if (deviceProp.major > 999)
1094            {
1095                printf(\"warning, CUDA Device  Emulation (CPU) detected, exiting\\n\");
1096                        exit(1);
1097            }
1098
1099
1100            cudareturn=cudaSetDevice(gpudevice);
1101            if (cudareturn == cudaErrorInvalidDevice)
1102            {
1103               perror(\"cudaSetDevice returned  cudaErrorInvalidDevice\");
1104            }
1105            else
1106            {
1107
1108                cudaGetDevice(&device);
1109                 printf(\"Device =%d is  valid.\\n\",device);
1110            }
1111        }
1112
1113        void  driver_version(){
1114
1115                 int driverVersion=0;
1116                 cudaDriverGetVersion(&driverVersion);
1117                 printf(\"CUDA Driver Version: %d.%d\\n\", driverVersion/1000, driverVersion%100);
1118         }
1119
1120        void runtime_version(){
1121                 int runtimeVersion=0;
1122                 cudaRuntimeGetVersion(&runtimeVersion);
1123                 printf(\"CUDA Runtime Version: %d.%d\\n\", runtimeVersion/1000, runtimeVersion%100);
1124
1125         }
1126        void setDevice(int device){
1127                 cudaError_t cudareturn;
1128                 cudareturn=cudaSetDevice(device);
1129                 if (cudareturn == cudaErrorInvalidDevice)
1130                 {
1131                         perror(\"cudaSetDevice returned  cudaErrorInvalidDevice\");
1132                 }
1133
1134        }
1135
1136        int getDeviceId(){
1137                 int device;
1138                 cudaGetDevice(&device);
1139                 return device;
1140         }
1141
1142        void resetDevice(int device){
1143                 cudaSetDevice(device);
1144                 cudaDeviceReset();
1145        }
1146
1147        void resetAllDevice(){
1148                 int devCount;
1149                 cudaGetDeviceCount(&devCount);
1150                 for(int i=0; i< devCount;i++){
1151                         cudaSetDevice(i);
1152                         cudaDeviceReset();
1153                 }
1154         }
1155
1156
1157
1158        void printDevProp(cudaDeviceProp devProp)
1159        {
1160          printf(\"Major revision number:         %d\\n\",devProp.major);
1161          printf(\"Minor revision number:         %d\\n\",devProp.minor);
1162          printf(\"Name:                          %s\\n\",devProp.name);
1163          printf(\"Total global memory:           %.0f MBytes (%llu bytes)\\n\",
1164                 (float)devProp.totalGlobalMem/1048576.0f, (unsigned long long) devProp.totalGlobalMem);
1165          printf(\"Total shared memory per block: %u\\n\",devProp.sharedMemPerBlock);
1166          printf(\"Total registers per block:     %d\\n\",devProp.regsPerBlock);
1167          printf(\"Warp size:                     %d\\n\",devProp.warpSize);
1168          printf(\"Maximum memory pitch:          %u\\n\",devProp.memPitch);
1169          printf(\"Maximum threads per block:     %d\\n\",devProp.maxThreadsPerBlock);
1170          for (int i = 0; i < 3; ++i)
```

```
         printf(\"Maximum dimension %d of block: %d\\n\",i,devProp.maxThreadsDim[i]);
         for (int i = 0; i < 3; ++i)
         printf(\"Maximum dimension %d of grid:  %d\\n\",i,devProp.maxGridSize[i]);
         printf(\"Clock rate:                    %d\\n\",devProp.clockRate);
         printf(\"Total constant memory:         %u\\n\",devProp.totalConstMem);
         printf(\"Texture alignment:             %u\\n\",devProp.textureAlignment);
         printf(\"Concurrent copy and execution: %s\\n\",(devProp.deviceOverlap ? \"Yes\" : \"No\"));
         printf(\"Number of multiprocessors:     %d\\n\",devProp.multiProcessorCount);
         printf(\"Kernel execution timeout:  %s\\n\",(devProp.kernelExecTimeoutEnabled? \"Yes\":\"No\"));

}

        void deviceQuery(){
                int devCount;
                cudaGetDeviceCount(&devCount);
                printf(\"CUDA Device Query...\\n\");
                printf(\"There are %d CUDA devices.\\n\", devCount);

                // Iterate through devices
                for (int i = 0; i < devCount; ++i)
                {
                        // Get device properties
                        printf(\"\\nCUDA Device #%d\\n\", i);
                        cudaDeviceProp devProp;
                        cudaGetDeviceProperties(&devProp, i);
                        printDevProp(devProp);
                }
    }
}\n\n" cudalib-file-port)
 (force-output cudalib-file-port)
)
```

Listing H.5: Code snippet in Gambit that generates CUDA-C functions for the Scheme library functions to manage GPUs

This code snippet in Listing G.6 implements macros for GPUs. This code snippet is in file `_nonstd.scm` under `lib` directory of Gambit development source `4.6.2`.

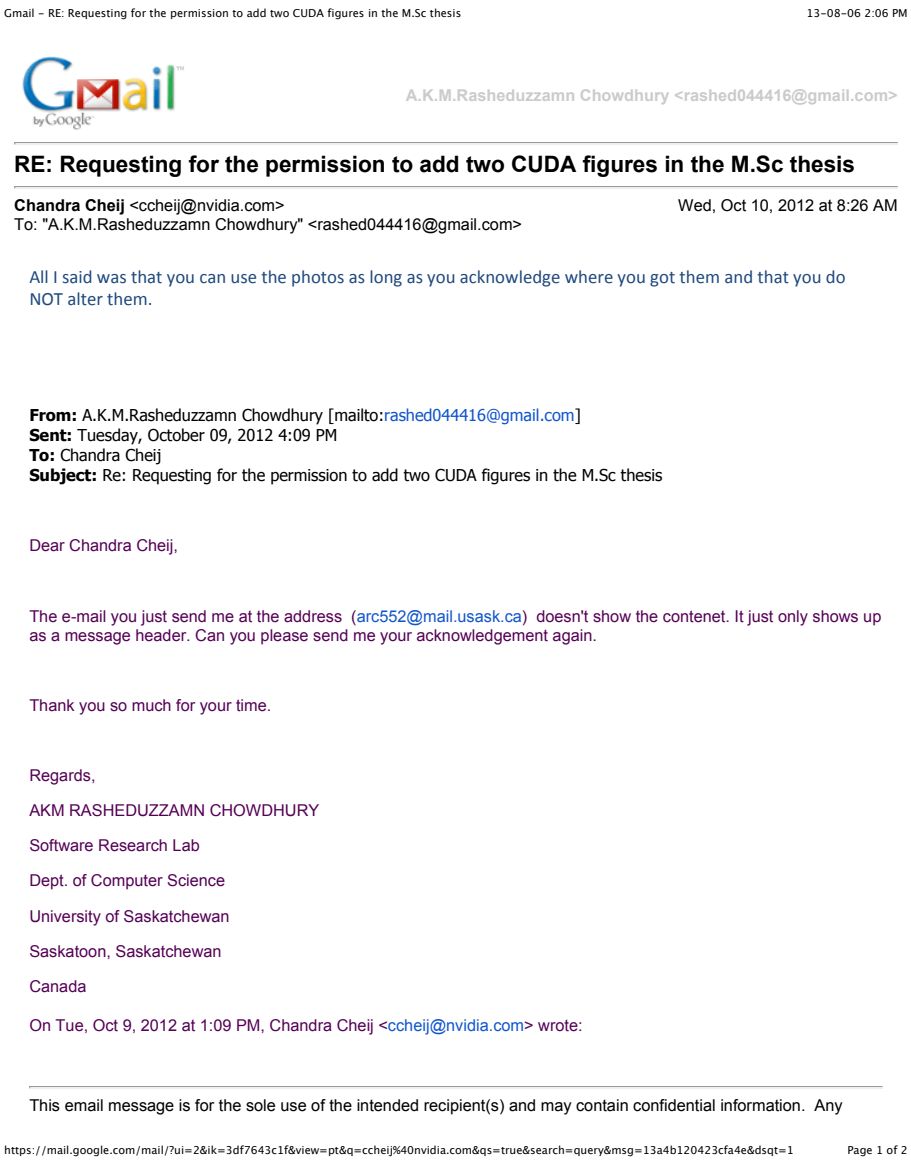
```
(define-runtime-macro (async . args)
  (if (##null? args)
      (display "Warning! async  does not have a kernel call")
      (if (##null? (##cdr args))
          (##car args)
          `(let ((temp ,(##car args)))
              temp
              (async ,@(##cdr args))))))

(define-runtime-macro (sync . args)
  (if (##null? args)
      (display "Warning! sync does not have a kernel call")
      (if (##null? (##cdr args))
          `(begin
             ,(##car args))
          `(let ((temp ,(##car args)))
              temp
             (call-syncthread)
             (sync ,@(##cdr args))))))

(define-runtime-macro (gpu-time . args)
  (if (##null? args)
      (display "Warning! gpu-time does not have a kernel call")
      `(begin
         (call-cudaeventlib-bf-gpu-call)
```

```
2736          (begin ,@args)
2737          (call-cudaeventlib-af-gpu-call))))
```

Listing H.6: Code snippet in Gambit that implements runtime macros for our implementation

Appendix I

NVIDIA permission



Gmail - RE: Requesting for the permission to add two CUDA figures in the M.Sc thesis 13-08-06 2:06 PM

Gmail by Google

A.K.M.Rasheduzzamn Chowdhury <rashed044416@gmail.com>

RE: Requesting for the permission to add two CUDA figures in the M.Sc thesis

Chandra Cheij <ccheij@nvidia.com> Wed, Oct 10, 2012 at 8:26 AM
To: "A.K.M.Rasheduzzamn Chowdhury" <rashed044416@gmail.com>

All I said was that you can use the photos as long as you acknowledge where you got them and that you do NOT alter them.

From: A.K.M.Rasheduzzamn Chowdhury [mailto:rashed044416@gmail.com]
Sent: Tuesday, October 09, 2012 4:09 PM
To: Chandra Cheij
Subject: Re: Requesting for the permission to add two CUDA figures in the M.Sc thesis

Dear Chandra Cheij,

The e-mail you just send me at the address (arc552@mail.usask.ca) doesn't show the contenet. It just only shows up as a message header. Can you please send me your acknowledgement again.

Thank you so much for your time.

Regards,

AKM RASHEDUZZAMN CHOWDHURY

Software Research Lab

Dept. of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

On Tue, Oct 9, 2012 at 1:09 PM, Chandra Cheij <ccheij@nvidia.com> wrote:

This email message is for the sole use of the intended recipient(s) and may contain confidential information. Any

https://mail.google.com/mail/?ui=2&ik=3df7643c1f&view=pt&q=ccheij%40nvidia.com&qs=true&search=query&msg=13a4b120423cfa4e&dsqt=1 Page 1 of 2

Figure I.1: Permission from NVIDIA corporation to use Figures 2.1 and 2.2 in this thesis