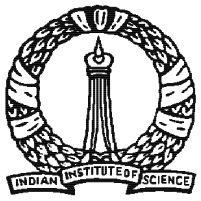
Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Ashwin Prasad



Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

January 2012

To,

Appa, Amma, Thatha and Pati

Acknowledgements

I express my sincere gratitude to my advisor, Prof. R. Govindarajan, for his unlimited patience, guidance and encouragement. None of the work described in this thesis would have been possible without his invaluable technical and moral support.

During my stay at IISc, I have had the pleasure of meeting several remarkable people. I owe my colleague Jayvant Anantpur a sincere debt of gratitude for his help with the implementation of the compiler described in this thesis, for his many useful pieces of advice and for so many rides home that I can't even count. I would also like to thank Mani for many suggestions about the work described in this thesis and for several extremely interesting technical and non-technical discussions, not to mention innumerable "free" lunches. I am grateful to the other members of the HPC lab, Sreepathi, Rupesh, Sandya, Prasanna, Nagendra, Aravind, Raghu and Radhika, for creating a fun work atmosphere. I would also like to thank my other friends at IISc – Bruhathi for her support and understanding, Anirudh for several humorous and relaxing conversations, Kaushik for his infinite wisdom and "smartness", Nithin for many interesting conversations over coffee and Pranava for showing me that the world has bigger pessimists than me! My friends outside of IISc, Vijay, Ananth, Sham, Vikas, Sharath, Shalini, Nirupama, Anand Ramaswamy, Anand Kodaganur and Praveen provided welcome distraction from work.

Finally, I would like to thank my family. I owe my parents and my brother Nitin a huge debt of gratitude for the unconditional support and encouragement they have always given me. I am deeply indebted to my grandparents for always cheering me up regardless of how bad things seemed. I would also like to thank my cousins, Abhay, Varun and Anuj, for our innumerable fun escapades together.

Publications based on this Thesis

1. Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. "*Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors*". In the Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-2011), San Jose, California, June 2011.

2. Ashwin Prasad and R. Govindarajan. "*Compiler Optimizations to Execute MATLAB Programs on Memory Constrained GPUs*". (Under Submission)

Abstract

MATLAB is an array language, initially popular for rapid prototyping, but is now being increasingly used to develop production code for numerical and scientific applications. Typical MATLAB programs have abundant data parallelism. These programs also have control flow dominated scalar regions that have an impact on the program's execution time. Today's computer systems have tremendous computing power in the form of traditional CPU cores and also throughput-oriented accelerators such as graphics processing units (GPUs). Thus, an approach that maps the control flow dominated regions of a MATLAB program to the CPU and the data parallel regions to the GPU can significantly improve program performance.

In this work, we present the design and implementation of MEGHA, a compiler that automatically compiles MATLAB programs to enable synergistic execution on heterogeneous processors. Our solution is fully automated and does not require programmer input for identifying data parallel regions. Our compiler identifies data parallel regions of the program and composes them into kernels. The kernel composition step eliminates a number of intermediate arrays which are otherwise required and also reduces the size of the scheduling and mapping problem the compiler needs to solve subsequently. The problem of combining statements into kernels is formulated as a constrained graph clustering problem. Heuristics are presented to map identified kernels to either the CPU or GPU so that kernel execution on the CPU and the GPU happens synergistically, and the amount of data transfer needed is minimized. A heuristic technique to ensure that memory accesses on the CPU exploit locality and those on the GPU are coalesced is also presented. In order to ensure that data transfers required for dependences across basic blocks are performed, we propose a data flow analysis step and an edge-splitting

strategy. Thus our compiler automatically handles kernel composition, mapping of kernels to CPU and GPU, scheduling and insertion of required data transfers.

Additionally, we address the problem of identifying what variables can coexist in GPU memory simultaneously under the GPU memory constraints. We formulate this problem as that of identifying maximal cliques in an interference graph. We approximate the interference graph using an interval graph and develop an efficient algorithm to solve the problem. Furthermore, we present two program transformations that optimize memory accesses on the GPU using the software managed scratchpad memory available in GPUs.

We have prototyped the proposed compiler using the Octave system. Our experiments using this implementation show a geometric mean speedup of 12X on the GeForce 8800 GTS and 29.2X on the Tesla S1070 over baseline MATLAB execution for data parallel benchmarks. Experiments also reveal that our method provides up to 10X speedup over hand written GPUmat versions of the benchmarks. Our method also provides a speedup of 5.3X on the GeForce 8800 GTS and 13.8X on the Tesla S1070 compared to compiled MATLAB code running on the CPU.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

During the nineties and the early part of the twenty first century, architects were able to use the growing number of transistors available on chips to improve the single threaded performance of general purpose processors. This was primarily due to the design of processors with larger caches, wider instruction issue widths and deeper pipelines. These methods of improving performance are transparent to user programs, meaning users did not have to rewrite their programs to get additional performance on newer machines. Newer generations of processors automatically executed single threaded programs faster by extracting more Instruction Level Parallelism (ILP) and because of improved cache hit rates.

However, even though feature sizes continued to decrease, investing a larger number of transistors to exploit ILP started yielding diminishing returns in terms of single threaded program performance. Further, as operating frequencies increased, the power dissipation of newer generations of microprocessors increased drastically, making power dissipation and wire delays [43] first order processor design constraints. To counter the increasing power consumption and wire delays, most modern microprocessors are multi-core processors. Therefore, the performance of single threaded applications no longer improves automatically when they are run on newer processors.

To get optimal performance on modern processors, programmers need to rewrite their sequential applications to expose parallelism explicitly. This complicates application development because programmers now need to develop parallel programs, which is a harder task than devel-

oping sequential applications. This is primarily because problems such as deadlocks and data races make it difficult to write correct parallel code. Additionally, debugging parallel programs is very hard due to their non-deterministic execution.

1.1 Trends in Parallel Processors

Current processors from all major processor manufacturers are multi-core processors. For example, IBM's POWER7 [59], has eight cores, each with 32KB of private L1 cache and 256KB of private L2 cache. There is also 32MB of shared L3 cache. Current Intel Xeon processors like the Nehalem-EP (Xeon X5570) have four cores and have up to 12MB of last level cache [30]. The current trend in the design of multicore processors is an increasing number of cores on a single chip, with each core having one or two levels of private cache. These processors also have a last level cache shared across multiple cores. This architecture is well suited to general purpose computations where task-level or thread-level parallelism can be exploited. The architecture is optimized to run control-intensive code efficiently. It is also well suited for running multiprogrammed workloads. While the multicore architectures work well for most general-purpose applications, many scientific and streaming applications exhibit a lot of fine grained data parallelism. These applications could potentially benefit from a large number of relatively small processing cores. This has led to the design and development of several *accelerators* [39, 47, 64].

Accelerators are typically very high throughput processors capable of performing specific types of computations very fast. Initially, accelerators were special purpose processors designed to accelerate a particular class of applications like Graphics Processing Units and accelerators for cryptography applications [55, 76]. These processors were mostly fixed function devices that could not be programmed very easily [55]. More recently, these processors have become much more programmable thus enabling the acceleration of a much broader class of applications. Of all the accelerators currently available, the two most widely used are probably the Cell Broadband Engine (Cell BE) [39] and Graphics Processing Units (GPUs). Other accelerators are the Larrabee from Intel [64], AMD Fusion [7] which integrates CPU cores and a GPU on

the same chip and custom accelerators constructed using Field Programmable Gate Arrays (FPGAs) [62, 63]. In this section, we briefly discuss the Cell BE and GPUs. The focus of the rest of this thesis is on GPUs.

The Cell BE was developed by Sony, Toshiba and IBM in the mid-2000s. The Cell contains a single general purpose Power PC core, called the Power Processing Element (PPE) and eight co-processors called Synergistic Processing Elements (SPEs). These processing elements are connected by a high speed ring network. The SPEs are SIMD processors each having 256KB of local memory and DMA engines to transfer data from main memory to the SPE's local memory or vice-versa. The SPEs accelerate the compute intensive parts of the application running on the Cell. However, offloading work to the SPEs is the responsibility of the programmer. Further, he has to DMA the required data to an SPE's local store and then launch a computation on the SPE. Even though IBM provides a complete Linux based development platform for the Cell BE and a compiler for Cell, software development for the Cell is considered to be challenging [44, 65]. The Cell has therefore not gained very widespread adoption.

Recent advances in the design of Graphics Processing Units (GPUs) [6, 47, 52] have significantly increased the compute capabilities and programmability of these processors. Graphics Processing Units, have evolved from being fixed function processing units to powerful programmable stream processors capable of delivering up to 2 TFLOPs of processing power [52]. GPUs are commodity processors and present a very affordable alternative to specialized accelerators. Due to their ability to accelerate data-parallel applications, GPUs are well suited to accelerating numerical and scientific computation. They have become the accelerators of choice because of their low cost, tremendous computational power, ubiquity and programmability.

Typically GPUs have tens of processing cores, called *Streaming Multiprocessors*(SMs) in the NVIDIA architecture. Each SM has several execution pipelines which execute instructions in a SIMD fashion. A GPU typically has a few hundred cores that can execute different threads. The hardware is capable of quickly switching between threads to hide latency due to memory accesses or dependences. Thus the GPU hardware supports thousands of concurrently active threads. The GPU has a global memory that is implemented using a DRAM. Each SM also has a fast software managed cache, called *Shared Memory* in the NVIDIA architecture. More

details about the NVIDIA GPU architecture are provided in Chapter 2.

Almost all desktops and servers currently manufactured have GPUs in addition to having multiple CPU cores. The presence of heterogeneous cores (CPU cores and GPU cores) on these machines further complicates the task of programming them. Several frameworks have been designed to allow developers to write programs for these heterogeneous machines. The following section discusses some of these.

1.2 Programming Models for Heterogeneous Platforms

Early programming models for GPU based heterogeneous platforms were not very easy to use. Programmers had to use either low level programming interfaces like ATIs CAL [5] and CTM [33], that were almost at the level of assembly language, or write GPU programs in shader languages like `Cg` and run them using standard graphics APIs like OpenGL [67]. However, these were superseded by platforms such as CUDA [51] and OpenCL [42] that give the user a much higher level abstraction to work with. In these frameworks, programmers can program GPUs in a language similar to `C`. This has led to them gaining widespread adoption in a short period of time.

However, programming in these platforms remains a challenge mainly because they expose a programming interface that requires the programmer to have a detailed understanding of the machine architecture. Firstly, the programmer has to manually partition tasks between the CPU and the GPU. He needs to identify the parts of the program that can possibly benefit from GPU acceleration and rewrite these in a form that is suitable for execution on the GPU. Once the work is partitioned, the programmer needs to orchestrate data movement since the CPU and GPU work in disjoint address spaces. He needs to identify inputs and outputs of various tasks and make sure that data is moved correctly so that an up to date copy is available to the device trying to access it. These factors make the process of programming the GPU an error prone and tedious one.

Writing performant programs is an even harder proposition due to the multitude of factors that affect execution on the GPU [34, 8]. It is extremely difficult for programmers to intuitively

ascertain what changes will improve performance and what changes will not. Also, it is not immediately obvious, which parts of the program will benefit from being moved to the GPU because gains will need to amortize the data movement costs. This essentially makes the process of writing performant code on heterogeneous machines an iterative one. Moreover, the decisions made in the context of one accelerator need not be applicable even when the program is run on a different generation of the same accelerator.

The fact that programming heterogeneous platforms is cumbersome and error prone has been observed by several researchers [71, 72, 69, 14]. They have recognized the the need to raise the level of abstraction of the GPU programming model. Doing this significantly eases the programmers' task as the many idiosyncrasies of the underlying architecture are hidden by the compiler and runtime. The following paragraphs describe some of the past work in this direction.

Microsoft Accelerator [69] is a .NET framework library [49] that allows users to run code on the GPU without having to write GPU code explicitly and without having to manage details of data movement. It defines a type called the Data Parallel Array. All operations on Data Parallel Array objects are offloaded to the GPU. The system automatically generates GPU code for the specified operations and handles all necessary data movement.

Brook [14] is an extension of ANSI C which provides abstractions to simplify GPU programming in a familiar language. It provides the programmer with a stream abstraction, which is the data type whose operations are performed on the GPU. Operations on streams are specified as kernels. Brook also allows the user to perform operations such as reduction on streams. To program in this model, the user need not be concerned with the details of the underlying system.

StreamIt [70], a high-level stream programming language specifically designed for media applications, has also been successfully compiled to heterogeneous machines [71, 72]. A StreamIt program consists of a set of nodes which perform operations on data and a set of FIFO channels through which the nodes communicate. A StreamIt graph is constructed by a hierarchical composition of filters using the basic StreamIt constructs: Pipeline, Split-Join and the Feedback Loop. Udupa et al propose automatic compilation of StreamIt programs for syn-

ergistic execution on CPU and GPU cores. In particular, stateless StreamIt programs can be executed in a pipeline purely on the GPU [71]. More complicated StreamIt programs are compiled to work in a pipelined manner across the CPU and GPU [72]. Their compiler can software pipeline the execution of multiple kernels to exploit task, data and pipeline parallelism.

1.3 MATLAB on GPUs : Motivation

We believe that the abstraction for programming heterogeneous architectures should be a language or API that does not require the programmer to (i) know the target architecture in great detail (ii) program and manage data explicitly and (iii) tune program parameters differently for different architectures. Further, it should be a language that has a wide user base and is familiar to a lot of programmers. This helps a large set of existing programs leverage the performance benefits of accelerators. MATLAB [48] provides one such well suited abstraction. MATLAB [48] is an array language popular for implementing numerical and scientific applications. MATLAB programs are declarative and naturally express data-level parallelism as the language provides several high-level operators that work directly on arrays. Traditionally, MATLAB is used as programming language to write various types of simulations. It is used extensively to simulate and design systems in areas like control engineering, image processing and communications. These programs are typically long running and developers expend significant effort in trying to shorten their running times. Currently, MATLAB programs are translated into compiled languages like C or Fortran to improve performance. These translations are normally done either by hand or by automated systems that compile MATLAB code to C or Fortran [22, 38].

Many projects have focussed on compiling MATLAB programs to languages such as FORTRAN [22] or C [38]. The resulting code was compiled and executed on general purpose CPUs. Just-in-time compilation techniques for execution of MATLAB programs on CPUs were implemented in MaJIC [4] and McVM [19]. However, these efforts on automatic translation are limited to CPUs and do not consider GPUs. Further, they do not try to exploit the data parallelism inherent in MATLAB programs. MATLAB programs typically operate on extremely

large matrices and operations on large matrices have a lot of data-parallelism. They are therefore ideal candidates for acceleration using throughput-oriented processors like GPUs.

Currently, Jacket [35] and GPUmat [28] provide the user with the ability to run MATLAB code on GPUs. However these are not automatic systems and require users to specify arrays whose operations are to be performed on the GPU by declaring them to be of a special type. The user needs to identify which operations are well suited for acceleration on the GPU, and is required to manually insert calls to move the data to and from the GPU memory. The following listing is pseudocode that shows how a programmer would perform a matrix addition on the GPU using either GPUmat or Jacket.

```
1  A = rand(10, 10);
2  B = rand(10, 10);
3  A_gpu = GPUtype(A);
4  B_gpu = GPUtype(B);
5  C_gpu = A_gpu + B_gpu
6  C = CPUtype(C_gpu);
```

Two 10×10 matrices, `A` and `B` are first defined. These are then transferred to the GPU by casting them to the type `GPUtype`. The addition on line 5 is performed on the GPU because both operands are GPU resident arrays. To bring the result of the addition back to the host, C_{gpu} needs to be cast to the type `CPUtype`. This forces the contents of the matrix C_{gpu} to be transferred to CPU memory.

To extract the maximum performance from modern heterogeneous machines, programmers need to use all available computing resources, namely CPU cores as well as available accelerators. However, as described above, manually translating MATLAB programs to use available accelerators can be a tedious and error-prone process because the programmer is required to manage details such as data transfer, mapping and scheduling. Additionally, the quality of the partitioning of tasks across the available processors is dependent on the target machine, i.e., a good partition on one machine is not necessarily a good partition on all machines. Therefore, obtaining portable performance even for hand-translated code is not straightforward.

Given these challenges and the wide-spread adoption of MATLAB in the scientific community, a tool to automatically transform MATLAB programs so that accelerators are used to

speedup the data-parallel parts of the program and the CPU is used to efficiently execute the rest of the program would be very useful. In this thesis, we describe MEGHA[1], a compiler that automatically compiles MATLAB programs for execution on machines that contain both CPU and GPU cores. To the best of our knowledge, MEGHA is the first effort to *automatically* compile MATLAB programs for parallel execution across CPU and GPU cores.

Our compiler automatically identifies regions of MATLAB programs that can be executed efficiently on the GPU and tries to schedule computations on the CPU and GPU in parallel to further increase performance. There are several associated challenges. First, the CPU and GPU operate on separate address spaces requiring explicit memory transfer commands for transferring data from and to the GPU memory. Second, sets of statements (kernels) in the input program that can be run together on the GPU need to be identified. This is required to eliminate intermediate arrays and to reduce the size of the scheduling problem the compiler needs to solve. Third, program tasks[2] need to be mapped to either GPU cores or CPU cores and scheduled so that the program execution time is minimized. Time overheads for the required data transfers also need to be taken into account while mapping and scheduling tasks. Finally, efficient code needs to be generated for each task depending on which processor it is assigned to.

1.4 Contributions

In this thesis, we describe the structure of a compiler that translates MATLAB programs to CUDA. We also present possible solutions to each of the problems described above. The main contributions of this thesis are as follows.

- We present a compiler framework, MEGHA, which translates MATLAB programs for synergistic parallel execution on machines with GPU and CPU cores.
- We formulate the identification of data parallel regions of a MATLAB program as a constrained graph clustering problem and propose an efficient heuristic algorithm to solve it.

[1] **M**ATLAB **E**xecution on **G**PU based **H**eterogeneous **A**rchitectures.

[2] Statements that can only run on the CPU and kernels (which can be run on CPU or GPU) are collectively called tasks.

- We propose an efficient heuristic algorithm to map kernels to either the CPU or GPU and schedule them so that memory transfer requirements are minimized. Techniques for generating efficient code for kernels mapped to either the CPU or GPU are also described.

- We describe two optimizations on memory accesses in GPU code that significantly improve the performance of generated code. These optimizations automatically make use of the software-managed *shared memory* present in GPUs.

- We formulate the problem of managing GPU memory as that of computing maximal cliques of an interference graph. As this problem is NP-hard, we simplify it and present an efficient algorithm to obtain approximate solutions.

- Experimental evaluation on a set of MATLAB benchmarks shows a geometric mean speedup of 12X on the GeForce 8800 GTS and 29.2X on the Tesla S1070 over baseline MATLAB execution for data parallel benchmarks. Experiments also reveal that our method provides up to 10X speedup over hand written GPUmat versions of the benchmarks.

The rest of this thesis is organized as follows: Chapter 2 briefly reviews the necessary background. In Chapter 3, we provide an overview of the compiler and describes the details of our proposed compilation methodology. Chapter 4 describes additional optimizations performed by the compiler. We also present our experimental methodology and report performance results in each of these chapters. Chapter 5 compares our work with other related work. Chapter 6 discusses possible directions for future work and concludes.

Chapter 2

Background

In this chapter, we review the necessary background for the work described in the remainder of this thesis. Section 2.1 describes the NVIDIA GPU architecture. The CUDA programming model is described in Section 2.2. In Section 2.3 we provide an overview of the MATLAB language. Section 2.4 discusses some of the assumptions we make about the input MATLAB program.

2.1 NVIDIA GPU Architecture

This section describes the organization of the NVIDIA GPUs in the context of with the the GeForce 8800 GTS 512 GPU. The architecture of the other NVIDIA GPUs are similar, differing only in details such as the number of SMs they have.

The high level architecture of the NVIDIA GeForce 8800 GTS 512 GPU [51] is shown in Figure 2.1(a). It has 16 Streaming Multiprocessors (SMs), shown as SM_0–SM_{15} in Figure 2.1(a). Each SM has several functional units and is capable of processing several hardware threads in parallel. The GPU is connected to a global device memory (implemented using a DRAM) through a high bandwidth memory bus, which is up to 512 bits wide. However, the latency of accessing the global memory is high (600–800 cycles) [51].

The structure of each Streaming Multiprocessor is shown in Figure 2.1(b). Every Streaming Multiprocessor has eight *Scalar Units* (SUs). All the SUs in an SM share an instruction fetch

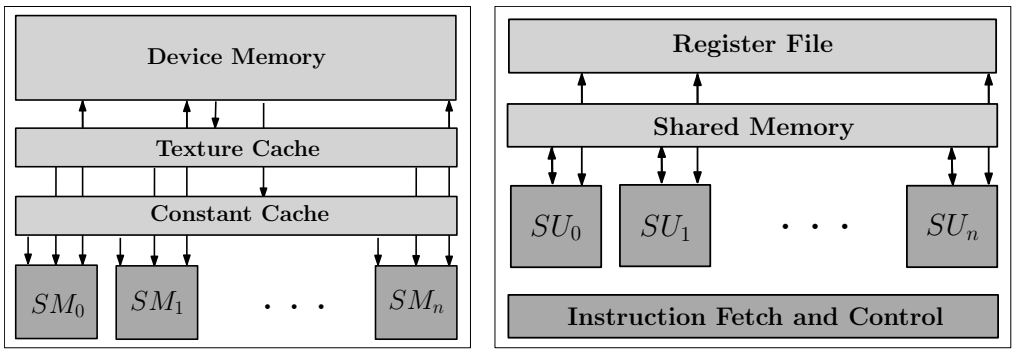


Figure 2.1: NVIDIA GPU Architecture

unit and control unit. In any given cycle, each of the SUs in an SM execute the same instruction, i.e., they operate in *lock step*. Each SM has can have hundreds of "active" threads, many more than the eight SUs in an SM, and executes them in a multithreaded manner. These threads are however much lighter weight than threads on the CPU and switching between them is a very fast operation. The SM switches between threads to hide the latency of memory accesses as well as to hide the latency due to true dependences.

The GPU also has two read-only caches. One is a *constant cache* that caches memory addresses which contain variables declared to be constant at compile time. The other cache is a *texture cache* which attempts to exploit locality in the way textures (images which are rendered onto the screen) are accessed. Memory needs to be accessed through special operations called *texture fetches* for the read values to be cached in the texture cache. Newer NVIDIA GPUs, like the Fermi [52], also have a per SM L1 data cache and an L2 cache which is shared across all SMs.

Code is executed on the GPU through what are called *kernel* calls. A kernel call specifies the number of threads that are to be created when the kernel is executed. Each thread executes exactly the same static code. However, threads are free to follow different control flow paths at run-time. When threads follow different control flow paths, these threads are said to *diverge*. The hardware ensures that diverged threads reconverge at the control flow merge point [27]. The hardware assigns an integer *thread ID* to each thread and 32 successive threads are grouped into what is called a *warp* (defined precisely in the following section). The warp is the fundamental

schedulable entity on NVIDIA GPUs and all threads that belong to the same warp execute in lock step.

Each SM also has a register file containing 8192 32-bit registers. These registers can hold either integers or floating point values. Values in registers can be read in a single cycle. SMs also have 16kB of *shared memory*. Shared memory is a fast on-chip memory which is similar to a software managed cache and is explicitly managed by the programmer. All simultaneously active threads get disjoint sets of registers and disjoint space in shared memory. Therefore, the total register and shared memory usage of all threads that are active on an SM simultaneously cannot exceed the amount available per SM. Thus, the number of threads which can be simultaneously active on an SM is decided by the shared memory and register usage of each thread. The address spaces of the CPU and GPU are disjoint. It is the programmer's responsibility to transfer required data to and from GPU memory using DMA transfers. The CUDA framework provides functions to perform DMA transfers and manage GPU memory (Section 2.2).

The GPU device memory is organized as multiple banks and is connected to the GPU through a very wide memory bus. This bus is 256 or 512 bits wide for 8800 series GPUs depending on the model. The device memory is capable of servicing requests at a very high bandwidth because of the wide bus. However, the latency of the device memory is large (500-700 cycles). The memory is capable of operating at close to its ideal bandwidth when memory accesses are *coalesced*. Coalesced accesses[1] are accesses in which all threads in a warp access consecutive memory addresses, i.e., the address accessed by thread n of a warp must be of the form $BaseAddress + n$. In this case, all the loads of the threads of the warp can be serviced by a single memory transaction, therefore better utilizing the high bandwidth provided by the wide memory interface.

It is important to note that the GPU is only a *slave processor* to the host CPU processor. The GPU cannot initiate actions such as kernel invocations and DMA transfers. All such actions must be initiated by the host processor. GPUs have very limited support for synchronization. The hardware does however provide the following synchronization primitives :

- The GPU provides *barrier* synchronization for threads running on the same SM.

[1]Recent processors have more relaxed constraints for coalesced accesses.

- GPUs have support for basic *atomic primitives* (like atomic add and atomic compare and swap) for synchronization across threads running on different SMs. These primitives can be used to build mechanisms like mutexes.

GPUs also have weak consistency guarantees [32, 1]. Memory operations from a single thread are guaranteed to be sequentially consistent [51]. However, only a relaxed ordering is guaranteed across threads. All writes by threads running on an SM are guaranteed to be visible to all threads running on that SM after a barrier synchronization has been performed. However, since no guarantee exists on the order in which threads are executed, no global barriers can exist. Therefore, writes to device memory are guaranteed to be visible to threads on different SMs only across kernel call boundaries.

2.2 CUDA Programming Model

NVIDIA CUDA is an extension of the C++ programming language. It extends the base C++ language to provide data parallel constructs to write code that is to run on the GPU. Programs written in CUDA have two distinct parts – the part of the program written in base C++ which is to execute on the host processor and code written using the CUDA language extensions that is to execute on the device (GPU). The CUDA toolkit provides a compiler, `nvcc`, to compile CUDA programs into executables. When the resulting executable is run, it starts executing on the CPU and then initiates required DMA transfers and GPU kernel executions.

2.2.1 CUDA Terminology

This section provides definitions for some terms which are used throughout the thesis. More details can be found in the CUDA programming guide [51].

- **Thread Block:** A thread block is a user specified set of threads that run on the same SM. Thread blocks have up to 512 threads on the NVIDIA 8800 GTX (This number is larger on the Fermi GPUs). The user specifies the geometry and size of each thread block at kernel launch time. Blocks can have up to three dimensions. Threads within the same thread

block have access to the same region of shared memory and can therefore communicate through it. They can also synchronize using the thread barrier, `__syncthreads()`. Therefore, thread blocks are also called *Cooperative Thread Arrays* (CTAs).

- **Grid:** A grid is a set of thread blocks. Each kernel invocation launches exactly one grid. The size and geometry of the grid is specified by the programmer at kernel launch time. A grid can have up to two dimensions[2].

- **Warp:** A warp is a group of 32 threads having consecutive *threadIds*. A warp is the granularity at which the hardware schedules work. All threads in a warp execute in lock-step. Warps are not visible to the programmer.

- **Execution Configuration:** The programmer needs to specify a set of parameters before a kernel can be launched. He needs to specify the size of each thread block, the size of the grid and the shared memory usage of each thread block.

In a CUDA program, the programmer is responsible for assigning each function to run on either the CPU or the GPU. CUDA extends C++ with several keywords to allow the programmer to do this.

- The `__global__` qualifier indicates that a function is to be run on the GPU. It also indicates that the annotated function can be called from CPU code, i.e. the function is an entry point for a GPU kernel.

- The `__device__` keyword indicates that a function will be run on the GPU but will only be called from GPU code, i.e. it will only be called from another function marked either `__global__` or `__device__`. The `__device__` keyword is also used to specify that a particular global variable resides in GPU memory.

Programmers also need to specify the location of variables in the GPU memory hierarchy. This is done using keywords that are type modifiers. They specify exactly which part of the GPU memory hierarchy a variable resides in.

[2]The recently released CUDA 4.0 supports grids with three dimensions. In this thesis, we only use two dimensional grids.

- Programmers may specify that a variable resides in GPU shared memory using the `__shared__` type qualifier. Variables that are declared to be in shared memory are shared across all threads in a thread block.

- Variables that are not modified by GPU code are defined using the `__constant__` type qualifier. These variables reside in constant memory and are cached in the constant cache.

To write useful CUDA programs, programmers need to be able to dynamically determine information about the execution configuration and the identity of the thread currently being executed. CUDA allows access to this information inside a kernel through the following built-in variables.

- `gridDim` is a 3-tuple that contains the dimensions of the grid.

- `blockDim` is a 3-tuple that contains the dimensions of the thread block.

- `blockIdx` is a 3-tuple that contains the index of the thread block in the grid.

- `threadIdx` is a 3-tuple that contains the index of the thread in the containing thread block.

2.2.2 The CUDA Runtime API

The CUDA framework provides a high-level API, called the run-time API, to perform operations such as DMA transfers, device memory management, event management and stream management. This section provides a brief description of the more commonly used parts of the CUDA run-time API. CUDA provides many more API functions whose details can be found in the CUDA Programming Guide [51].

DMA Transfers: The CUDA runtime API provides two functions to perform DMA transfers. The first is `cudaMemcpy` which is a synchronous function and allows the user to transfer data between the CPU memory and GPU memory or vice versa. A call to `cudaMemcpy` blocks the calling thread until the DMA transfer is completed. CUDA also provides an API function, `cudaMemcpyAsync` to start DMA transfers asynchronously. A call to this function queues a request for a DMA transfer and returns. It is the programmers responsibility to ensure that the

transfer is actually complete before the results are used. Such synchronization can be achieved through mechanisms like *streams* [51]. These are explained later in this subsection. Recent versions of CUDA also allow these functions to be used for transfers between two GPU devices.

Memory Management: The CUDA API provides functions to allocate and deallocate memory in the GPU memory space. The function `cudaMalloc` provides the user the ability to allocate memory. Memory allocated with `cudaMalloc` can be freed by calling `cudaFree`. These functions can only be called from host code. GPU memory cannot be allocated dynamically from inside a kernel. CUDA also provides the function `cudaMallocHost` to allocate pinned memory on the host side. This is useful as extra buffering can be avoided while DMA transfers are being performed when the source buffer is pinned. Pinned memory is required for calls to `cudaMemcpyAsync`.

Stream Management: CUDA provides programmers with an abstraction called *streams* to enable concurrent programming. Streams can be thought of as work queues. Every CUDA operation, like a memory transfer or kernel invocation, can be optionally bound to a stream. The CUDA runtime executes operations bound to the same stream sequentially and in the order in which they were called. However, whenever possible, operations bound to different streams are executed in parallel. The CUDA API provides functions to create and destroy streams. It also provides functions to wait for the completion of all operations bound to a stream.

Event Management: Events provide a way to closely monitor the device's progress, as well as perform accurate timing. Applications can asynchronously record events at any point in the program and query when these events are actually recorded. CUDA provides the `cudaEvent_t` object which can be used to record events. Events can be recorded at any time in the programs execution with the `cudaEventRecord` function. The time between recorded events can be computed using the `cudaEventElapsedTime` function. We use events and their associated functions to profile our generated code at compile time.

2.2.3 A Simple CUDA Example

As an example, consider the problem of having to add the elements of two one-dimensional arrays. The following listing shows a program to do this in C++.

```
1 int main()
2 {
3   int n = getArrayLength();
4   int *a = getVector(n);
5   int *b = getVector(n);
6   int *c = new int[n];
7
8   for(int i=0; i<n ; ++i)
9     c[i] = a[i] + b[i];
10
11  printVector(c, n);
12 }
```

In the above code, we first initialize the vectors `a` and `b`. We then allocate an array `c` to hold the results of the addition. The `for` loop on line 8 goes over each element in the input vectors, sums them and writes the result into the appropriate element of `c`. It is easy to see that each iteration of the loop on line 8 can be executed in parallel since no iteration reads or writes to a location written to by a previous iteration. Therefore, this loop can be written as a CUDA kernel which executes on the GPU.

An equivalent program that executes the vector addition on the GPU is shown below. It performs the following steps.

1. As in the CPU version, initialize the vectors `a` and `b` that need to be added.

2. Allocate memory for `a`, `b` and `c` on the GPU (Lines 13–15).

3. Copy the contents of `a` and `b` from CPU memory to GPU memory (Lines 17–18).

4. Spawn a kernel that adds the vectors `a` and `b` on the GPU (Line 23).

5. Copy the resulting vector from GPU memory to CPU memory (Line 25).

6. Free the memory allocated on the GPU and the CPU (Lines 27–29).

```
1  __global__ void addVectors(int *c, int *a, int *b, int n)
2  {
3    int tidx = blockDim.x*blockIdx.x + threadIdx.x;
4    if (tidx >=n ) return;
5    c[tidx] = a[tidx] + b[tidx];
6  }
7  int main()
8  {
9    //Initialize vectors on the CPU ...
10
11   int *aGPU = NULL, *bGPU = NULL, *cGPU = NULL;
12
13   cudaMalloc(&aGPU, n*sizeof(int));
14   cudaMalloc(&bGPU, n*sizeof(int));
15   cudaMalloc(&cGPU, n*sizeof(int));
16
17   cudaMemcpy(aGPU, a, n*sizeof(int), cudaMemcpyHostToDevice);
18   cudaMemcpy(bGPU, b, n*sizeof(int), cudaMemcpyHostToDevice);
19
20   dim3 blockDim(256, 1, 1);
21   int numBlocksX = ceil(n/256.0);
22   dim3 gridDim(numBlocksX, 1, 1);
23   addVectors<<<gridDim, blockDim>>>(cGPU, aGPU, bGPU, n);
24
25   cudaMemcpy(c, cGPU, n*sizeof(int), cudaMemcpyDeviceToHost);
26
27   cudaFree(aGPU);
28   cudaFree(bGPU);
29   cudaFree(cGPU);
30   printVector(c, n);
31 }
```

Before calling the kernel `addVectors`, the size of thread blocks and the size of the grid need to be computed. Line 20 defines the thread block size for this kernel call to be 256. Each thread block has 256 threads in the x-dimension and 1 in each of the y and z dimensions. Lines 21–22

calculate the required grid size to compute all elements of the output vector `c`. Line 23 calls the kernel `addVectors`.

The kernel code is written so that each GPU thread computes the sum of one pair of elements, i.e., each thread performs work equivalent to one iteration of the CPU loop. Each thread uses the variables `threadIdx`, `blockIdx` and `blockDim` to compute the index of the elements it must add and then performs the addition. Once the kernel is executed, line 25 copies the result back to CPU memory. Memory allocated on the GPU is then freed using the `cudaFree` function.

In the next section, we briefly present the necessary background on MATLAB.

2.3 The MATLAB Language

MATLAB is a high-level language developed by MathWorks [48]. It is a dynamically typed array based programming language that is very popular for developing scientific and numerical applications. MATLAB has grown into a diverse and vast language over the years. This section describes some of the important features of the MATLAB language. More information on other MATLAB constructs can be found on the MathWorks website [48].

2.3.1 MATLAB Variables and Operators

MATLAB programs, at a basic level, are similar to programs written in a language like C++. Each program has a set of variables and the program manipulates these variables through operators and function calls. Values are assigned to variables through the assignment operator =. For example,

```
1   a = 42;
```

assigns the integer value 42 to the variable `a`. MATLAB supports variables of several primitive types like `logical`, `int`, `real`, `complex` and `string`. It is also possible to construct arrays with elements of these primitive types. A programmer may construct a matrix of random `real` elements as follows.

```
1  n = 100;
2  a = rand(n, n);
```

In the above example, `a` is a 100×100 matrix of `reals`. Each element is initialized with a random value. In MATLAB, all variables are matrices. Scalars are just single element matrices. In MATLAB however, a variable does not need to be defined to be of a particular type before the variable is used. MATLAB is a *weakly dynamically typed language*. It is said to be *dynamically typed* because the types of variables are determined only at runtime. It is *weakly typed* because the type of a variable can change through the course of a program. For example, the following is a valid MATLAB program.

```
1  a = 42;
2  disp(a); //'a' is an integer
3  // More code ...
4  a = "Hello World";
5  disp(a); //'a' is a string
6  // Even more code ...
```

Here the type of `a` changes from integer to string when a string is assigned to it on line 4. This is one of the features of MATLAB that makes it difficult (if not impossible) to statically compile MATLAB code.

MATLAB provides a rich set of operators to operate on matrices. They are overloaded to perform appropriate actions depending on the size and type of their input operands. Consider the following code segment.

```
1  x = 10;
2  y = 20;
3  a = rand(100, 100);
4  z = x + y;
5  b = a + a;
6  c = x + a;
```

The "+" operators on lines 4, 5 and 6 all perform different operations at runtime. The plus on line 4 performs a scalar addition on the variables `x` and `y`. The + operator on line 5 however adds two 100×100 matrices. The + operator on line 6 adds the scalar `x` to each element of the matrix `a`. All arithmetic operators in MATLAB are similarly overloaded. The $\star$ operator for example

performs the appropriate form of multiplication depending on the sizes of its arguments. For example, if both arguments are matrices, it performs a matrix multiplication. However, if one is a matrix and the other is a vector, it performs a matrix vector multiplication. MATLAB also provides operators to perform element-wise multiplications and division.

```
1  a = rand(100, 100);
2  b = rand(100, 100);
3  c = a .* b;
4  d = a * b;
```

In the above code, each element in `c` is the product of the corresponding elements of `a` and `b` whereas `d` is the matrix product of `a` and `b`.

2.3.2 Control Flow Constructs

MATLAB provides most of the common control flow structures. It has support for *if..else* statements and *for*, *while* and *do..until* loops. It also has support for user-defined functions.

2.3.3 Array and Matrix Indexing

The basic indexing mechanism is the same as in languages like C++ where an array variable is indexed using an integer index. In the following code, line 2 assigns 42 to the fifth element of the vector `a`.

```
1  a = ones(10, 1);
2  a(5) = 42;
```

The function `ones` returns an array of the requested size (a vector of length 10 in this case) with each element initialized with the value 1. MATLAB also supports more sophisticated forms of indexing than the primitive indexing described above. It is possible to index arrays with other arrays. For example, consider the following code segment.

```
1  a = ones(10, 1);
2  i = 1:3;
3  a(i) = 42;
```

`i` is a vector containing the elements 1, 2 and 3 (the colon operator is described below). After line 3 is executed, the first three elements of `a` are assigned the value 42. It is also possible to index arrays with arrays of dimensionality higher than one.

The idiom in the above example is very common in MATLAB because of which MATLAB provides the colon operator. The colon operator can be used to construct arrays whose values are linearly changing according to a predefined step size. The array `a` in the example below has all integers between 10 and 25 in steps of 5, i.e., its elements are 10, 15, 20 and 25.

```
1  a = 10:5:25;
```

The colon also plays a special role in array indexing. It is used to specify all elements of an array along a particular dimension.

```
1  a = ones(10, 10);
2  a(:, 1) = 42;
```

Line 2 in the above example assigns the value 42 to every element in the first column of the 10×10 matrix `a`. MATLAB also provides the keyword `end`. The `end` keyword when used within the indexer for a particular array represents the index of the last element of that array in that dimension.

```
1  a = ones(10, 10);
2  a(5:end, 1) = 42;
```

The last line in the above code segment assigns 42 to elements 5 to 10 of the first column of the matrix `a`. Obviously, such indexing mechanisms are also valid on the right hand side of assignment statements. MATLAB requires the programmer to ensure the compatibility of sub-array dimensions when they are specified by the mechanisms described above.

MATLAB also does not require the indexer of an array to be smaller than the length of the array. When an array is indexed past its end in any dimension, the array simply grows to accommodate the index. Consider the following example.

```
1  a = ones(10, 1); //a is a vector of length 10
2  a(15) = 42; //a now has length 15
```

After line 2 in the above program is executed, the vector `a` has a length of 15. Elements created when the array expands are assigned a value of zero. Thus, elements `a(11)` to `a(14)` get the value 0, while `a(15)` gets the value 42.

2.3.4 Libraries

MATLAB has a wide variety of toolsets that provide users with domain specific functionality. For example, the communication toolbox provides functionality required for the design and simulation of communication systems and the image processing toolbox provides APIs to several frequently used image processing functions. However, most of these toolboxes are closed source.

2.4 Supported MATLAB Subset

As MATLAB has grown into a diverse and vast language over the years, the compiler described in this thesis supports only a representative subset of MATLAB. A brief description of the subset of MATLAB supported and a set of other assumptions made by our compiler implementation are presented below.

1. MATLAB supports variables with primitive types `logical`, `int`, `real`, `complex` and `string`. It is also possible to construct arrays of these types with any number of dimensions. Currently, our compiler supports all primitive types except `complex` and `string`. Further, arrays are restricted to a maximum of three dimensions.

2. MATLAB supports indexing with multi-dimensional arrays. However, our implementation currently only supports indexing with single dimensional arrays.

3. In MATLAB, it is possible to change the size of arrays by assigning to elements past their end. We currently do not support indexing past the end of arrays. Further, in this thesis, we refer to assignments to arrays through indexed expressions (For example, `a(i)`) as *indexed assignments* or *partial assignments*.

4. We assume that the MATLAB program to be compiled is a single script without any calls to user-defined or toolset functions. Support for user defined functions can be added by extending the frontend of the compiler. Also, anonymous functions and function handles are not currently supported.

5. In general, types and shapes (array sizes) of MATLAB variables are not known until run-time. Our compiler currently relies on a simple data flow analysis to extract sizes and types of program variables. It also relies on programmer input when types cannot be determined. We intend to extend our type system to support symbolic type inferencing in future [37]. Ultimately, we envision that the techniques described in this thesis will be used in both compile time and run-time systems.

Chapter 3

MEGHA : Design and Implementation

In this chapter, we first describe the high-level structure of the compiler and then discuss each stage of the compiler in more detail, providing a detailed description of its design and implementation. The input to the compiler is a MATLAB script and the output is a combination of C++ and CUDA code. As already mentioned, the compiler has to identify parts of the input program that can be run efficiently on the GPU and then has to insert the necessary data transfers between CPU and GPU memory. A high level overview of our compiler, which can accomplish these tasks, is shown in Figure 3.1.

The frontend of the compiler simplifies the input code and constructs a Static Single Assignment (SSA) [21] intermediate representation (IR) from it. The backend then performs various analyses and optimizations before generating the final output code. The focus of the work described in this thesis is the backend of the compiler. Even though the frontend of a MATLAB compiler provides several interesting areas for research, we do not investigate these. Past work [22, 38, 37, 19, 4] has focussed on the design of a frontend for a MATLAB compiler.

3.1 The Frontend

The frontend first simplifies the input MATLAB code by converting it to a form similar to three address code called *single operator form* [37, 38]. Section 3.1.1 provides more details and examples of the conversion to single operator form. The simplified code is converted to a

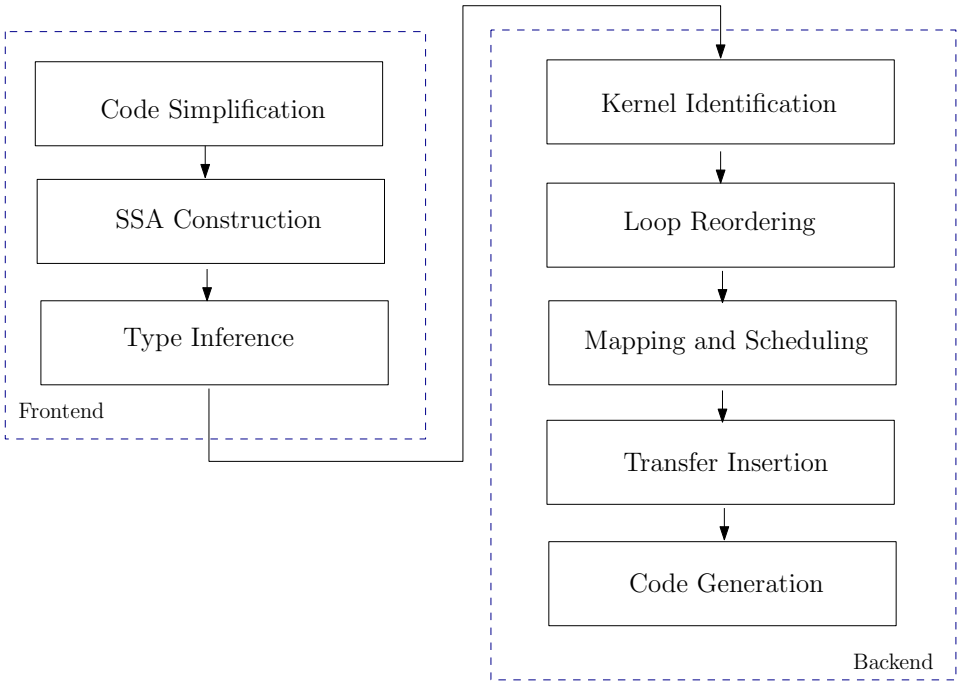


Figure 3.1: Compiler Overview

static single assignment (SSA) form [21]. Section 3.1.2 provides details about the SSA form and its construction. Then a type inference pass is needed because MATLAB programs are dynamically typed [48]. This is described in Section 3.1.3. The output of the frontend of the compiler is a *Control Flow Graph* (CFG). This is passed to the backend of the compiler.

3.1.1 Code Simplification

The compiler first simplifies the input source code by breaking complex expressions into simpler ones. The output of this pass is a single operator intermediate form; i.e., each statement in the resulting IR has at most one operator on the right hand side of assignments. Thus, expressions are broken into single operator form by introducing temporaries. For example,

```
1    x = (a + b) - c;
```

is converted to

```
1    tempVar1 = a + b;
2    x = tempVar1 - c;
```

Array indexing is also treated as an operator. First each indexing expression is simplified if needed. Then a simplified array reference is emitted. For example,

```
x = a(2*i);
```

is converted to

```
tempVar1 = 2*i;
x = a(tempVar1);
```

We decided to simplify the input code, at the cost of inserting temporaries, for the following reasons.

1. Simplifying the code simplifies the implementation of analysis passes like type inference and live variable analysis.

2. The backend of the compiler constructs entities called *kernels* which are potentially run on the GPU (Section 3.2.1). Converting the input code into single operator form gives the backend finer control when it is constructing kernels.

The single operator form is conceptually similar to *three-address code* [3]. However, there are a few differences. Firstly, index expressions may involve more than just 3 variables (for example, `a(i, j, k) = 2`). We do not break these down further because we want the IR to be valid MATLAB code. Secondly, MATLAB supports assigning to multiple variables in the same statement (multi-assignments). This construct also cannot be represented in traditional three address code.

Also, it is not always possible to completely simplify indexing expressions. This is the case when indexing expressions contain *magic ends* (for example, `a(2:end)` which refers to all elements of array `a` starting from the second element) and *magic colons* (for example, `a(:)` which refers to all elements of array `a`). These are handled using a set of semantics preserving transformations described below in Section 3.1.1.1.

3.1.1.1 Semantics Preserving Transformations

MATLAB supports "context-sensitive" indexing of arrays. It is possible to write operators and keywords which mean different things depending on which array's index they are used as and

in which dimension they are used. Users can specify all elements along a dimension by simply using a ":" in the index corresponding to that dimension. For example, to set all elements of a vector `a` to 42, one could write:

```
1  a( : ) = 42
```

One can also use the keyword `end` inside an array index expression in MATLAB to refer to the last element of the array along that dimension. For example, if `a` is a vector,

```
1  a(4:end) = 42;
```

assigns 42 to all elements of vector `a` starting from element 4. Our compiler removes these context-sensitive operators by replacing them with the indices they imply : e.g., `a(:)` is replaced by `a(1:length(a))` and `a(2:end)` is replaced by `a(2:length(a))`.

Some transformations are also needed on for-loops in user code to preserve MATLAB semantics. In MATLAB, assignments to the loop index variable or to the loop bounds inside the loop body do not have any effect on the iteration count of the loop. For example, the loop in the following program runs for ten iterations even though `i` gets incremented to a value larger than `n` after the first iteration.

```
1  n = 10;
2  for i=1:n
3    % ...
4    i = i + 10;
5    % ...
6  endfor
```

Our compiler preserves these semantics by renaming the loop index (and bounds). A statement to copy the new loop index into the old loop index is inserted at the head of the loop. Therefore, the above code would be transformed to the following equivalent code.

```
1  n = 10;
2  for tempIndex=1:n
3    i = tempIndex;
4    % ...
5    i = i + 10;
6    % ...
7  endfor
```

3.1.2 Static Single Assignment Construction

In the Static Single Assignment (SSA) form [21], each variable in the program is defined (assigned to) exactly once in the static representation of the program[1]. An input program is converted into the SSA form by renaming the left hand side of each assignment. For example,

```
1  n = 42;
2  n = n + 1;
3  x = n;
```

is converted to

```
1  n1 = 42;
2  n2 = n1 + 1;
3  x = n2;
```

For cases where a use of a variable has multiple reaching definitions [3], the ϕ function is defined [21]. The ϕ function takes multiple arguments, each one representing one reaching definition and returns the definition that actually reaches the use dynamically. For example,

```
1  n = 42;
2  if (condition)
3    n = n + 1;
4  else
5    n = n - 1;
6  x = n;
```

is converted to

```
1  n1 = 42;
2  if (condition)
3    n2 = n1 + 1;
4  else
5    n3 = n1 - 1;
6  n4 = φ(n2, n3);
7  x1 = n4;
```

[1]Of course, variables may be assigned to multiple times when the program is run as assignments may be executed multiple times.

The ϕ function on line 6 is assumed to correctly return either n_2 or n_3 depending on which control flow path the program takes.

The SSA form is constructed in two steps by modifying the control flow graph (CFG) of the program [3] as described by Cytron et al [21]. The control flow graph our compiler constructs is slightly different from traditional CFGs. Each node of the graph is a MATLAB basic block rather than a traditional basic block.

Definition 1 *A **MATLAB Basic Block** is a set of IR statements that has a single entry point and a single exit point, with statements possibly containing complex MATLAB operators.*

A MATLAB basic block differs from a traditional basic block [3] because of the control flow implied by high level MATLAB operators. For example, a statement that performs matrix addition implies the presence of two loops (one for each dimension of the matrix). To keep the program's control flow graph at a reasonably high-level, we do not consider these loops as part of the program's control flow and therefore define the MATLAB basic block as in Definition 1. However, user-written `for` loops, `while` loops etc are considered to be part of the program's control flow.

The construction of the SSA form follows the steps mentioned below for each program variable. The procedure is similar to that described by Cytron et al [21].

1. The first step inserts ϕ statements at the beginning of every MATLAB basic block that is in the *dominance frontier* [21] of a MATLAB basic block containing a definition of the variable under consideration. Each inserted ϕ statement is considered a definition and the insertion step is repeated. The process is continued until no more ϕ statements are added.

2. After ϕ statements have been inserted, each instance of the variable on the left hand side of an assignment is renamed. Each use of this definition is modified to refer to the correct definition. We are guaranteed that each use of the variable has a single reaching definition because of the insertion of ϕ statements.

We choose the static single assignment form (SSA) as our intermediate form due to the following reasons.

1. The dynamic nature of MATLAB means that each assignment can potentially change the type of a variable. The SSA form mitigates this difficulty by ensuring that each variable is assigned to exactly once. This means that each SSA variable has exactly one type.

2. Converting to SSA form exposes more clearly the task-level parallelism present in the input program.

However, since the variables in the program are possibly arrays, it is important to optimize the insertion of ϕ nodes. One optimization we perform is that our compiler inserts ϕ nodes for a variable at a point only if it is live at that point. The set of live variables at each program point is computed using the standard dataflow analysis [3].

For example, consider the following code.

```
1  n = 42;
2  x = 1;
3  while (condition)
4    x = n + 1;
5    n = x + 1;
6  endwhile
```

The SSA representation of the above code is shown below.

```
1  n1 = 42;
2  x1 = 1;
3  while (condition)
4    n2 = φ(n1, n3);
5    x2 = n2 + 1;
6    n3 = x2 + 1;
7  endwhile
```

Here, there is no ϕ for the variable `x` because it is not live at the start of the `while` loop. However, a ϕ is needed for the variable `n`.

Further, assignments to array variables through indexed expressions are not renamed; in other words, values are assigned in-place when assigned through an indexed expression. This is because we assume that indexed assignments cannot change the size of the array that is being assigned to.

```
1  n = 100;
2  A1 = ones(n, n);
3  B1 = rand(n, n);
4  A2 = B1 + B1;
5  A2(10) = 42;
```

Here, the assignments to `A` in lines 2 and 4 are renamed. However, the *partial assignment* in line 5 is not renamed. All assignments through indexed expressions are treated as partial assignments. It is difficult to determine whether or not an indexed assignment assigns to the whole array. Our compiler currently does not perform any analysis to determine whether or not an assignment is a partial assignment, i.e all indexed assignments are treated as partial assignments.

3.1.3 Type and Shape Inference

Since MATLAB is a dynamically typed language, type and shape information of each variable needs to be inferred before a MATLAB script can be compiled to a statically typed language like C. Our definition of the type of a variable is based on the definition used by De Rose et al. [22]. The type of a variable is a tuple consisting of the following elements :

1. *Intrinsic Type*, which can be `boolean`, `integer` or `real`;
2. *Shape*, which can be `scalar`, `vector`, `matrix` or `3D-array`; and
3. *Size*, which indicates the size of each dimension of the variable.

Our compiler currently tries to infer base types of variables and shapes of arrays based on program constants and the semantics of built-in MATLAB functions and operators by performing a forward data flow analysis that is similar to what was used in FALCON [22] and MaJIC [4]. For example, consider the following IR code segment.

```
1  n = 100;
2  A = rand(n, n);
```

Here, the variable `n` is assigned the constant value 100. This value is the size parameter to the `rand` function used to initialize the value of `A`. Our compiler can therefore infer that the type of

`A` is a `real matrix` of size 100×100. Our compiler performs constant propagation so that compile time constants can be used to infer the sizes of as many program variables as possible. It also propagates shapes through MATLAB operators. For example, the compiler knows that the size of the matrix produced as a result of adding two matrices is the same as the sizes of the two matrices that were added. It uses similar rules for operations such as multiplication and transpose.

It is also possible for the user to specify types of variables in the input MATLAB script via annotations. The user must do this for variables whose size and type cannot be inferred by our compiler. Currently, our compiler does not implement symbolic type inference and symbolic type equality checking [37]. We leave this for future work.

3.2 Backend

The backend of the compiler performs kernel identification, mapping, scheduling and data transfer insertion. Kernel identification identifies sets of statements, called *kernels*[2]. It also transforms kernels into loop nests from which the code generator can generate efficient lower level code. Each kernel is treated as a single entity by the mapping and scheduling phase. The mapping and scheduling phase assigns a processor (CPU or GPU) to each identified kernel and also decides when it should be executed. Kernel identification and mapping are performed per MATLAB basic block (refer to Definition 1). The global transfer insertion phase then performs a global data flow analysis and inserts the required data transfers to correctly satisfy dependences across MATLAB basic blocks. The various phases of the backend are shown in Figure 3.1.

The following sections describe each step performed by the backend of the compiler in more detail.

[2]We use the term "kernel" to refer to either the smallest unit of data-parallel work in a set of IR statements or a set of GPU friendly IR statements. Which is meant will be clear from context.

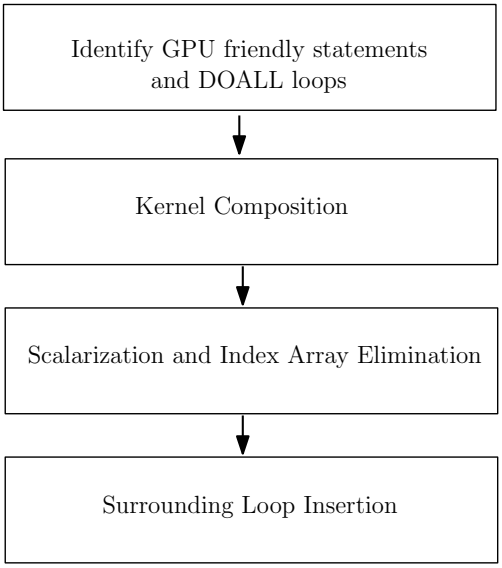


Figure 3.2: Steps involved in kernel identification

3.2.1 Kernel Identification

The purpose of the kernel identification process is to identify the groups of IR statements on which mapping and scheduling decisions can be made. We call these entities *kernels*. They are sets of IR statements that can be efficiently run on the GPU. We define a kernel more precisely as follows.

Definition 2 *A **Kernel** is a set of IR statements that can be viewed as a* single perfect loop nest *each of whose loops is parallel (DOALL).*

Therefore, each kernel can be thought of as having two parts – a set of surrounding, parallel, perfectly nested loops and a set of statements that actually perform the computation. For example, consider the following kernel in which `A`, `B` and `C` are matrices of size 100×100.

```
1  A = B + C;
```

This kernel can equivalently be written as

```
1  for i = 1:100
2    for j = 1:100
3      A(i, j) = B(i, j) + C(i, j);
4    endfor
5  endfor
```

where the surrounding for loops are all *parallel loops*. (We refer to loops whose iterations can be executed in parallel as parallel loops).

The kernel identification process consists of four steps as shown in Figure 3.2. They are as follows.

1. Identify IR statements that are data parallel and DOALL `for` loops as "GPU friendly" statements.

2. *Kernel Composition* identifies sets of statements that can be grouped together in a common kernel.

3. *Scalarization* and *Index Array elimination* eliminate arrays that are not needed because of the way statements are grouped together.

4. *Surrounding Loop Insertion* transforms kernels into loop nests consisting of parallel loops.

3.2.1.1 Identifying GPU Friendly Statements

Our compiler currently tags statements that perform element-wise operations such as matrix addition or other data parallel operations like matrix multiplication as "GPU friendly" statements. It also identifies perfect nests of DOALL (parallel) loops as "GPU friendly".

Definition 3 *Statements that perform element-wise operations, matrix multiplication or matrix transpose are termed* ***GPU friendly*** *statements. Perfect nests of parallel loops are also GPU friendly.*

In addition to identifying data parallel MATLAB operators, our compiler can identify `for` loops with data parallelism as being GPU friendly using a straight forward analysis. To identify DOALL loops, our compiler uses a standard dependence analysis based on the Banerjee test [2, 9]. The implementation of the dependence analyzer closely follows the description in the book by Allen and Kennedy [2]. It should be noted that this step just identifies statements that *could* potentially run on the GPU; however, it does not make any decisions on what should actually be run on the GPU.

3.2.1.2 Kernel Composition

```
1:  A = (B + C) + (A + C);
2:  C = A * C;
```

(a) MATLAB Statements

```
1:  tempVar0 = B + C;
2:  tempVar1 = A + C;
3:  A_1 = tempVar0 + tempVar1;
4:  C_1 = A_1 * C;
```

(b) IR Representation

Figure 3.3: A simple MATLAB program and its IR representation. `A`, `B` and `C` are 100×100 matrices.

Once GPU friendly statements have been identified, the kernel composition step decides the granularity at which mapping decisions are to be made. It does this by grouping IR statements together. (Currently, DOALL loops identified in the program are treated as separate kernels and are not merged with other statements). Kernel Composition is performed as it is inefficient to make mapping and scheduling decisions at the level of IR statements as there may be too many of them. Further, kernel composition could also lead to a significant decrease in the memory requirement of the compiled program. To understand why, consider the example in Figure 3.3(a). Code simplification generates a considerable number of temporaries that are used only once after they are defined as can be seen in Figure 3.3(b). If mapping decisions are made per IR statement, the definition and use of these temporaries could be mapped to different processors necessitating the storage of these temporaries, which may be arrays, in memory. In the example in Figure 3.3(b), three temporaries, each of which is a matrix, have been generated by the simplification process. In the worst case, when all these temporaries need to be stored in memory, the memory requirement of the program increases significantly. However, if statements 1, 2 and 3 are grouped into the same kernel, it is possible to replace the arrays `tempVar0` and `tempVar1` with scalars. When this kernel is rewritten as a loop nest, these variables are only live within one iteration of the loop nest. Hence they can be replaced by scalars as in the following code.

```
1  for i=1:100
2    for j=1:100
3      tempVar0_s = B(i, j) + C(i, j);
4      tempVar1_s = A(i, j) + C(i, j);
5      A_1(i, j) = tempVar0_s + tempVar1_s;
6    endfor
7  endfor
```

The above problem is even more exaggerated in real benchmarks. To give a specific example, in the benchmark `fdtd` [36], we observed that the worst case memory requirement increase can be as high as 30×. Combining statements into kernels is also important because the mapping and scheduling phase would then be able to solve a scheduling problem with a fewer number of nodes.

We refer to this process of reducing arrays to scalar variables as *Scalarization*. However, the newly created scalars are likely to be saved in registers thereby increasing the register utilization.

Legality of Kernel Composition:

Two statements can be combined into the same kernel only if the following conditions hold.

1. Both statements perform element-wise operations.
2. Both statements have identical iteration spaces. (By iteration space of a statement, we mean the iteration space of the parallel loops surrounding the statement.)
3. Combining the statements will not result in cyclic dependencies among kernels.
4. The loops surrounding the kernel created by combining the two statements should be parallel[3].

In the example in Figure 3.3(b), it is legal to group the first three statements, in the simplified IR, into the same kernel. However, it is illegal to put statement 4 into the same group as matrix multiplication is not an element-wise operation and requires elements at other points in the iteration space to have been computed. Thus, combining the addition and the multiplication

[3]Parallel loops are loops whose iterations can all be run in parallel.

statements introduces cross iteration dependencies and therefore the surrounding loops of the resulting group are no longer parallel. Even though condition 1 is subsumed by condition 4, it is specified separately as it is easier to check for violations of condition 1.

Condition 4 is similar to the condition for legal loop fusion [40] but is stricter. For loop fusion to be legal, it is only required that directions of dependencies between statements in the loops being fused are not reversed because of the fusion. However, for kernel composition to be legal, we require that the surrounding loops still be parallel loops after the statements are combined as our objective is to map identified kernels for parallel execution on the GPU.

Combining statements into kernels has many benefits. Compiler generated temporary arrays and user arrays can be converted to scalars. Locality between the combined statements can be exploited. Lastly, combining statements into kernels also results in a reduction in the CUDA kernel invocation overhead at run-time. However, forming larger kernels can also increase register usage and the memory footprints of kernels. These factors are extremely important if the kernel is to be run on the GPU. Larger kernels may also require more variables to be transferred to and from the GPU. This may lead to decreased opportunities for overlapping computation and data transfer. Thus there are trade offs in kernel composition.

We model the kernel composition problem as a constrained clustering problem modelling the costs and benefits described above. It is loosely related to the parameterized model for loop fusion [68]. However, compared to the model for loop fusion, our formulation additionally models memory utilization of kernels. Register utilization is also modelled by our formulation. The details of the construction and a heuristic algorithm to compute a solution are described in the following subsections.

Graph Formulation

The problem of composing statements into kernels is that of clustering the nodes of an augmented data flow graph $G = (V, E)$ into clusters. A cluster is a subset of nodes of the graph G. The graph G is defined as follows.

1. Each node $n \in V$ represents an IR statement in the MATLAB basic block under consideration. The statement represented by $n \in V$ is denoted by stm_n.

2. The set of edges, E has two types of edges.

   - *Dependence edges* are directed edges between nodes whose statements are data dependent. An edge (n_1, n_2) implies that n_1 must complete execution before n_2 can run. All types of dependences (true, anti and output) are represented uniformly.
   - A *Fusion preventing edge* connects a pair of nodes that cannot be merged as one of the legality conditions for kernel composition is violated. For example, such an edge would exist between the vertices for statements 3 and 4 in Figure 3.3(b) as condition 1 would be violated. There are also fusion preventing edges between nodes representing GPU friendly statements and other statements (Definition 3).

The kernel composition problem is that of clustering the set of nodes V into a set of clusters $\{C_1, C_2, ...C_n\}$. All the C_i's are disjoint and their union is V. An edge $e = (n_1, n_2)$ is said to be in a cluster C_i if both n_1 and n_2 are in C_i.

To model register utilization, we consider the original register usage of each statement in the MATLAB basic block and the increase in register usage due to scalarization of arrays. First, the register requirement of each statement stm_n, for each $n \in V$, denoted by reg_n, is estimated as the number of scalars used in stm_n. Secondly, additional registers required for variables that can be scalarized need to be taken into account. For this, we need to differentiate between *scalarizable* and *non-scalarizable* variables.

Definition 4 ***Scalarizable variables*** *are those that can be scalarized by combining exactly two IR statements into the same kernel, i.e., variables with one definition and exactly one use. All other variables are* ***non-scalarizable*** *variables.*

To model the increase in register utilization when two statements are combined, we define the weight reg_e on each edge $e = (n_1, n_2) \in E$. The value of this quantity is the number of scalarizable variables that can be scalarized by combining stm_{n_1} and stm_{n_2}. For example, it is 1 for the edge between the nodes for statements 1 and 3 in Figure 3.3(b) since `tempVar0` can be eliminated by combining these statements. Now, when a cluster is formed, the register utilization of the kernel represented by the cluster is the sum of the reg values of all nodes and

edges in the cluster. Therefore, to limit the register utilization of each cluster, we introduce the following constraint for each cluster C_i.

$$\sum_{e \in C_i} reg_e + \sum_{n \in C_i} reg_n \leq R_{max} \tag{3.2.1}$$

Since we do not model the overlap in the lifetimes of variables, Equation 3.2.1 is an approximation of the actual register requirement of the kernel.

To model memory utilization, the distinction between scalarizable variables and non-scalarizable variables is needed again. For scalarizable variables, such as `tempVar0`, the memory utilization becomes zero when its defining node and use node are combined (statements 1 and 3 in this case). However, for non-scalarizable variables, the memory usage is the sum of the sizes of all such variables used by statements in the cluster.

To model the memory utilization due to scalarizable variables, we use weights on both edges and nodes. For a node n, we define mem_n as the sum of sizes of all scalarizable variables in stm_n. Therefore mem_n, for the node representing statement 1 is $size(tempVar0)$. For an edge e between n_1 and n_2, if there is a variable $scal\_var$ that can be scalarized by combining stm_{n_1} and stm_{n_2}, we define $mem_e = -2.size(scal\_var)$. Therefore, the memory utilization of a scalarizable variable reduces to zero if both its definition and use are in the same cluster as the weights on the nodes are cancelled out by the weight of the edge between them. The total memory usage of the kernel represented by the cluster C_i, due to scalarizable variables is therefore given by:

$$\sum_{n \in C_i} mem_n + \sum_{e \in C_i} mem_e = M_{scal,i} \tag{3.2.2}$$

For non-scalarizable variables, we use a bit vector to model the memory utilization. For each node $n \in V$, and each non-scalarizable variable var, we define $used_n(var)$ to be 1 if n references var. For example, the node m, representing statement 1 in Figure 3.3(b), has $used_m(B) = 1$ and $used_m(C) = 1$. A variable var is referenced in a cluster if any of the nodes in the cluster has $used(var)$ as 1. The memory utilization of a cluster due to non-scalarizable variables is the sum of sizes of all such variables used in the cluster. The memory utilization due to non-scalarizable variables is given by:

$$\sum_{var \in Vars} (\bigvee_{n \in C_i} used_n(var)).size(var) = M_{nonscal,i} \tag{3.2.3}$$

where $Vars$ is the set of all non-scalarizable variables in the program.

To limit the total memory utilization of each kernel, we introduce the following constraint.

$$M_{scal,i} + M_{nonscal,i} \leq M_{max} \tag{3.2.4}$$

Additionally, for all fusion preventing edges $e \in E$, e must not be in any cluster C_i. Also, when nodes are grouped together as kernels, the graph must remain acyclic.

For each edge $e = (n_1, n_2) \in E$, $benefit_e$, quantifies the benefit of combining the statements stm_{n_1} and stm_{n_2} into the same kernel. This is defined as

$$benefit_e = \alpha Mem(n_1, n_2) + \beta Loc(n_1, n_2) + \gamma$$

where, $Mem(n_1, n_2)$ is the sum of sizes of array variables that can be eliminated by combining stm_{n_1} and stm_{n_2}, $Loc(n_1, n_2)$ quantifies the locality between the two statements. Currently, our compiler estimates this as the number of pairs of overlapping array references in the two statements. If two statements being considered for composition have a common array reference, then this can be read into shared memory and reused. The term $Loc(n_1, n_2)$ is meant to model this. γ represents the reduction in the kernel call overhead. α and β are parameters.

The objective of kernel composition is to maximize the total $benefit$:

$$benefit = \sum_{C_i} \sum_{e \in C_i} benefit_e$$

subject to the above constraints. The above problem can be shown to be NP-hard by a reduction from the loop fusion problem defined by Singhai and McKinley [68]. We therefore use a heuristic algorithm to solve the kernel composition problem.

Kernel Composition Heuristic

Our heuristic kernel clustering method uses a k-level look ahead to merge nodes of the graph G into clusters. It computes the benefits and memory requirement for collapsing all possible legal sets of k edges starting at node n by exhaustively searching through all possibilities. It then collapses the set of k edges that results in the maximum benefit. By collapsing an edge, we mean combining its source and destination nodes. Combining a set of nodes $N = n_1, n_2, ..., n_k$

results in a single node η. All edges (n_i, n_j) such that $i, j \in [1, k]$ are absorbed. Any edge from a node in N to a node x not in N results in an edge from η to x with appropriate weight. Similarly, an edge from x to n_i is replaced by an edge from x to η. For small values of k($k = 3$ or $k = 4$), this proposed k-level look ahead achieves reasonably good results without a significant increase in compile time.

The details of the clustering algorithm are presented in Algorithm 1. The function $legalKDepthMerges$ returns multiple sets of (up to) k edges that can be *legally* collapsed. This function makes sure that only legal clusters are formed; that is, clusters that are formed do not have fusion preventing edges. It also ensures that no cycles are formed in the graph and that the constraints listed earlier are not violated. The function $maxBenefit$ returns the set of edges with the maximum benefit and $collapseEdges$ collapses all edges in a set of edges and adds all combined nodes to C_i.

Algorithm 1 Kernel Composition Heuristic

```
1:  procedure KernelComposition(G = (V, E))
2:  nodes ← V
3:  clusters ← φ
4:  while nodes ≠ φ do
5:      n ← node in nodes that uses minimum memory
6:      C_i ← n
7:      nodes ← nodes − {n}
8:      while true do
9:          T ← legalKDepthMerges(C_i, k)
10:         if T ≠ φ then
11:             bestSet ← maxBenefit(T)
12:             collapseEdges(C_i, bestSet)
13:             for all e = (n_1, n_2) ∈ bestSet do
14:                 nodes ← nodes − {n_1, n_2}
15:             end for
16:         else
17:             break
18:         end if
19:     end while
20:     clusters ← clusters ∪ C_i
21: end while
22: end procedure
```

Kernel Composition Example

In this section, we describe the process of kernel composition in the context of the example shown in Figure 3.3(b). Figure 3.4 shows the augmented dataflow graph for the example. The graph has four nodes, one for each of the statements. There are dependence edges between nodes 1 and 3, nodes 2 and 3 and nodes 3 and 4. Also, there is a fusion preventing edges between the nodes 3 and 4 (shown as a dotted edge in the figure). Figure 3.4 also shows the reg and mem values for each node and edge in the graph. For example, node 1 has $mem_1 = 10000$ because it uses the scalarizable variable `tempVar0` (all variables are 100×100 matrices) and reg_1 is 0 because it has no scalar variables. The edge e between nodes 1 and 3 has $mem_e = -20000$ and $reg_e = 1$ because `tempVar0` can be scalarized by collapsing this edge. It also has $used_1(B) = used_1(C) = 1$. $used_1$ is zero for all other variables. In this example, the variables `A_1`, `A`, `B`, `C` and `C_1` are considered to be non-scalarizable variables.

Figure 3.5 shows the $benefit$ value for each edge. We use $\alpha = 1$, $\beta = 0$ and $\gamma = 100$ to compute these values. Currently, our compiler does not use shared memory to take advantage of locality between combined statements. We therefore use $\beta = 0$. The figure also shows the clustering for the dataflow graph and the memory usage, register usage and benefit for each cluster. As a result of the clustering process, nodes 1, 2 and 3 belong to the same cluster while node 4 is in a cluster of its own because of the fusion preventing edge between nodes 3 and 4. The memory usage of the first cluster is the sum of the sizes of `A_1`, `A`, `B` and `C` which is 40000. The sizes of `tempVar0` and `tempVar1` cancel out while adding mem values of edges and nodes in the cluster because they are scalarized. The register utilization is estimated as 2 because these two variables are scalarized. The $benefit$ is the sum of the $benefit$s of edges (1, 3) and (2, 3). Similarly, the second cluster's memory usage is 30000 because it uses `A_1`, `A` and `C`. However, this cluster has $benefit = 0$ because no edges are collapsed in the forming of this cluster.

3.2.1.3 Identifying In and Out Variables

Once kernels have been identified, we need to compute the set of variables that are required for the execution of the kernel and the set of variables that are modified by the kernel and are needed

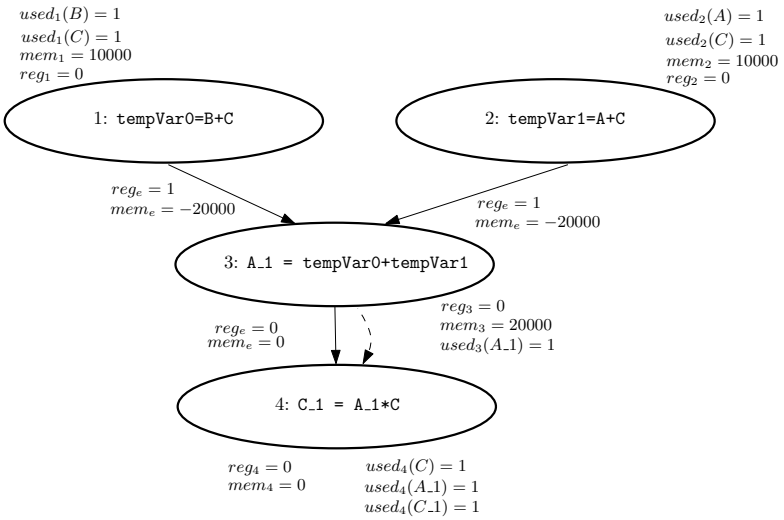


Figure 3.4: Augmented dataflow graph for the example in Figure 3.3(b)

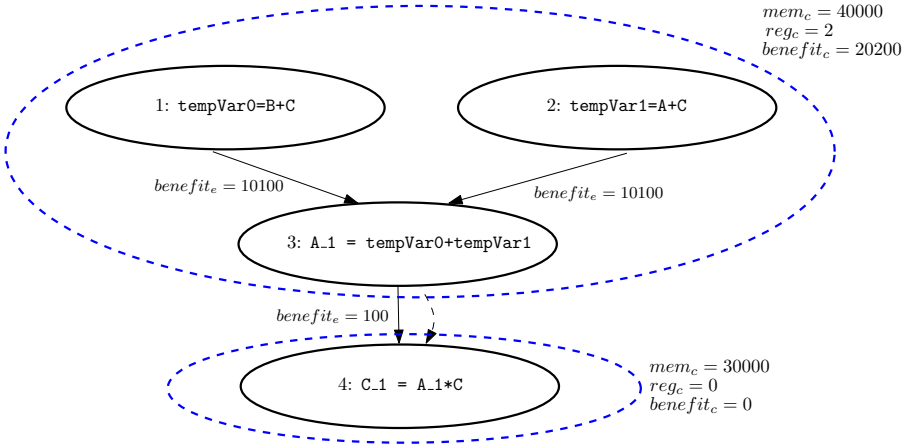


Figure 3.5: Clustering of the augmented dataflow graph for the example in Figure 3.3(b)

by future computations. To do this we first perform a live-variable analysis [3] on the SSA IR. A variable is an "in-variable" of a given kernel if it is live on entry to the kernel and is accessed (either read or partially written[4]) within the kernel. We define a variable as an "out-variable" if the variable is modified inside the kernel and is live at the exit of the kernel. In-variables of a kernel are the variables that need to be present in GPU memory before the kernel can be executed on the GPU. Out-variables are those that the kernel will produce in GPU memory if the kernel is executed on the GPU.

[4]As the unmodified part of the array may be used for further computation, the array is considered to be read initially before it is written.

3.2.1.4 Array Elimination

As discussed in Section 3.2.1, arrays that become unnecessary after kernel composition need to be eliminated. Replacing these arrays with scalar variables not only reduces memory requirements but also eliminates memory accesses that are expensive, especially if they are serviced out of GPU device memory. To remove array variables, we use the two transformations, scalarization and index array elimination, described below.

Scalarization

After in-variables and out-variables of each kernel have been identified, it is possible to replace an array that is neither an in-variable nor an out-variable by a scalar as shown in the earlier example. This is possible because it is guaranteed that no subsequent statement depends on writes to the array within this kernel and that this kernel does not depend on the prior value of this array. Also, as each iteration of the kernel's loop nest is independent, there can be no cross iteration dependences due to such variables. We refer to this process as *scalarization*.

Index Array Elimination

In MATLAB programs, arrays (especially those defined using the colon operator) are frequently defined and used either for indexing or in element-wise computations. It is possible to replace some references to these arrays with scalars that are computed inside the kernel in which the original array was used. Any array representable as a function $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ that maps the index of an array element to the element value and used to index one or more dimensions of another array is called an *index array*. A total reference to such an array in a kernel can be replaced by a scalar that takes the value $f(i)$ in the i^{th} iteration of the surrounding loop. Consider the following example, in which a subset of the 3D array `Ey` is copied into `tempVar1`.

```
1  tempVar0 = Nz + 1;
2  tempVar1 = 2:tempVar0;
3  tempVar2 = Ey(:, :, tempVar1);
```

Here, `tempVar1` is an index array with the function f given by $f(i) = 2 + (i - 1)$. It can be eliminated by replacing it with a scalar while adding surrounding loops as shown below (details

in Section 3.2.1.5).

```
1  tempVar0 = Nz + 1;
2  for i = 1:Sx
3    for j = 1:Sy
4      for k = 1:Sz
5        indexVar1 = 2 + (k-1);
6        tempVar2(i, j, k) = Ey(i, j, indexVar1);
7      end
8    end
9  end
```

`Sx`,`Sy` and `Sz` represent the size of the iteration space of the original assignment in the first, second and third dimension respectively. We refer to this transformation as *index array elimination*.

Currently, our compiler identifies arrays defined using the colon operator as index arrays. Inferring the function f for these arrays is straight forward. Consider the array `A = l:s:u`. For the array `A`, $f(i) = l + s(i-1)$. Arrays defined using functions like `linspace`[5] and in user loops could also be identified as index arrays. We intend to implement these extensions in the future.

3.2.1.5 Surrounding Loop Insertion

Before code can be generated, identified kernels need to be transformed into a form more suitable for code generation. The compiler achieves this by adding loops around kernels and introducing the necessary indexing.

For kernels that contain element-wise operations, the required number of surrounding loops are inserted and array references are changed as described next. Consider the example in Section 3.2.1.4. Introducing indexing is simple for complete references like `tempVar2`. The indices are just the loop variables of the surrounding loops. However, the case for partial references (Eg: `Ey(:, :, tempVar1)`) is slightly more involved. The k^{th} *non-scalar* subscript in the partial reference is transformed according to the following rules.

[5] `linspace` is a MATLAB function that generates linearly spaced vectors. `linspace(x, y, n)` generates `n` linearly spaced numbers from `x` to `y`. For example, `linspace(1, 9, 3)` returns the array `[1 5 9]`.

1. If it is a magic colon, replace it by i_k that is the loop variable of the k^{th} surrounding loop. For example, the first and second subscripts in the reference to `Ey` in the above example are changed to `i` and `j`.

2. If the subscript is an index array, replace the array by a scalar variable initialized with the value $f(i_k)$, where f is the function described in Section 3.2.1.4. This rule is used to change the third indexing expression in the reference to `Ey` from `tempVar2` to `indexVar1`.

3. Otherwise, if the k^{th} non-scalar subscript is x, then replace it by $x(i_k)$.

Once surrounding loops are inserted, the body of the loop nest represents exactly the CUDA code that would need to be generated if the kernel under consideration gets mapped to the GPU.

Kernels that contain more complex operators, i.e., operations other than element-wise operations, are treated as special cases when they are transformed into loop nests. The compiler keeps the original high-level IR as this is useful during code generation (for example to map a matrix multiplication kernel to a BLAS library call).

3.2.2 Parallel Loop Reordering

After the steps described in the previous section have been performed, the IR contains a set of kernels and their surrounding parallel loops. The in-variables and out-variables of each kernel are also known. When these kernels are ultimately executed, the way in which the iteration space of the surrounding loops is traversed can have a significant impact on performance regardless of whether the kernel is mapped to the CPU or the GPU. Further, for efficient execution, the iteration space may need to be traversed differently depending on whether the kernel is mapped to the CPU or the GPU. Since the surrounding loops are parallel, the iteration space of these loops can be traversed in any order. The loop reordering transform decides how the iteration space must be traversed to maximize performance.

For a kernel that is executed on the CPU, locality in the inner most loop is important from a cache performance viewpoint. Currently, in our implementation, three dimensional arrays

are stored as stacked row major arrays[6]. An element `a(i1, i2, i3)` is stored in location $i3 * d_1 * d_2 + i_1 * d_2 + i_2$ where d_1, d_2 and d_3 are the sizes of the first, second and third dimension respectively. For example, consider the code listed in Section 3.2.1.4. There are three surrounding loops with indices `i`, `j` and `k`. Therefore, `tempVar2(i, j, k)` and `tempVar2(i, j+1, k)` are in successive memory locations. Hence, the two indexed references to `tempVar2` and `Ey` will access consecutive memory locations for consecutive values of `j`. This implies that having the `j` loop as the innermost loop will maximize locality.

GPU memory optimizations are however very different from memory optimizations implemented by the CPU. When all threads in a warp access a contiguous region of memory, the GPU is able to coalesce [51] these multiple memory requests into a single memory request thereby saving memory bandwidth and reducing the overall latency of the memory access. If a kernel is assigned to the GPU, the code generator maps the index of the outermost loop to the x dimension of the thread block. The GPU groups neighboring threads in the x dimension into the same warp. Therefore, in the GPU case, the loop with the maximum locality needs to be the outermost loop (which in the above example is the `j` loop).

At this stage, since it is not known whether a kernel will be mapped to the CPU or GPU, our compiler computes loop orders for both of them. Also, both versions are needed during the profiling stage. Currently, a simple heuristic is used to compute the loop ordering. The number of consecutive memory references due to all variables or arrays in successive iterations of each loop (with all other loop indices staying constant) is computed. The loops are then sorted based on this number. The loop that causes accesses to consecutive memory locations for the most references in the kernel is treated as the loop with the maximum locality.

For the CPU loop order, the loops are sorted in increasing order of locality. (The first loop in the sorted sequence is the outermost loop and the last loop is the inner most loop.) For the example above, the `j` loop has the most locality since both indexed references access successive memory locations for successive values of `j`. The other loops have no locality. Therefore, the `j` loop becomes the innermost loop when the above kernel is executed on the CPU.

For the GPU, the loop with the maximum locality becomes the outer-most loop since this

[6]The transformations proposed in this section can easily be adapted for other orderings such as row-major or column-major ordering.

enables coalescing for the greatest number of accesses. If a particular kernel is mapped to the GPU, final code generation maps the iteration number in the outermost loop in the GPU order to the thread ID in the X direction. It generates thread blocks of size (32, 1, 1). Since CUDA supports only 2 dimensional grids, the iteration space of the next two loops (if they exist) is flattened into the block ID in the y direction. In the above example, `j` becomes the outer most loop and the `i` and `k` loop get flattened into a single loop as shown below.

```
1  tempVar0 = Nz + 1;
2  for j = 1:Sy
3    for tidy = 0:(Sx*Sz)-1
4      i = floor(tidy/Sz) + 1;
5      k = (tidy%Sz) + 1;
6      indexVar1 = 2 + (k-1);
7      tempVar2(i, j, k) = Ey(i, j, indexVar1);
8    end
9  end
```

It is possible that changing the layout of certain arrays depending on the surrounding loops may improve performance by either improving locality on CPUs or increasing the number of coalesced accesses on the GPU. We leave this problem for future work.

3.2.3 Mapping and Scheduling

Once kernels have been identified by the preceding steps of the compiler, each MATLAB basic block can be viewed as a directed acyclic graph (DAG) whose nodes represent kernels. Edges represent data dependencies. It is now required to assign a processor and a start time to each node in the DAG such that the total execution time of the MATLAB basic block under consideration is minimized. This problem is similar to the traditional resource constrained task/instruction scheduling problem except that some tasks can be scheduled either on the CPU or GPU. Therefore, when dependent tasks are mapped to different types of processors, required data transfers also need to be inserted. The above problem can be shown to be NP-hard by reducing the single processor scheduling problem to it. Thus, the above problem is harder than the scheduling problem. Hence, we propose a heuristic approach for the integrated mapping

and scheduling problem.

Our compiler uses a variant of list scheduling to assign nodes to processors. The heuristic is based on the observation that a node whose performance is very good on one processor and very bad on the other needs to be given maximum chance to be assigned to the processor on which it performs best. We define $skew(n)$ where n is a node in the DAG as

$$skew(n) = max\left(\frac{T_{GPU}(n)}{T_{CPU}(n)}, \frac{T_{CPU}(n)}{T_{GPU}(n)}\right)$$

where T_{GPU} and T_{CPU} are functions that give the estimated execution time of n on the GPU and CPU respectively. To obtain estimates for the execution time for each node, we currently use profiling information.

The scheduling algorithm maintains resource vectors for the GPU, CPU and the PCI bus. In every iteration, it gets the node from the ready queue with the highest skew and tries to schedule it so that the finish time is minimized. While scheduling a node, data transfers required to execute the node need to be considered. The algorithm computes the earliest time at which the node can start execution on the GPU and the CPU, considering the data transfers that need to be performed so that all required data is available. Also, the finish time of the node is computed for both the CPU and the GPU and the node is then scheduled on the processor on which it has the lower finish time. The details are given in Algorithm 2.

In the algorithm, $readyQueue$ is a priority queue that is prioritized based on $skew$. The function $computeMinimumStartTime$ computes the minimum time at which a node can start execution on a given processor considering the data transfers that are needed due to processor assignments of predecessor nodes and the availability of the GPU, CPU and the memory bus. The functions $scheduleOnGPU$ and $scheduleOnCPU$ schedule a node onto the GPU and CPU respectively. They also insert and schedule the required data transfers and update the resource vectors for the GPU, CPU and the PCI bus.

3.2.4 Global Data Transfer Insertion

All the previously described optimizations are performed per MATLAB basic block. The mapping heuristic assumes that data is initially available on both the GPU and CPU. However, where

Algorithm 2 Processor Assignment

1: **procedure** Assignment (DataFlowGraph $G = (V, E)$)
2: $readyQueue \leftarrow \phi$
3: $\forall v \in V$ with no ***in edges*** : $readyQueue.insert(v)$
4: **while** $readyQueue \neq \phi$ **do**
5: $v \leftarrow readyQueue.remove()$
6: $start_{GPU} \leftarrow computeMinimumStartTime(v, GPU)$
7: $start_{CPU} \leftarrow computeMinimumStartTime(v, CPU)$
8: $finish_{GPU} \leftarrow start_{GPU} + T_{GPU}(v)$
9: $finish_{CPU} \leftarrow start_{CPU} + T_{CPU}(v)$
10: **if** $finish_{GPU} \leq finish_{CPU}$ **then**
11: $scheduleOnGPU(v, start_{GPU})$
12: **else**
13: $scheduleOnCPU(v, start_{CPU})$
14: **end if**
15: $V = V - v$
16: **for all** $v_1 \in V$ with all preds scheduled **do**
17: $readyQueue.insert(v_1)$
18: **end for**
19: **end while**
20: **end procedure**

the up-to-date copy of the data actually is, when a MATLAB basic block begins executing, depends on the program path that was taken to get to the MATLAB basic block.

Consider the code segment in Figure 3.6. Assume that statement 4 is executed on the GPU while the rest of the statements are executed on the CPU. When execution reaches statement 4, an up-to-date copy of the matrix `A` may either be in the CPU memory or GPU memory depending on the path the program took to get to statement 4. If the path taken was 1→2→4, then `A` is in CPU memory since statement 2 runs on the CPU. However, if the path taken to reach statement 4 was 1→2→4→4, then `A` is in GPU memory since statement 4 runs on the GPU. Additionally, statement 4 assumes that `A` is available on the GPU.

To remove this dependence of a variable's location on the program path taken, we use a two step process. In the first step, a data flow analysis is performed to compute the locations of all program variables at the entry and exit of each MATLAB basic block in the program, i.e., for each program variable at each MATLAB basic block boundary, the analysis computes whether

```
1:  n = 100;          (CPU)
2:  A = rand(n, n);   (CPU)
3:  for i=1:n
4:    A = A * A;      (GPU)
5:  end
6:  print(A);         (CPU)
```

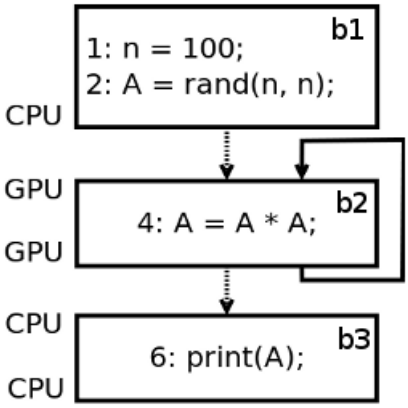


Figure 3.6: Motivating Example for Edge Splitting. Processor name in brackets indicates mapping. The control flow graph shows split edges as dotted lines and the location of `A` at each program point.

an up-to-date copy of the variable is present on the GPU or the CPU. Then, control flow edges whose source and destination have variables in different locations are split.

3.2.4.1 Variable Location Analysis

The aim of the data flow analysis is to compute the possible locations for each program variable at the beginning and end of every MATLAB basic block. The location of a variable is an element from the lattice $\mathcal{L}$, defined over the set

$$\mathcal{T} = \{Unknown, CPUAndGPU, GPU, CPU\}$$

with the partial order $\prec$ and the join operator $\wedge$ where,

$$\perp = Unknown, \top = CPU$$

$$Unknown \prec CPUAndGPU \prec GPU \prec CPU$$

Our reasoning for deciding on this particular lattice is as follows. First, the reason CPU dominates GPU in the lattice is that the CPU is the coordinating processor. At a program point where a variable is on the GPU along one reaching path, and on the CPU on another, it is probably better to move the variable to the CPU since the CPU in general has more memory. Also, $CPUAndGPU$ is dominated by both CPU and GPU to avoid extra transfers. Consider the

case when GPU is dominated by $CPUandGPU$. Then, at a program point where a variable is on the GPU on one reaching path, and on both along another, a transfer would have been needed for the GPU path to make the variable available on both. However, to say that the variable is only available on the GPU and no longer available on the CPU requires fewer transfers. Similar reasoning applies for the CPU case.

If $Vars$ is the set of all variables in the program and B is the set of all MATLAB basic blocks, for each MATLAB basic block $b \in B$ and each $v \in Vars$, the following values are defined.

1. $pos_s(v, b) \in \mathcal{T}$ is the location of v at the start of b.

2. $pos_e(v, b) \in \mathcal{T}$ is the location of v at the exit of b.

3. $asmp(v, b) \in \mathcal{T}$ is the location where v is assumed to be at the start of b. It depends on the mapping of all statements in b that read v before it is written in b. It is $\perp$ if v is not accessed in b. Intuitively, $asmp(v, b)$ is the assumption the schedule of b makes regarding the location of the variable v. For example, MATLAB basic block b2 in Figure 3.6 has $asmp(A, b2) = GPU$ as it assumes that `A` is available on the GPU when it starts executing.

4. $grnt(v, b) \in \mathcal{T}$ indicates the location of v after the execution of b. If v is not used in b it is defined to be $\perp$. If the last statement in b that modifies v is mapped to the GPU, then the value is GPU. Otherwise, its value is CPU. Intuitively, $grnt(v, b)$ is the location v is guaranteed to be in after the execution of b. For example, b2 in Figure 3.6 has $grnt(A, b2) = GPU$ as statement 4 produces the new value of `A` on the GPU.

The data flow analysis is defined by the following equations.

$$pos_s(v, b) = \begin{cases} asmp(v, b) & \text{if } asmp(v, b) \neq \perp; \\ \bigwedge_{b' \in preds(b)} pos_e(v, b') & \text{Otherwise.} \end{cases} \tag{3.2.5}$$

$$pos_e(v, b) = \begin{cases} grnt(v, b) & \text{if } grnt(v, b) \neq \perp; \\ pos_s(v, b) & \text{Otherwise.} \end{cases} \tag{3.2.6}$$

A standard forward data flow analysis algorithm [3] is used to find a fixed point for $pos_s(v, b)$ and $pos_e(v, b)$ using the data flow equations defined in equations 3.2.5 and 3.2.6. When a fixed point is reached for pos_s and pos_e, the values of $pos_s(v, b)$ and $pos_e(v, b)$ signify where v needs to be at the start and end of MATLAB basic block b respectively.

The motivating example has three MATLAB basic blocks as shown in Figure 3.6. After the data flow analysis is performed, the values of pos_s and pos_e for the variable A are: $pos_s(A, b_1) = Unknown$; $pos_e(A, b_1) = CPU$; $pos_s(A, b_2) = GPU$; $pos_e(A, b_2) = GPU$; $pos_s(A, b_3) = CPU$ and $pos_e(A, b_3) = CPU$.

3.2.4.2 Edge Splitting

Once the data flow analysis is performed, the location of every variable at the entry and exit of every MATLAB basic block is known. When a variable changes locations across a control flow edge, a transfer needs to be inserted. More precisely, a transfer for a variable v needs to be inserted on a control flow edge $e = (b_1, b_2)$, when $pos_e(v, b_1) \neq pos_s(v, b_2)$.

Edge splitting is needed because a MATLAB basic block b may have multiple incoming edges, only a subset of which need transfers for a certain variable. For example, MATLAB basic block $b2$ in Figure 3.6 has two incoming edges from $b1$ and $b2$ while a transfer is needed only on the edge from MATLAB basic block $b1$. Thus, the edge from $b1$ to $b2$ needs to be split. However, MEGHA does not have the ability to explicitly split a control flow edge as it emits structured C++ code[7]. We therefore split edges by dynamically determining which control flow edge was taken to reach the current MATLAB basic block. This is accomplished by assigning ID numbers to each MATLAB basic block and inserting code to set the value of a variable, `_prevBasicBlock`, to the ID at the end of the MATLAB basic block (as shown in statement 3, 8 and 13 in the listing below). Then, at the entry of MATLAB basic blocks, `_prevBasicBlock` can be checked to determine which control flow edge was taken and the required transfers can be performed. Conditional statements 5 and 10 in the listing below execute the data transfer only when it is required. This results in the following code.

[7]This decision was made so that the optimizations of the C++ compiler would be more effective.

```
n = 100;
A = rand(n, n);
_prevBasicBlock = 1;
for i=1:n
   if (_prevBasicBlock == 1)
      transferToGPU(A);
   A = A * A;
   _prevBasicBlock = 2;
end
if (_prevBasicBlock == 2)
   transferToCPU(A);
print(A);
_prevBasicBlock = 3;
```

3.2.5 Code Generation

Currently our code generator generates C++ code for parts of the input program mapped to the CPU and CUDA kernels for all kernels mapped to the GPU. For kernels that are assigned to the CPU, the loop nests are generated with the computed CPU loop order.

For a kernel mapped to the GPU, each thread of the generated CUDA kernel performs the computation for a single iteration of the surrounding loop nest. Assignment of threads to points in the iteration space is performed as described in Section 3.2.2. For the example in Section 3.2.2, the following CUDA kernel would be generated.

```
__global__ void gpuKernel(float *Ey, float *tempVar2, int Sx, int Sy, int Sz)
{
  int tidx = (blockIdx.x*blockDim.x)+threadIdx.x;
  int tidy = (blockIdx.y*blockDim.y)+threadIdx.y;
  j = tidx + 1;
  i = floor(tidy/Sz) + 1;
  k = (tidy%Sz) + 1;
  if (i>Sx || j>Sy || k>Sz) return;
  indexVar1 = 2 + (k-1);
  tempVar2(i, j, k) = Ey(i, j, indexVar1);
}
```

The above kernel would be called as follows.

```
1  dim3 grid(ceil(Sx/32), Sy*Sz, 1);
2  dim3 block(32, 1, 1);
3  gpuKernel<<< grid, block >>>(Ey, tempVar2,
4                               Sx, Sy, Sz);
```

Similar code is generated for DOALL loops. Each GPU thread performs the work of one iteration of the loop in the input program. We chose to generate GPU kernels in this form since it is suitable for further automatic optimization [77].

For kernels that perform matrix multiplication, i.e., use the "$*$" operator in MATLAB to multiply two variables whose types have been inferred as matrix, MEGHA's code generator emits calls to the matrix multiply in CUBLAS rather than trying to generate an optimized matrix multiply kernel. This is possible because a matrix multiply kernel cannot be composed with other kernels (Rule 1 for legal kernel composition).

To perform memory transfers, the code generator uses the blocking `cudaMemcpy` routine. Therefore, the generated code does not overlap memory transfers with computation. However, it is possible to implement a code generation scheme that achieves such overlap, which would further improve the performance of the generated code. We leave this to future work.

3.3 Experimental Evaluation

We prototyped MEGHA using the GNU Octave [23] system and used the benchmarks listed in Table 3.1 to evaluate it. These benchmarks are from several previous projects [36] with the exception of `bscholes`, `filter` and `MatrixMul` which were developed in-house. For each benchmark the table shows the number of tasks identified by our compiler, e.g. `clos` has a total of 37 tasks out of which 27 are GPU friendly kernels and 10 are CPU only tasks. The number of kernels can be larger than the number of MATLAB statements as the compiler frontend generates many IR statements for each statement in the MATLAB code. We had to make minor modifications to `capr` as our compiler does not support *auto-growing arrays*. We modified the loop counts to increase the amount of computation in `clos` and `fiff`. As our compiler does not support user functions, we manually inlined calls to user-defined functions in

Benchmark	Description	# of Lines	Tasks
bscholes(dp)	Stock Option Pricing	50	35/7
capr	Line Capacitance	60	69/23
clos(dp)	Transitive Closure	30	27/10
crni	Heat Equation Solver	60	55/14
dich	Laplace Equation Solver	55	68/15
edit	Edit Distance	60	35/15
fdtd(dp)	EM Field Computation	70	74/12
fiff	Wave Equation Solver	40	45/11
nb1d(dp)	1D N-Body Simulation	80	63/18
nb3d(dp)	3D N-Body Simulation	75	69/26
MatrixMul(dp)	Naive Matrix Multiplication	17	6/2
filter(dp)	Filter Bank	60	18/3

Table 3.1: Description of the benchmark suite. (dp) indicates benchmarks with data parallel regions. Number of kernels/number of CPU only statements are reported under the column Tasks.

all these benchmarks. The code generated by our compiler was compiled using `nvcc` (version 2.3) with the optimization level -O3. The generated executables were run on a machine with an Intel Xeon 2.83GHz quad-core processor[8] and a GeForce 8800 GTS 512 graphics processor. We also ran these on a machine with the same CPU but with a Tesla S1070 [47]. Both machines run Linux with kernel version 2.6.26. We inserted calls to the `gettimeofday` function to measure the runtimes of executables generated by MEGHA and used the MATLAB function `clock` to measure the runtime for MATLAB programs run in the MATLAB environment.

For each benchmark, we report the runtime in the MATLAB environment (referred to as "baseline MATLAB"), execution time of CPU only C++ code generated by our compiler (referred to as CPU-only execution) and the execution time of C++ and CUDA code generated by our compiler and the speedups achieved by the later two versions over baseline MATLAB execution. Each benchmark was run three times. The reported runtime is the average execution time per run.

Table 3.2 shows the runtimes for benchmarks with data parallel regions. Figure 3.7 plots

[8]Although our experimental system contained two quad-core Xeon processors (8 cores), in our experiments, we use only a single core as the present compiler schedules tasks to a single GPU and a single CPU core.

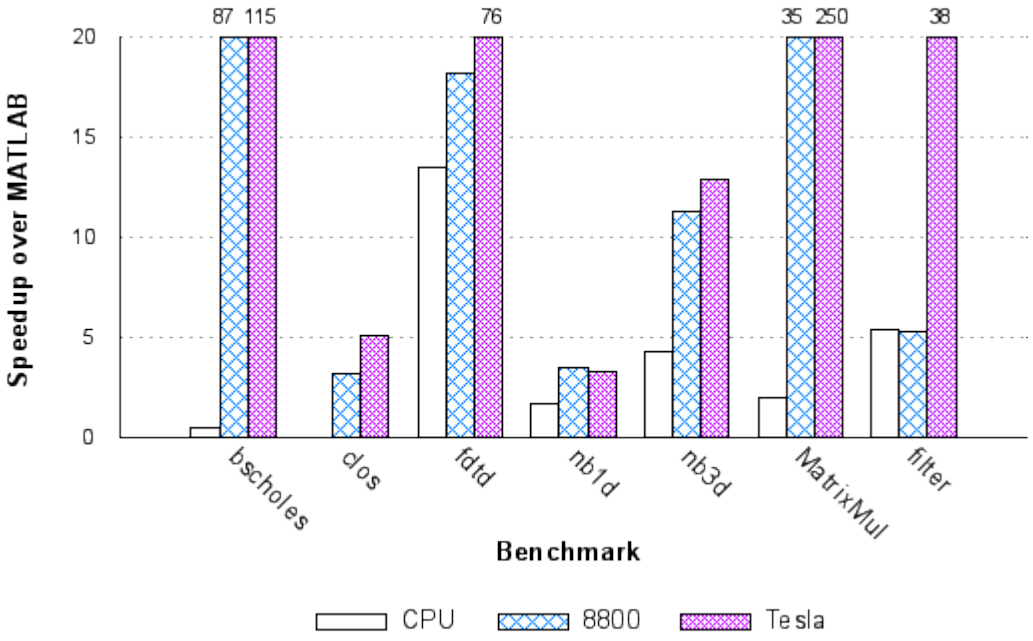


Figure 3.7: Speedup of generated code over baseline MATLAB execution

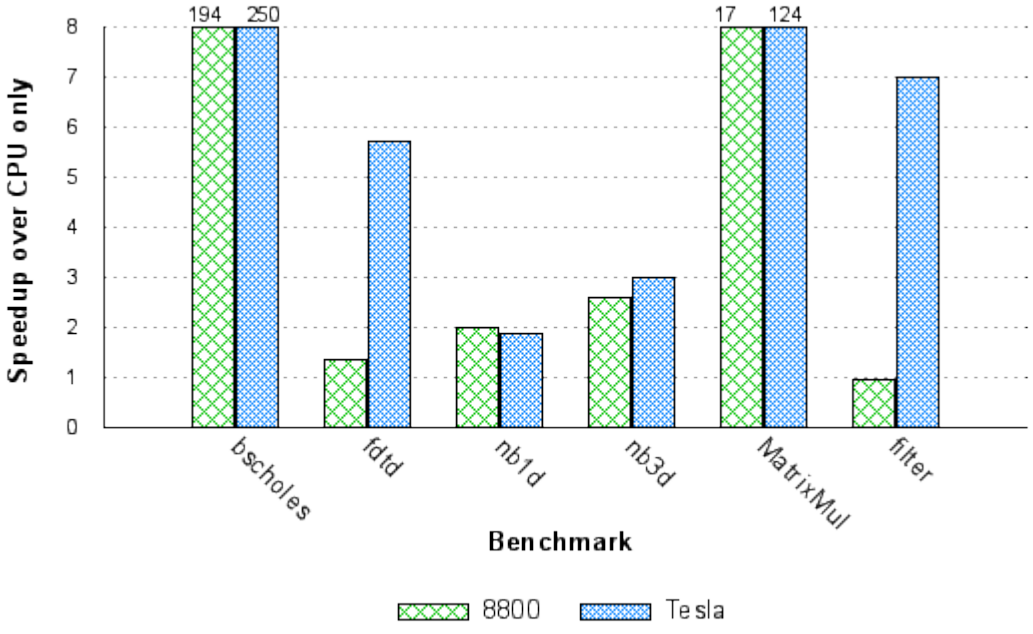


Figure 3.8: Speedup of generated CPU+GPU code over generated CPU only code

Benchmark	Size	Execution Time (in Seconds)			
		MATLAB	CPU	CPU+8800	CPU+Tesla
bscholes	51200 Options, 100 Iters	0.9	1.99	0.03	0.02
	102400 Options, 500 Iters	8.9	19.7	0.07	0.05
	204800 Options, 500 Iters	17.2	39.5	0.1	0.09
clos	1024 vertices	0.5	-	0.18	0.11
	2048 vertices	4.1	-	1.13	0.73
	4096 vertices	30.8	-	9.15	5.82
fdtd	$512 \times 512 \times 15$	187	13.5	10.1	2.94
	$512 \times 512 \times 32$	408	30.5	22.69	5.48
	$1024 \times 1024 \times 32$	1623	121.6	mem-exc	17.2
nb1d	4096 bodies	14.8	7.6	5.4	5.4
	8192 bodies	50.1	29.2	13.7	15
	16384 bodies	513	323	119	128
nb3d	512 bodies	42.3	12.4	4.82	3.64
	1024 bodies	224.8	47.5	17.8	16.7
	1536 bodies	508	101.5	38.6	37.15
MatrixMul	1024×1024	16.2	4.31	1.06	0.16
	2048×2048	580	550	7.4	0.94
filter	4×128 tap, 1M	5.95	1.15	1.15	0.17
	4×128 tap, 1.5M	8.92	1.73	1.76	0.23
	4×256 tap, 1M	11.5	2.07	2.15	0.3
	4×256 tap, 1.5M	17.42	3.02	3.22	0.45

Table 3.2: Runtimes for data parallel benchmarks

the geometric mean speedup, over different input sizes, of generated code over native MATLAB execution for each benchmark. In benchmark `bscholes` we get a speedup of 191X over MATLAB for large problem sizes. In this benchmark our compiler was able to identify the entire computation as one kernel and execute it on the GPU. Baseline MATLAB performs better than our CPU only code in `bscholes` as it handles full array operations better than our C++ code. The benchmark `fdtd`[9] achieves speedups of 18X and 76X on the 8800 and Tesla respectively. Speedups for `fdtd` are higher on Tesla as memory accesses are coalesced better on the Tesla than on the 8800. We did not report CPU-only runtimes for the benchmark `clos` as it uses matrix multiplication and our C++ code generator at present generates naïve code for matrix

[9]Benchmark `fdtd` (problem size of 1024x1024x32) could not be run on the 8800 as the memory requirement exceeded the available memory (512MB) in the GPU.

multiplication. However, these routines can easily be replaced by calls to optimized BLAS libraries. The GPU versions for `clos`, which use the CUBLAS library, achieve a speedup of 2.8X-5.6X on 8800 and Tesla. For `nb1d` and `nb3d` the GPU versions achieve speedups of 2.7X-4.0X and 8.8X-13.7X respectively. The benchmarks `filter` and `MatrixMul` have parallel loops which our compiler is able to identify and execute on the GPU. The speedups in these cases are disproportionately large because the execution of these loops in baseline MATLAB is slow. The generated code for `filter` is about 5X faster than baseline MATLAB on the 8800 but it is about 37X faster on the Tesla. In the case of `filter`, execution times are similar for the 8800 and the CPU-only versions because our code generator currently does not generate code to execute CPU code in parallel with GPU kernels and data transfers while the scheduler assumes this is possible. Lastly, even the CPU only version gives considerable speedups of 1.6X-13.8X for `fdtd`, `nb1d` and `nb3d` on a single CPU core.

Figure 3.8 shows the geometric mean speedup of generated CPU+GPU code over generated CPU-only code. Speedups are averaged over different input sizes shown in Table 3.2. The plot shows that the speedups for most benchmarks are reasonably high, thus demonstrating that there are significant benefits from compiling MATLAB code to use GPUs rather than just to CPU code. `bscholes` and `MatrixMul` show very high speedups when compiled to the GPU. The speedups for the remaining benchmarks are in the range 1.5X-7X. The geometric mean speedup of CPU+GPU code over CPU-only code over all benchmarks is 5.3X on the GeForce 8800 GTS and 13.8X on the Tesla S1070.

Table 3.3 shows the runtimes for benchmarks with no or very few data parallel regions. The performance benefits over MATLAB are mainly due to compiling the MATLAB code to C++. Even here, we observe an improvement in execution time over baseline MATLAB execution by factors of 2X-10X.

Table 3.4 compares performance of code generated by our compiler with that of GPUmat [28]. GPUmat requires the user to explicitly mark variables as GPU variables before an operation on them can be performed on the GPU. The user has to manually identify data parallel regions and insert code to move data to the GPU only in these regions. Incorrect use of GPU variables can have an adverse impact on performance. We used information from the code gen-

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	CPU	CPU+8800
capr	256 × 512	9.15	1.43	1.43
	500 × 1000	17.3	2.65	2.66
	1000 × 2000	34.7	5.37	5.38
crni	2000 × 1000	16.05	6.33	6.39
	4000 × 2000	64.1	24.7	23
	8000 × 4000	254.8	104	103
dich	300 × 300	1.24	0.3	0.3
	2000 × 2000	55.7	8.01	8.15
	4000 × 4000	249.2	31.95	31.89
edit	512	0.35	0.11	0.11
	4096	22.1	0.45	0.45
	8192	88.1	1.47	1.48
fiff	450 × 450	0.51	0.22	0.21
	2000 × 2000	2.04	0.41	0.41
	4096 × 4096	8.17	2.5	2.51

Table 3.3: Runtimes for non data parallel benchmarks

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	GPUmat	CPU+Tesla
`bscholes`	102400, 500	8.9	7.2	0.05
`clos`	2048	4.1	1.13	0.73
`fdtd`	512 × 512 × 15	187	30	2.94
`nb1d`	4096	14.8	12.03	5.44
`nb3d`	1024	224.8	172	16.67

Table 3.4: Comparison with GPUmat

erated by our compiler to mark the GPU variables. GPUmat performs considerably better than MATLAB. However, because our compiler can automatically identify GPU kernels and reduce the number of accesses to GPU global memory by intelligent kernel composition, we were able to get better performance (1.5X-10X) than GPUmat.

Our compiler reorders loops to improve cache locality on CPUs and coalesced accesses on GPUs as discussed in Section 3.2.2. Table 3.5 shows the performance gains obtained by such reordering of loop nests. The benchmark `fdtd` has 3 dimensional arrays and benefits a lot on

Benchmark	Execution Time (in Seconds)				
	MATLAB	CPU Only		CPU + GPU	
		No Opt	Opt	No Opt	Opt
`fdtd`	187	82.05	13.5	48.14	2.94
`nb3d`	224.8	50.1	47.5	47.6	16.7

Table 3.5: Performance gains with parallel loop reordering. Problem sizes are 512 $\times$ 512 $\times$ 15(`fdtd`) and 1024(`nb3d`).

both the CPU and GPU from this optimization. Loop reordering improved the performance of `fdtd` CPU+GPU code by a factor of 16 and CPU only code by a factor of 6.1. Whereas for `nb3d` this optimization results in an improvement of 3X for CPU+GPU code and 1.05X for CPU only code. The runtimes reported in Table 3.2 and the speedups in Figures 3.7 and 3.8 are for code generated with the loop reordering transformation enabled. Since the other data parallel benchmarks work primarily on single dimensional arrays, there are not many opportunities to perform loop reordering while compiling them.

For the results reported we used a look ahead factor of 3 ($k = 3$) while performing kernel composition (refer to Section 3.2.1.2. We believe this value provides a sufficient degree of look ahead without significantly increasing compile time. For the benchmarks considered in this thesis, we found that performance of generated code is not sensitive to changes in the value of k.

For the results reported above, we used the same inputs to generate profile data and run the generated code. However, to demonstrate that the performance of generated code does not depend on this, we experimented with profile data generated with inputs smaller than what the generated code was to be run with, and found that our compiler made identical mapping decisions. Therefore, the runtimes of the generated code was identical to those reported in Table 3.2.

In order to better understand the benefits of our global transfer insertion scheme (Section 3.2.4), we measured the reduction in total transfer time it achieves compared to a naïve scheme. In the naïve scheme, we ensure that up-to-date copies of all variables are available in CPU memory at the end of each MATLAB basic block. At the beginning of each MATLAB basic

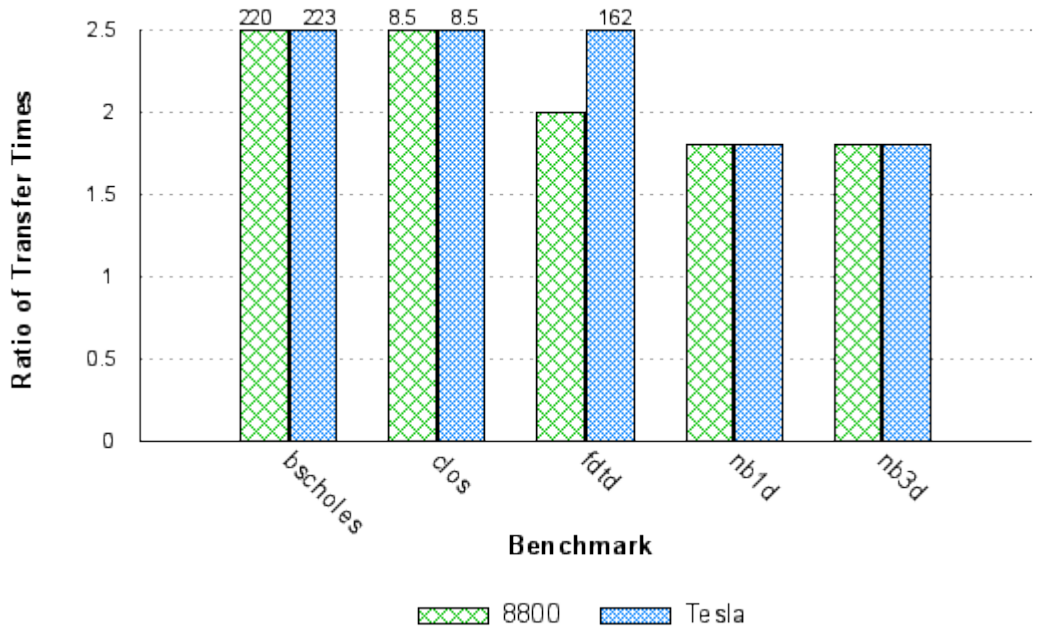


Figure 3.9: Reduction in total data transfer time using our global transfer insertion scheme (Section 3.2.4) compared to the naïve scheme.

block, all variables that are needed on the GPU in that MATLAB basic block are transferred to GPU memory. Figure 3.9 shows the reduction in the amount of data transfer our technique is able to achieve compared to this naïve scheme. To measure total data transfer time, we used the largest input size that successfully executed on the 8800 for each benchmark. For the benchmark `bscholes`, our technique is able to achieve a 220X saving in the amount of data transfer. This is because our technique is able to keep all required data on the GPU in the main loop of this benchmark, while the naïve scheme must perform transfers in every iteration. This is also the case for `fdtd` running on the Tesla. For the other benchmarks, our technique reduces the amount of data transfer by factors ranging from 1.8X to 8.5X. Therefore, our technique reduces the amount of data transfer needed significantly, thereby improving performance of generated code.

3.4 Summary

In this chapter, we described the design and implementation of MEGHA, a compiler to compile MATLAB programs for execution on heterogeneous machines. Our compiler first simplifies the

input MATLAB code and constructs an SSA IR. It then identifies "GPU friendly" IR statements and groups these into *kernels*. By doing this, the compiler is able to eliminate several temporary arrays. Subsequently, a heuristic based on list scheduling is used to map identified kernels to either the CPU or GPU and insert necessary data transfers. To correctly handle dependences across MATLAB basic block boundaries, our compiler performs a data flow analysis to determine the location (CPU or GPU memory) of program variables and then uses an edge splitting strategy to ensure data required by each computation is available in the correct location. Experimental evaluation on a set of data parallel MATLAB benchmarks shows that code generated by our compiler achieves a geometric mean speedup of 12X on the GeForce 8800 GTS and 29.2X on the Tesla S1070 over baseline MATLAB execution. The generated code is also up to 10X faster than MATLAB code accelerated using GPUmat [28]. Compared to compiled MATLAB code running on the CPU, the generated code is 5.3X faster on the GeForce 8800 GTS and 13.8X faster on the Tesla S1070.

The compiler described in this chapter does not perform any optimizations on the generated GPU kernel code. Additionally, it assumes that it is possible to keep variables live on the GPU for as long as required. However, this may not be possible due to the limited memory available on the GPU. In the next chapter, we discuss these problems in more detail and propose methods to address them.

Chapter 4

Additional Optimizations

In the previous chapter, we described the structure of the basic compiler and the details of its implementation. The compiler however did not take into account, for reasons of simplicity, certain constraints such as the amount of GPU memory available. In this chapter, we describe additional analysis and transformations needed in order to take into account the above constraints. Additionally, we describe two methods to optimize the generated code. Section 4.1 discusses the problem of managing GPU device memory and presents our technique to address this problem. Optimizations targeting memory accesses on the GPU are discussed in Section 4.2.

4.1 GPU Device Memory Management

In Section 3.2.3 (Chapter 3), we discussed a heuristic algorithm for partitioning work between the CPU and the GPU and inserting the required data transfers between CPU and GPU memory. The algorithm assumes that a transfer needs to be inserted only when a variable that is up-to-date in GPU memory, but stale in CPU memory, is needed by the CPU or vice versa. Note that several variables could be "live" in GPU memory simultaneously. Algorithm 2 does not take into account the fact that the GPU has limited memory and that it may not be possible to fit all variables assumed to be simultaneously live on the GPU into the memory available on the GPU.

GPU memory needs to be considered as a constraint in two contexts.

1. First, the working set[1] of each kernel scheduled on the GPU has to fit in GPU memory. It is the responsibility of the kernel composition process to ensure that the working set fits in GPU memory. This is enforced by Equation 3.2.4.

2. In the second case, the sum of sizes of all variables that are live on the GPU when a particular kernel is being run must not be larger than the size of GPU memory. For example, consider the case when some variable `A` is accessed by a previous kernel K1, but is not accessed in the current kernel K2 and is again accessed by a future kernel K3 without any intervening accesses on the CPU. Here, the GPU memory constraint may be violated when K2 is executing even if the working set of K2 fits in GPU memory. The compiler needs to ensure that the sum of sizes of all variables that are live on the GPU at any program point[2] is smaller than the size of the GPU memory. This is clearly not the same as the kernel composition problem.

We refer to the set of variables described in item 1 above as the kernel "working set" (WS) and the set of variables described in item 2 as the "live variable set" (LVS). In this section we describe a formulation and solution to ensure that the LVS always fits in the GPU memory.

4.1.1 Motivating Example

To better understand the GPU memory management problem, consider the following piece of code where all kernels are mapped to the GPU. Here, `A`, `B` and `C` are `real` matrices of size 1000×1000 while `D` and `E` are `real` matrices of size 500×500. Assuming that `reals` are represented using 4 bytes, `A`, `B` and `C` are of size 4 MB each and `D` and `E` are of size 1 MB each. For the sake of this example, assume that the GPU has 12MB of memory.

[1] The working set size of a kernel is defined as the sum of the sizes of all variables accessed in the kernel. For all array variables used, we assume that the whole array is used in order to compute the working set size even though only a few elements of the array may be accessed in the kernel. Using range analysis, a more accurate estimate of the working set size can be obtained.

[2] We use *program point* and *program statement* interchangeably in this section.

```
1  // ...
2  Kernel1(A, B, C); //WS size = 12 MB; LVS Size = 12 MB
3  Kernel2(C, D, E); //WS size =  6 MB; LVS Size = 14 MB
4  Kernel3(A, B, C); //WS size = 12 MB; LVS Size = 12 MB
5  // ...
```

Some required transfers are not shown in the above code for simplicity. For example, `D` and `E` need to be transferred to the GPU memory after line 2. However, these transfers can be easily inferred assuming that the references in the above code are the only references to all variables in the listing. The values of WS size and LVS size for each kernel are also shown in the code as comments. For example, `A`, `B` and `C` are live on the GPU at line 2 and therefore the LVS size and WS size are both 12 MB (assuming nothing else was on the GPU to start with). To execute `Kernel2` on the GPU, `D` and `E` need to be brought to the GPU. However, since `A`, `B` and `C` are already on the GPU, the LVS size on the GPU during the execution of `Kernel2` is 14 MB, which exceeds the 12 MB available on the GPU. However, once `Kernel2` has finished executing, assuming `D` and `E` are no longer needed on the GPU, the LVS size during the execution of `Kernel3` comes down to 12 MB.

From this example it is clear that ensuring the working set of each kernel fits in GPU memory is not sufficient. Further analysis and appropriate transformations are needed to ensure that the LVS always fits in GPU memory. For the example above, consider "spilling" the array `A` from GPU memory, i.e., we insert transfers to and from GPU memory before and after each kernel that uses `A`. If we do this, the code above changes to the following. For brevity, we again do not show transfers pertaining to other variables.

```
1  // ...
2  TransferToGPU(A);
3  Kernel1(A, B, C); //WS size = 12 MB; LVS Size = 12 MB
4  TransferFromGPU(A);
5  Kernel2(C, D, E); //WS size =  6 MB; LVS Size = 10 MB
6  TransferToGPU(A);
7  Kernel3(A, B, C); //WS size = 12 MB; LVS Size = 12 MB
8  TransferFromGPU(A);
9  // ...
```

Spilling `A` splits `A`'s *live range* on the GPU and therefore it now overlaps with the GPU lifetimes of fewer variables. By GPU lifetime or GPU live range of a variable, we mean that part of the program where the variable is live and an up-to-date copy is assumed to be present in GPU memory. After the transfers for `A` are inserted, `A` is no longer on the GPU when `Kernel2` is executed. Therefore, the total memory usage at line 5 goes down to 10 MB from the earlier 14 MB. The program can now be correctly executed on our hypothetical GPU with 12 MB of memory.

Thus, after assigning kernels to either the CPU or the GPU, additional steps are needed to ensure that in the generated code, the LVS size is always smaller than available GPU memory.

4.1.2 Overview of Our Approach

In this section we present a brief overview of our approach. Firstly, we need to determine the GPU live ranges[3] of each variable, i.e., we want to determine all program points at which a particular variable is assumed to exist in the GPU memory. Once the live ranges of variables are computed, we need to determine if the LVS fits into the GPU memory at all program points. We map this problem to that of finding all maximal cliques in a graph. By modeling the GPU live ranges as intervals, we can transform this problem to that of finding maximal cliques in an interval graph. We leverage the existing *linear scan* [57] algorithm to design an efficient solution for this. If the LVS does not fit in the GPU memory at a program point, then we need to determine which variables to "spill" and insert the transfers required to spill these variables. This is done using a greedy heuristic that takes into account variable sizes and spill costs.

The high level algorithm to ensure that each LVS fits in the GPU memory is shown in Algorithm 3. The algorithm takes as input the program P, the function $size$ that maps variables to their sizes and Mem_{GPU}, the available memory on the GPU. First, the GPU live ranges of all variables are computed. Then, the algorithm checks whether each LVS of the generated code fits in the GPU memory. If this is not the case, we pick one live range to spill and generate spill

[3]We define the GPU live range of a variable as the set of statements (or program points) at which the variable is live and is assumed to have an up-to-date copy in GPU memory.

code for it. This process is repeated until each LVS fits in the GPU memory.[4] We discuss each of these steps in more detail in the sections that follow.

Algorithm 3 Algorithm for GPU Memory Management

```
1: procedure TransformProgram(P, size, Mem_GPU)
2: while true do
3:    liveRanges ← ComputeGPULiveRanges(P)
4:    LVSfitsInGPU ← CheckLVSSize(liveRanges, size, Mem_GPU)
5:    if LVSfitsInGPU then
6:       break
7:    end if
8:    spillRange ← PickSpillRange()
9:    GenerateSpillCode(spillRange)
10: end while
11: end procedure
```

4.1.3 GPU Live Range Analysis

The GPU live range analysis needs to determine exactly the parts of the generated code where each variable is live on the GPU. As described earlier, a variable is said to be live on the GPU at a particular point in the code if it is assumed that a copy of the variable is present in GPU memory. For the purposes of this analysis, we define the GPU live range of a variable to be the set of program statements where the variable is live on the GPU.

The location of each variable at the beginning and end of each MATLAB basic block is already computed by the analysis described in Section 3.2.4. Live variable information has been computed as discussed in Section 3.2.1.3. We say that a variable is *live on the GPU* if it is on the GPU and is live. Thus, using the location and liveness information, we compute whether each variable is live on the GPU at each program point.

After a variable v is spilt, only the live ranges for that variable need to be updated. The location information of the variable v is updated to reflect that it is on the CPU at all MATLAB basic block boundaries. The new live ranges of the variable v are exactly the kernels that use

[4] In the remainder of this section, we use spilling live ranges and spilling variables interchangeably. They both mean spilling a variable within one live range of the variable.

v. Therefore, the GPU live range analysis needs to be performed only once and the information can then be incrementally updated.

4.1.4 Checking the LVS Constraint

For a program to be correctly executable on the GPU, we need to ensure that at any program point, the sum of sizes of variables live on the GPU at that point, is less than or equal to the memory available on the GPU. More precisely, if $Vars$ is the set of all program variables and $Pnts$ is the set of all program points, then for each $p \in Pnts$, we require

$$\sum_{v \in Vars} l_p(v).size(v) \leq M_{GPU} \tag{4.1.1}$$

Here, l_p is 1 if v is live on the GPU at p and 0 otherwise, $size$ is a function that maps a variable to its size and M_{GPU} is the total memory available on the GPU. We say that code that does not satisfy Equation 4.1.1 does not meet the LVS constraint.

To check whether generated code satisfies Equation 4.1.1, we need an efficient way to represent the overlap of GPU live ranges of various program variables. *Interference graphs* [16] are an ideal representation for information about the overlap of live ranges. They have been very successfully used in register allocation to represent live range interference information [16, 13]. First, we define a *spill candidate* as follows.

Definition 5 ***Spill candidates*** *are those live ranges that are live on the GPU at a program point where memory usage on the GPU exceeds the amount of memory available on the GPU.*

The following subsections describe how we can efficiently check if generated code meets the LVS constraint and determine spill candidates after presenting some background on interference graphs.

4.1.4.1 Interference Graphs

An *interference graph* is an undirected graph whose nodes represent the live ranges of program variables and whose edges represent interference or overlap of two live ranges. More precisely the interference graph of a program is an undirected graph $G = (V, E)$, where V

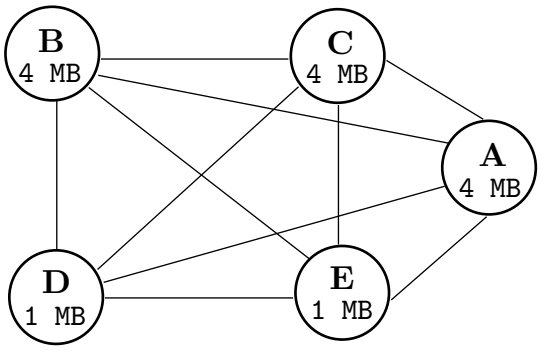


Figure 4.1: Interference graph for the motivating example

is the set of nodes in the graph and E is the set of edges. Each node n in the graph represents a single live range. $liverange(n)$ represents the live range the node n represents and $var(n)$ is the variable whose live range n represents. Each live range is a set of program points (or equivalently, program statements). There is edge $e \in E$ between n_1 and n_2 iff $liverange(n_1) \cap liverange(n_2) \neq \emptyset$. One extension we need to make to the structure of the interference graph is to add a weight to each node of the graph. The weight of a node n, denoted by w_n, is equal to the size of the variable whose live range n represents.

The interference graph for the motivating example discussed in Section 4.1.1 is shown in Figure 4.1. There are five variables, all of which have non empty GPU live ranges. `A`, `B` and `C` are first transferred to the GPU before `Kernel1` (line 2) is executed and can only be removed from GPU memory after the execution of `Kernel3` (line 4). They are therefore live on the GPU even during the execution of `Kernel2` (line 3). Thus, they have GPU live ranges consisting of the statements 2, 3 and 4, i.e.

$$LR_A = LR_B = LR_C = \{2, 3, 4\} \tag{4.1.2}$$

where LR_A, LR_B and LR_C represent the GPU live ranges of `A`, `B` and `C` respectively. However, `D` and `E` are only required to be in GPU memory during the execution of `Kernel2` (line 3). Therefore, their live ranges consist of the single statement 3, i.e.

$$LR_D = LR_E = \{3\}. \tag{4.1.3}$$

Since there is a single live range corresponding to each variable in the program, the interference graph has one node per variable. Spilling the live range of a variable v results in 2 or more live

ranges, each of which is represented by a node in the interference graph.

From equations 4.1.2 and 4.1.3, it is easy to see that the intersection of the live ranges of any two variables is non-empty. Therefore, the interference graph has an edge between every pair of nodes in the graph. In other words, the interference graph is a *complete graph* [5] or *clique* containing five nodes. Figure 4.1 also shows the weight of each node in the graph that is equal to the size of the variable the node represents.

4.1.4.2 LVS Constraint Check on the Interference Graph

As explained above, we need to check whether memory usage on the GPU exceeds the memory available on the GPU at any program point. To do this, we need to determine all sets of variables that need to coexist on the GPU, i.e., all live variable sets. In other words, we need to find all sets of variables $S = \{v_1.v_2, ..., v_k\}$ so that for any $i, j \in [1, k]$ the GPU live ranges of v_i and v_j overlap, i.e. $LR_{v_i} \cap LR_{v_j} \neq \emptyset$. For the generated code to meet the LVS constraint, the sum of sizes of the variables in each LVS must be smaller than the memory available on the GPU.

From the construction of the interference graph, we know that if two live ranges intersect, then there is an edge between the nodes that represent them. Therefore, the problem of finding the LVS sets may be restated as that of finding all sets of nodes $S' = \{n_1, n_2, ..., n_k\}$ in the interference graph such that for any $i, j \in [1, k]$, the edge $e = (n_i, n_j) \in E$. In other words, this is the problem of finding all *maximal cliques* in the interference graph. Maximal cliques are cliques to which no more nodes can be added. We can now restate the LVS constraint as follows.

LVS Constraint : *For every maximal clique of the interference graph, the sum of weights of the nodes in the clique should not exceed the available memory on the GPU.*

Consider the interference graph shown in Figure 4.1. The graph is a complete graph with five nodes and therefore has a single maximal clique, $S = \{A, B, C, D, E\}$. The sum of the weights of the nodes in the clique is 14 MB. This exceeds the 12 MB of memory available on our hypothetical GPU. We therefore conclude that the code does not meet the LVS constraint.

Figure 4.2 shows the interference graph for the program after `A` has been spilt from GPU

[5] A *complete graph* is one in which there is an edge between every distinct pair of nodes.

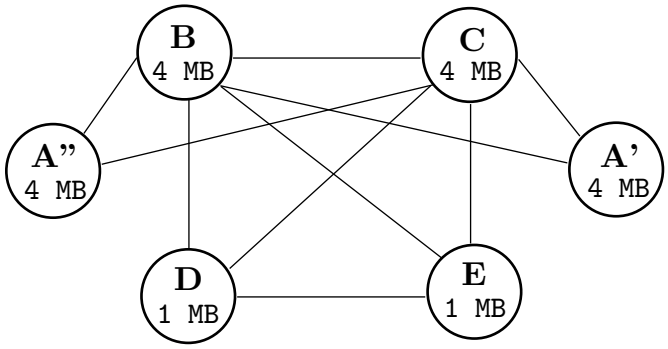


Figure 4.2: Interference graph after spilling

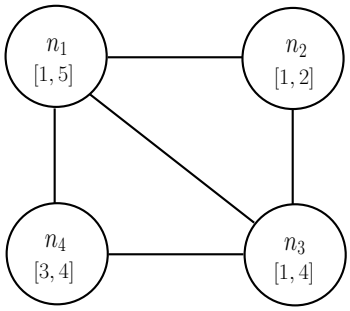


Figure 4.3: An Interval Graph

memory. In this case, `A` has two GPU live ranges, A' corresponding to its use in `Kernel1` and A'' corresponding to its use in `Kernel3`. Both of these only interfere with the live ranges of `B` and `C`. The graph now has three maximal cliques : $S_1 = \{A', B, C\}$, $S_2 = \{A'', B, C\}$ and $S_3 = \{B, C, D, E\}$. The sum of node weights for each of these cliques is 12 MB, 12 MB and 10 MB respectively. Each of these is less than or equal to the 12 MB available on the GPU which means the code now meets the LVS constraint.

Unfortunately, the problem of finding the maximal cliques of a graph is NP-Hard in the general case [20]. However, by reducing the interference graph to an *interval graph* [46, 75], it is possible to simplify the maximal clique problem and solve it efficiently.

An interval graph, $G = (N, E)$, has a set of nodes N and a set of edges E. Each node n of the interval graph can be associated with an interval of real numbers $[x, y] \subset \mathbb{R}$ represented by $interval(n)$. An edge exists between two nodes n_1 and n_2 iff $interval(n_1) \cap interval(n_2) \neq \emptyset$. Figure 4.3 shows an example of an interval graph. The interval corresponding to each node is shown inside the node.

A property of interval graphs that is useful in the current context is that problems like finding the maximal cliques of these graphs are no longer NP-Hard. Therefore, we approximate the interference graph by an interval graph and solve the maximal clique problem efficiently thereby quickly determining whether or not the generated code meets the LVS constraint.

The process of converting the interference graph to an interval graph is a simple one. To a GPU live range, we add all statements between its first and last statement thus making sure that each GPU live range can be represented as an interval of statements. This transformation can only strictly increase the GPU live range of a variable if at all the live range changes, i.e., we are over-estimating the GPU live range of some variables. Therefore, if the transformed graph meets the LVS constraint, it is obvious that the original graph also does.

4.1.4.3 Efficiently Computing LVS Size

As mentioned earlier, our algorithm to determine whether or not generated code meets the LVS constraint is based on the linear scan algorithm for register allocation [57]. Our algorithm checks whether there is a maximal clique of the approximated interference graph that violates the LVS constraint. The basic algorithm to check whether the code meets the LVS constraint is listed as Algorithm 4.

Algorithm 4 Efficiently Checking the LVS Constraint

1: **procedure** CheckLVSSize(L, $size$, Mem_{GPU})
2: $I \leftarrow roundToInterval(L)$
3: $currentSize \leftarrow 0$
4: $currentVars \leftarrow \emptyset$
5: $LVSFits \leftarrow true$
6: $spillCandidates \leftarrow \emptyset$
7: **for all** $p \in ProgramPoints$ **do**
8: $currentVars \leftarrow GPULiveVars(p, I)$
9: $currentSize \leftarrow SumSizes(currentVars, size)$
10: **if** $currentSize > Mem_{GPU}$ **then**
11: $LVSFits \leftarrow false$
12: $spillCandidates \leftarrow spillCandidates \cup liveIntervals(currentVars)$
13: **end if**
14: **end for**
15: return $LVSFits$
16: **end procedure**

The algorithm first rounds (approximates) each GPU live range (each element of the set L) to a live interval (function $roundToInterval$). To determine whether or not the code meets the LVS constraint, it iterates over all program points. At each program point p, it determines the set of variables which are live on the GPU at that point ($currentVars$) and computes the sum of their sizes, i.e., the LVS size. If this size exceeds the available memory on the GPU (Mem_{GPU}), then the currently live GPU live intervals are added to the set of spill candidates ($spillCandidates$). If at any program point, the LVS size exceeds the available memory on the GPU, the algorithm decides that the code does not meet the LVS constraint.

The basic algorithm presented above can be further optimized using the fact that GPU live ranges have been approximated to intervals. As a consequence, first, LVS sizes need to be checked only at program points at which at least one GPU live interval starts or ends. Secondly, the set of variables live on the GPU at each program point can be incrementally updated at the start and end of GPU live intervals (like the active list in the linear scan method [57]). These optimizations are similar to those made in the linear scan register allocation algorithm. The resulting algorithm gives us an efficient way to check if generated code obeys the LVS constraint.

4.1.5 Picking Spill Candidates

When the first phase of Algorithm 3 determines that the input program does not meet the LVS constraint, we need to pick one live interval from the set of spill candidates to spill. When a variable is spilt from GPU memory the compiler needs to insert transfers to GPU memory before a call to a kernel if the variable is an in-variable and a transfer to CPU memory after the kernel if the variable is an out-variable.

We would like to make spill decisions so that the number of transfers inserted is as small as possible[6]. We try to achieve this through the use of a priority metric. For each live interval we define a term called the *spill cost*. This is the number of transfers needed to spill a particular live interval from GPU memory. In the motivating example, the spill cost of `A` is 4. Also, to spill the

[6]Alternatively, it is possible to try to reduce the total data transferred (number of bytes transferred), which depends on the sizes of the variables that are spilt. While our heuristic makes use of the sizes of variables to make the spill decision, we do not use the objective mentioned above for reasons of simplicity.

least number of variables possible, the spill candidate representing the largest variable should be considered first for spilling. Taking these factors into account, we define the spill priority for a spill candidate i representing variable v as follows.

$$SpillPriority(i) = \frac{Size(v)}{SpillCost(i)} \tag{4.1.4}$$

When the compiler finds that the generated code does not meet the LVS constraint, it picks the spill candidate with the largest spill priority as the interval to spill.

4.1.6 Reverting to CPU Code

Since the GPU memory management algorithm inserts several transfers while spilling variables, the mapping decisions of Algorithm 2 need to be revisited. Ideally, we would like to rerun the mapping and scheduling algorithm to take each spill into account. However, this would be very expensive.

We currently use a simple strategy to ensure that the code emitted by the compiler does not become slower than CPU only code. The compiler estimates the running times of each modified MATLAB basic block (using the profile information) after spill transfers are inserted. If this time is greater than the estimated CPU only execution time for that MATLAB basic block, all statements in that MATLAB basic block are reverted to CPU only code. We leave finding a better strategy to handle this problem for future work.

4.2 Memory Access Optimizations

In this section we describe two memory access optimizations. Section 4.2.1 describes an optimization for memory accesses that read data shared by multiple GPU threads. Section 4.2.2 describes a transformation to coalesce memory accesses that are uncoalesced because they do not meet alignment requirements. Both of these transformations generate code that uses the *shared memory* on the GPU.

4.2.1 Data Reuse

In many numeric and scientific applications, loops whose iterations can be executed in parallel (parallel loops or DOALL loops) are common. Even though different iterations of these loops can be run in parallel, they frequently access common data. It is essential to exploit this reuse to get maximum performance. CPUs do this naturally through the use of caches. Data brought into cache by one iteration of the loop is (possibly) available to a subsequent iteration that needs it. However, this is not the case in GPUs. The loop needs to be rewritten in order to exploit locality.

The following code shows a simple implementation of an FIR filter [54]. (The translation to SSA has been left out for simplicity). The code filters input samples in the array `x` and stores the result into the array `y`. The outer `i` loop is a parallel loop that iterates over all indices where the output `y` needs to be computed. The inner `k` loop computes a weighted sum of the input over all the FIR filter taps.

```
1 for i=len:n
2   outSample = 0;
3   for k=1:len
4     outSample = outSample + f(k)*x(i-(k-1));
5   endfor
6   y(i) = outSample;
7 endfor
```

In this example, every iteration of the `i` loop reads all the values from the array `f` from `f(1)` to `f(len)`. On a CPU, if the entire array `f` could be held in some level of cache (e.g., L2 or L3), then successive iterations of the `i` loop can exploit this locality. This pattern is very common – convolution, matrix multiplication and several other applications have such access patterns.

The code listed below is the GPU code generated for the FIR filter example.

```
1 __global__ void kernel(float *x, float *y, float *f, int n, int len)
2 {
3   int tidx = threadIdx.x + blockIdx.x*blockDim.x;
4   int i = tidx+len;
5   if(i>n) return;
6   float outSample = 0;
7   for (int k=1 ; k<=len ; ++k)
8     outSample = outSample + f(k)*x(i-(k-1));
9   y(i) = outSample;
10 }
```

In the GPU code, each iteration of the outer `i` loop is executed on a different GPU thread. Therefore, all threads in a warp now read the same value of `f(k)` from GPU memory as they execute in lock step. Such accesses do not exploit any locality in the `f(k)` accesses. Our compiler automatically detects such accesses and optimizes them.

To identify such accesses, the compiler first computes the set of variables that are *identical* across different threads of the same warp at any given point in time during the kernel's execution. It does this by first assuming that parameters to the kernel are identical across a warp (`n` and `len` here). Variables that are defined as functions of identical variables are also identical. In the above example, `k` is therefore identical across all threads in the warp. Additionally, we identify if the variable is a loop index. If it is, the step size is determined. Here, `k` is a loop index with a step size of 1. Once this is done, we identify array accesses whose indices are identical across the warp. Here `f(k)` is such an access. The compiler classifies these accesses as having locality. It rewrites these accesses to exploit this locality thus improving the performance of the generated code.

To efficiently exploit the locality in the access `f(k)`, we need to bring the elements of `f` into the GPU's shared memory with coalesced accesses from GPU memory. Since successive iterations of the `k` loop access successive elements of the `f` array, we can rewrite the code so that each thread loads one element into shared memory. Thus, neighboring threads can copy successive elements from GPU memory into shared memory. Since the generated code uses thread blocks with 32 threads, 32 elements are loaded into shared memory in one iteration of the `k` loop. This also means the load into shared memory needs to be performed once every 32

iterations of the k loop. In the remaining iterations of the k loop, the threads access f(k) from the shared memory. To accomplish this, the compiler rewrites the code as follows.

```
1  __global__ void kernel(float *x, float *y, float *f, int n, int len)
2  {
3    __shared__ float sharedBuffer[32];
4    int tidx = threadIdx.x + blockIdx.x*blockDim.x;
5    int i = tidx+len;
6    if(i>n) return;
7    float outSample = 0;
8    for (int k=1 ; k<=len ; ++k)
9    {
10     if ((k-1)%32 == 0)
11     {
12       sharedBuffer[threadIdx.x] = f(k + threadIdx.x);
13       __syncthreads();
14     }
15     outSample = outSample + sharedBuffer[(k-1)%32]*x(i-(k-1));
16   }
17   y(i) = outSample;
18 }
```

The listing above does not show code that is needed to handle corner cases[7] for the sake of simplicity. In the modified code, a shared memory buffer sharedBuffer is defined. All threads load elements of f into this buffer and use them while performing the computation. Once every 32 iterations of the k loop, all threads in the block cooperate to load 32 successive elements of f into shared memory. This is done by the code in lines 10–14. A __syncThreads call is needed to ensure that all threads in the thread block have completed the load on line 12 before execution proceeds. Line 15 then uses the values loaded into sharedBuffer for the computation. Therefore, each element of f is loaded exactly once per thread block. Also, these loads are now coalesced loads. Note that the conditional statement on line 10 does not cause divergence as all threads in the thread block have exactly the same value of k.

[7]For example, the generated code needs to handle the case when len is not a multiple of 32. This can be done by bounds checking the index of f before line 12.

The compiler also identifies single accesses that are shared by all threads in a block. This is the case when the array index is identical but not a loop index. The compiler transforms the code so that only the first thread in the block loads the required data into shared memory. Threads then synchronize and use data from the shared memory buffer.

The access `x(i-(k-1))` also has locality that can be exploited. However, identifying and exploiting this automatically in GPU code is more complicated. We leave this for future work.

4.2.2 Memory Access Coalescing

As mentioned in Section 2.1, to exploit the large bandwidth available on the GPU, it is important to ensure accesses to the GPU memory are coalesced. Several lower end GPUs like the GeForce 8800 GTX and the GeForce 9800 GTX have strict coalescing constraints. One of these requirements is that the address the first thread in the warp accesses is on a 64 byte boundary. Consider the following GPU kernel.

```
1 __global__ void kernel(float *x, int n)
2 {
3   int tidx = threadIdx.x + blockIdx.x*blockDim.x;
4   int tidy = threadIdx.y + blockIdx.y*blockDim.y;
5   int i = tidy;
6   int j = tidx;
7   float f = x[i*n + j];
8   // More code ...
9 }
```

The kernel takes two arguments, `x` and `n`. `x` is a 2-dimensional square matrix of floats stored in row major order while `n` is an integer that represents the size of the matrix `x`. When the kernel is executed, successive threads in a thread block get successive values of `j`. Therefore, neighboring threads access successive memory locations.

On the GeForce 8800, the access on line 7 is either coalesced or uncoalesced depending on the value of `n`. Consider the case when `n` has the value 64. Then, the expression `i*n` always has a value that is a multiple of 64 and therefore, the address accessed by the first thread in a thread block is always a multiple of 64 assuming that the base address `x` is on a 64 byte boundary

which is guaranteed by the CUDA runtime API. However, if we consider the case when `n` has the value 65, the address accessed by the thread with `i=1` and `j=0`, which is the first thread in its block, is no longer on a 64 byte boundary.

As another example, consider the following access in MATLAB code.

```
1  X = a( : , 3:end);
```

Our compiler translates the above line of MATLAB to the following kernel.

```
1 __global__ void kernel(float *X, float *a)
2 {
3   int tidx = threadIdx.x + blockIdx.x*blockDim.x;
4   int tidy = threadIdx.y + blockIdx.y*blockDim.y;
5   int i = tidy;
6   int j = tidx;
7   X[i*M + j] = a[i*N + 3+j];
8 }
```

Here, `M` and `N` represent the number of columns of `X` and `a` respectively. The access to `a` on line 7 cannot be a coalesced one regardless of the value of `N` because of the offset of 3 in the original array access. This offset needs to be taken into account when deciding whether or not the array access is coalesced. An access meets the alignment requirement if the following condition holds.

$$(size_{cols} + offset).sizeof(elem) \equiv 0(mod64) \tag{4.2.5}$$

$size_{cols}$ is the size of the second dimension of the array being accessed, $offset$ is the start offset of the access in the second dimension (3 in the above example) and $sizeof(elem)$ is the size of each element in the array.

Our compiler addresses the alignment problem for coalesced accesses by transforming the code appropriately. It first identifies array accesses that violate Equation 4.2.5 and transforms them so that the required data is read into shared memory through coalesced accesses. The transformation converts the identified unaligned access into two aligned ones using shared memory. The two aligned accesses read a superset of the data needed by the original unaligned access. The shared memory is used as temporary storage for the data read by the two aligned accesses. Then, the original access is modified to read the required data from the shared memory.

The previous example is transformed into the following code.

```
1  __global__ void kernel(float *X, float *a)
2  {
3    __shared__ float sharedBuffer[64];
4    int tidx = threadIdx.x + blockIdx.x*blockDim.x;
5    int tidy = threadIdx.y + blockIdx.y*blockDim.y;
6    int i = tidy;
7    int j = tidx;
8    int index = i*N + 3+j;
9    int base = index/64 * 64;
10   sharedBuffer[threadIdx.x] = a[base+threadIdx.x];
11   sharedBuffer[threadIdx.x+32] = a[base+threadIdx.x+32];
12   __syncthreads();
13   X[i*M + j] = sharedBuffer[index%64];
14 }
```

Lines 8–11 bring the required data into the shared memory buffer `sharedBuffer` using two coalesced accesses on line 10 and 11. The original access to `a` is rewritten to access the same data from `sharedBuffer`. Accessing the data this way, rather than through uncoalesced accesses is much faster since the memory bandwidth of the GPU is used more efficiently.

There are two reasons for using the method described above rather than padding an array so that the alignment problem is addressed. Firstly, it is possible that there are several accesses to the same array, each having a different indexing offset. Padding the array so that all these accesses are coalesced is extremely hard. Rewriting the accesses as described above is much simpler. Second, padding arrays leads to much higher memory usage and we believe using GPU memory efficiently is important.

In the following section, we present experimental results that show the benefits of the optimization techniques described in this chapter.

4.3 Experimental Evaluation

The optimizations described in this chapter were implemented in the MEGHA compiler. We first measure the performance gains due to the optimizations discussed in Section 4.2. Next, we investigate the overheads introduced by the GPU memory management transformation discussed in Section 4.1.

As before, the CUDA code generated by our compiler was compiled using `nvcc` (version 2.3) with the optimization level -O3. The generated executables were run on a machine with an Intel Xeon 2.83GHz quad-core processor and a GeForce 8800 GTS 512 graphics processor. We also ran these on a machine with the same CPU but with a Tesla S1070 [47]. Both machines run Linux with kernel version 2.6.26. We inserted calls to the `gettimeOfday` function to measure the runtimes of executables generated by MEGHA and used the MATLAB function `clock` to measure the runtime for MATLAB programs run in the MATLAB environment.

For each benchmark, we report the runtime in the MATLAB environment (referred to as baseline MATLAB), and the execution time of the original and optimized C++ plus CUDA code generated by our compiler (referred to as CPU+GPU and CPU+GPU(Opt) respectively). Note that CPU+GPU code refers to code generated as described in Chapter 3 which already includes several optimization. Thus, CPU+GPU(Opt) here means that the additional optimizations described in this chapter are performed. Each benchmark was run three times. The reported runtime is the average execution time per run. In all cases, we found the variation in execution times to be negligible.

4.3.1 Data Reuse

To assess the benefits of the data reuse optimization (described in Section 4.2.1, we measured runtimes for two benchmarks, `filter` and `MatrixMul`, with and without the optimization. Both of these benchmarks have accesses that our compiler can optimize. There are no opportunities to perform the data reuse optimization in the other benchmarks. In the `filter` benchmark, the compiler automatically detects each of the accesses to the filter coefficients in the filter implementation as candidates for locality optimization. In the `MatrixMul` benchmark,

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	CPU+8800	CPU+8800(Opt)
filter	4×256 tap, 1 M	11.5	2.15	0.98
	4×256 tap, 1.5 M	17.42	3.22	1.47
	4×128 tap, 1 M	5.95	1.15	0.53
	4×128 tap, 1.5 M	8.92	1.76	0.79
MatrixMul	1024×1024	16.2	1.06	0.18
	2048×2048	580	7.4	1.15

Table 4.1: Effect of the data reuse optimization on the 8800

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	CPU+Tesla	CPU+Tesla(Opt)
filter	4×256 tap, 1 M	11.5	0.3	0.16
	4×256 tap, 1.5 M	17.42	0.45	0.23
	4×128 tap, 1 M	5.95	0.17	0.11
	4×128 tap, 1.5 M	8.92	0.23	0.16
MatrixMul	1024×1024	16.2	0.16	0.14
	2048×2048	580	0.94	0.84

Table 4.2: Effect of the data reuse optimization on the Tesla

the access to successive elements elements of each row of the matrix is identified as a candidate for locality optimization. This access is not coalesced by the loop reordering transformation described in Section 3.2.2.

Table 4.1 lists the runtimes of generated code for the two benchmarks with and without the data reuse optimization described in Section 4.2.1 when the generated code is run on the machine with the GeForce 8800 GTS. The optimized CPU+GPU code for the benchmark `filter` is about 2X faster than the original CPU+GPU code generated by our compiler, which itself was 5X faster than baseline MATLAB execution. The optimized code is up to 11.8X faster than baseline MATLAB execution.

For the `MatrixMul` benchmark, the optimized code is more than 5X faster than the original CPU+GPU code, which itself is 35X faster than baseline MATLAB execution. Thus, the overall geometric mean speedup of the optimized code over the CPU+GPU code is 3.1X on the GeForce 8800.

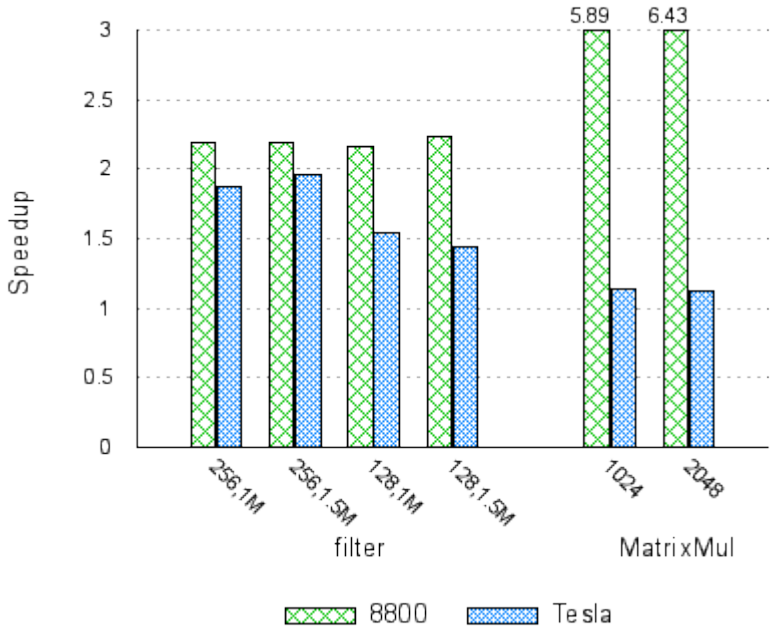


Figure 4.4: Speedup over code generated by baseline MEGHA due to the data reuse optimization

Table 4.2 lists the runtimes for runs on the machine with the Tesla S1070. On the Tesla, optimized code for the `filter` benchmark is 1.5X-1.9X faster than CPU+GPU code. For the `MatrixMul` benchmark, optimized code is only 20% faster than CPU+GPU code. Overall, the geometric mean speedup of optimized code over CPU+GPU code on the Tesla is 1.5X. Even though this speedup is significant, it is not as dramatic as the speedup obtained on the 8800 because the coalescing constraints on the Tesla are much weaker than on the 8800. Memory accesses to the same location by multiple threads of a warp are coalesced on the Tesla while this is not the case on the 8800. This makes the effect of the locality optimization much more pronounced in the 8800. Figure 4.4 shows a plot of the speedups of optimized code over CPU+GPU code for both benchmarks for various input sizes.

4.3.2 Memory Access Coalescing

In this section, we report performance measurements to quantify the benefits of the coalescing transform described in Section 4.2.2. The benchmarks `fdtd` and `MatrixMul` have accesses that are uncoalesced as they do not meet alignment requirements (Section 4.2.2). As observed

Benchmark	Size	Execution Time (in Seconds)	
		CPU+8800	CPU+8800(Opt)
fdtd	$512 \times 512 \times 15$	10.1	3.9
	$512 \times 512 \times 32$	22.7	8.4
MatrixMul	1025×1025	0.77	0.38
	1537×1537	2.51	1.17

Table 4.3: Effect of the memory access coalescing optimization on the 8800

in Section 3.3, there is a significant difference in the performance of the generated code for the `fdtd` benchmark on the Tesla as compared to the 8800. This is primarily due to uncoalesced accesses.

Table 4.3 lists the runtimes of generated code with and without the coalescing optimization. The data reuse optimization described in Section 4.2.1 is also performed while generating both versions of the code (with and without the coalescing optimization). For the benchmark `fdtd` the optimized code is faster by a factor of 2.5X while for `MatrixMul` the optimized code is about 2.1X faster than baseline CPU+GPU code on average. The geometric mean speedup of optimized code over baseline code across both benchmarks is 2.3X. As mentioned in Section 4.2.2, this optimization is for lower end GPUs like the 8800 and is not beneficial while targeting the Tesla because of the weaker constraints the Tesla has for coalescing.

4.3.3 GPU Memory Management

In this section, we investigate the effects of the GPU memory management transformation described in Section 4.1. To do this, we consider three benchmarks `filter`, `fdtd` and `nb3d`. We use input sizes that are large enough that the generated code cannot be run on the GeForce 8800 without the transformations described above. While compiling these benchmarks, we restrict the LVS size to 425 MB[8].

Table 4.4 lists the runtimes of baseline MATLAB execution, the CPU-only code generated

[8] We choose a value slightly smaller than 512 MB because of a known problem in the NVIDIA memory allocator (`http://stackoverflow.com/questions/4750293/problems-with-cudamalloc-and-out-of-memory-error`). The value of 425 MB was determined empirically.

Benchmark	Size	Execution Time (in Seconds)			
		MATLAB	CPU	CPU+8800	CPU+8800(Naïve)
nb3d	3872	1076.9	324.3	155.8	261
fdtd	560 × 560 × 64	1001.5	67.2	52.7	98.7
filter	4×256 tap, 32 M	370.6	61.7	31.7	31.9

Table 4.4: Effect of the GPU memory management transform on the 8800

Benchmark	Size	Candidates	Spilt
`nb3d`	3872	14	3
`fdtd`	560 × 560 × 64	7	3
`filter`	4×256 tap, 32 M	9	6

Table 4.5: Number of spill candidates and number of spills

by our compiler and the runtime of the transformed CPU+GPU running on the machine with the GeForce 8800. To estimate the benefits of intelligent spilling, we compare our transformation with a naïve scheme. In the naïve scheme, before a kernel is executed, we transfer variables required by the kernel to GPU memory, and transfer modified variables back to CPU memory after the kernel finishes executing. This ensures that the LVS size is the same as the working set size and never exceeds available GPU memory. The runtimes of code generated using this naïve scheme is also listed in Table 4.4. Table 4.5 lists the number of spill candidates that were identified in each case and how many of them were spilt.

For the benchmark `nb3d`, the CPU-only code is about 3X faster than baseline MATLAB execution. The transformed CPU+GPU code, however, is about 2X faster than the CPU-only version which makes it 6X faster than baseline MATLAB. On the other hand, the code generated with the naïve scheme is only about 20% faster than CPU-only code. Thus, the code generated using our GPU memory management strategy is about 1.7X faster than code generated using the naïve technique described above. In the case of `fdtd`, the CPU-only code is about 15X faster than baseline MATLAB execution. The transformed CPU+GPU code is 19X faster than baseline MATLAB execution which means that it is about 30% faster than CPU-only execution. In this benchmark, the amount of computation performed in each kernel is not extremely large.

Therefore, introducing additional transfers has an exaggerated effect on performance. This is demonstrated by the fact that the naïve code is about 1.9X slower than the CPU+GPU code and is also about 1.4X slower than the generated CPU-only code. For the benchmark `filter`, the CPU-only code is 6X faster than baseline MATLAB and the CPU+GPU code is close to 12X faster than baseline MATLAB. However, in this case, the naïvely generated code performs similarly to the code generated using our GPU memory management technique. This is because the fraction of execution time spent performing transfers is very small in this benchmark due to the large amount of computation. Overall, the generated CPU+GPU code, with the GPU memory management transform, is 1.7X faster than CPU only code. The generated code also out-performs the naïve scheme by 1.5X on average. These results demonstrate that generating code to use the GPU can yield performance gains compared to only using the CPU, even when spilling is required, if the variables that are spilt are carefully chosen.

We also measured the total time spent performing memory transfers when code was generated using our GPU memory management technique and when code was generated using the naïve scheme. Figure 4.5 plots the total transfer time when the naïve scheme is used normalized to the total transfer time when our scheme (discussed in Section 4.1) is used. For the benchmarks `fdtd` and `nb3d`, our technique yields more than 2X reduction in the amount of data transfer. In the case of `filter`, our scheme yields a 30% reduction in the time spent performing memory transfers. Across all three benchmarks, using our technique to generate code results in a 2X reduction in the total amount of data transfer time compared to the naïve scheme.

4.4 Summary

The basic compiler described in Chapter 3 does not take GPU memory constraints into account while scheduling kernels on the GPU. It does not take into account the fact that just ensuring that the working set of each kernel fits in available GPU memory is not sufficient to correctly execute generated code on the GPU. We first explained why this is the case using a motivating example. Subsequently, we show that the problem of determining whether or not generated

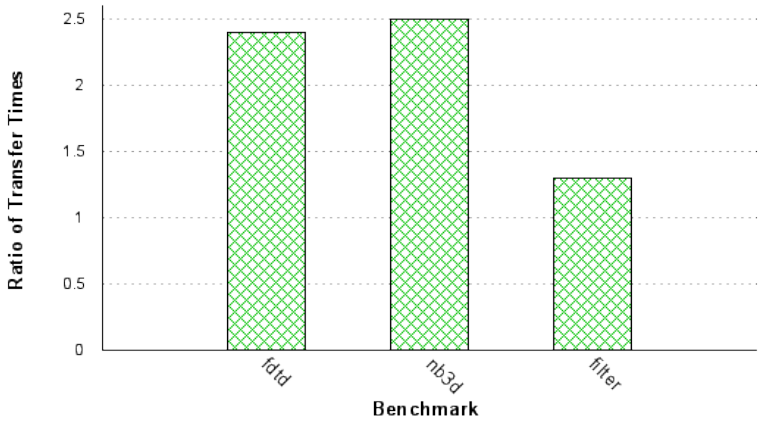


Figure 4.5: Reduction in total data transfer time using our GPU memory management scheme (Section 4.1) compared to the naïve scheme.

code obeys GPU memory constraints is equivalent to the problem of computing all maximal cliques of an interference graph. Since this problem is NP-hard, we simplify it by reducing the interference graph to an interval graph and then leverage the linear scan algorithm to efficiently determine whether generated code meets the required constraints. We also describe heuristics for deciding what variable to spill. Our experiments show that code generated using our scheme performs 1.7X better than CPU only code on average and outperforms a naïve scheme by 1.5X on average.

Additionally, we describe two optimizations that target memory accesses in GPU code. The first optimization identifies array references that access the same memory locations across all threads in a warp. It rewrites these accesses so that the required value is read only once into shared memory and all threads use this value. This optimization yields a 3.1X performance improvement on the GeForce 8800 and 1.5X improvement on the Tesla. The second optimization is for accesses that are not coalesced because they do not meet alignment requirements. The shared memory is used as temporary storage so the the required data can be read using multiple coalesced accesses. This optimization yields a 2.5X performance gain on the GeForce 8800.

Chapter 5

Related Work

In this chapter, we discuss prior work that is related to the work described in this thesis. We first discuss prior work on GPU programming frameworks in Section 5.1. Section 5.2 describes earlier work on optimizing compilers for GPUs. Finally, we discuss prior work on the execution of MATLAB programs in Section 5.3.

5.1 GPU Programming Frameworks

In the early days of General Programming on GPUs (GPGPU), programmers had to program the graphics pipeline explicitly [55]. The code to be run on the GPU was written in shader languages like Cg [26] and this code was run on the GPU through graphics APIs like OpenGL [67] or DirectX [50]. However, GPUs have since evolved into more programmable devices and programming environments like CUDA [51] and OpenCL [42] have become popular. CUDA is a proprietary programming environment to program NVIDIA GPUs. It provides a `C` like abstraction to write programs for GPUs and library functions for various operations like DMA transfers. OpenCL is an open standard that provides an interface similar to CUDA. However, OpenCL programs can be run on many different data parallel architectures.

Several systems that try to raise the level of abstraction provided to programmers have been built. One of the earliest such systems was Brook [14]. Brook allowed users to program platforms with GPUs in `C++` by providing them with a *stream* abstraction. In Brook, computation

is expressed in the form of data parallel kernels and data is passed between these kernels as streams. Communicating data through streams abstracts the details of the underlying system from the programmer.

The Accelerator framework [69] is a .NET library that uses object orientation to provide programmers with a higher level of abstraction. It defines a structure called the *data parallel array* and defines several operations on these arrays. It also provides methods to convert between normal arrays and data parallel arrays. Operations on data parallel arrays are automatically performed on the GPU by lazily generating and executing GPU shader code. Even though Accelerator provides an elegant mechanism to program GPUs, it still relies on the programmer for information on what operations to run on the GPU. This requires the programmer to have some understanding of the underlying system.

Previous works have also explored compiling programs in other high level languages to GPUs. For example, compilation of stream programs to GPUs and heterogeneous machines was studied by Udupa et al [71][72]. They describe a compiler to automatically compile StreamIt [70] programs with only stateless filters for execution on a GPU in a pipelined manner [71]. They also describe how more general programs can be compiled for pipelined execution across CPU cores and GPUs [72]. Their compilation methodology makes extensive use of integer linear programming and profiling to automatically assign work to either the CPU or GPU. Another example of a system that compiles a high level language to GPUs is Copperhead [15]. In Copperhead, the source language is a restricted subset of Python [58]. The compiler performs type inference on the input Python code and then generates GPU code for the data parallel parts.

Lee et al [45] have developed a compiler that converts OpenMP [53] programs to CUDA. Their compiler automatically constructs kernels from parallel loops in the input code and generates CUDA kernels for these. The compiler also automatically inserts required data transfers and performs a few optimizations on the generated CUDA code. This compiler however, maps all parallel regions in the OpenMP program to the GPU.

5.2 Optimizing Compilers for GPUs

Several previous works have studied problems related to generating optimized kernel code for GPUs. Techniques for optimizing naïve CUDA kernels were studied by Yang et al [77]. The input to their compiler is a simple kernel that performs a single unit of data parallel work per GPU thread. The compiler then applies a series of transformations so that the memory hierarchy is used more efficiently. It also groups threads into thread blocks to improve data reuse in registers and shared memory. CUDA-lite [73] is another system that tries to optimize a simple kernel. It uses static analysis and programmer annotations to infer information about various memory accesses in the input kernel and tries to optimize them. The use of the polyhedral model [24, 25] to optimize code for accelerator architectures was studied by Baskaran et al [10, 11]. They use the polyhedral model to schedule iterations of a parallel loop nest so that accesses are coalesced and also to infer what variables can reside in shared memory. These techniques are complementary to those discussed in this thesis as they can be used to further optimize kernels and low level code generated by our compiler.

5.3 MATLAB Compilers and Runtimes

Several previous projects have studied the problem of compiling MATLAB. The first of these was the FALCON project that compiled MATLAB programs to FORTRAN [22]. Most of our frontend is based on the work done in the FALCON project. MaJIC [4] is a just-in-time MATLAB compiler that compiled and executed MATLAB programs with a mixture of compile time and run-time techniques. The MAGICA project [37] focussed on statically inferring symbolic types for variables in a MATLAB program and establishing relationships between shapes of various variables. Mat2C [38] is a MATLAB to C compiler based on MAGICA. MATCH [31] is a virtual machine that executes MATLAB programs in parallel by mimicking superscalar processors. The possibility of using extensive compile time analysis for generating high performance libraries from MATLAB routines for execution on CPUs was studied by Chauhan et al [18]. More recently, McVM [19], a virtual machine for the execution of MATLAB programs, which performs specialization based on types has been developed. However, none of

these projects studied automatically compiling MATLAB to machines with distributed memory or heterogeneous processors as is done in our work.

Jacket [35] and GPUMat [28] are systems that enable users to execute parts of MATLAB programs on GPUs. However, these require users to specify what is to be run on the GPU by annotating the source code. Arrays whose operations are to be performed on the GPU are declared with a special type. Similarly, Khoury et al extend Octave to target the Cell BE processor [41]. However, their system still requires users to cast matrices to a specific type so that operations on these matrices can be performed on the SPEs. It also uses a virtual machine based approach to perform matrix operations on the SPEs as opposed to generating custom code. The MATLAB Parallel Computing Toolbox [56] provides users with the ability to run computations on clusters. More recently, support for running computation on the GPU has also been added. However, the user is still required to cast arrays to a particular type in order to run computation on the GPU as is the case with GPUmat and Jacket.

Parallel to our work, Shei et al have developed a source-to-source MATLAB compiler [66] that automatically transforms an input MATLAB program to use the GPUmat library. Their compiler uses a Binary Integer Linear Programming (BILP) approach to map statements to either the GPU or CPU. Their compiler also performs mapping at the level of a MATLAB basic block. They, however, do not perform any kernel composition or custom CUDA code generation for statements that are mapped to the GPU. Also, they use a dynamic approach to ensure that dependences across basic block boundaries are satisfied correctly whereas our compiler uses a dataflow analysis to statically insert the required data transfers.

Chapter 6

Conclusions

In this chapter, we summarize the work described in this thesis (Section 6.1) and subsequently discuss possible directions for future work (Section 6.2).

6.1 Summary

In this thesis, we presented a compiler framework, MEGHA, which automatically compiles MATLAB programs for execution on systems with heterogeneous processors. Our compiler automatically identifies parts of the input program that are data parallel and maps these to the GPU while using the CPU to execute the control flow dominated parts of the program.

In order to achieve this, the compiler first simplifies the input source code and constructs an intermediate form suitable for analysis and optimization. It then constructs kernels that represent data parallel regions in the input source code. It takes into account constraints like memory and register usage while constructing kernels. Parallel `for` loops (DOALL loops) in the input program are also identified as kernels by our compiler. In the process of constructing kernels, the compiler is able to eliminate several unnecessary array variables.

Having identified kernels, the compiler maps each kernel to either the CPU or the GPU. We propose a profile-driven heuristic based on list scheduling to map kernels to either the CPU or GPU. This heuristic takes the required data transfers into account while making decisions on where to run each kernel. It also inserts the data transfers required to satisfy local dependences

(dependences within the same MATLAB basic block). Since mapping and scheduling are done at a MATLAB basic block level, dependences across MATLAB basic blocks need to be correctly satisfied. We propose a simple data flow analysis based edge splitting strategy to handle such dependences.

To optimize the generated code, we propose a simple heuristic method to decide the order in which the iteration space of each kernel needs to be traversed. Our method computes different orders depending on whether the kernel is executed on the CPU or the GPU. It maximizes cache utilization when a kernel is executed on the CPU and maximizes the number of coalesced accesses when a kernel is executed on the GPU.

We also identify the problem of managing when variables are live in GPU memory. We formulate this problem as that of finding all maximal cliques of an interference graph. As this problem is NP-hard, we simplified it by approximating the interference graph as an interval graph and designed an efficient algorithm to solve it. Finally, we described two additional methods to optimize memory accesses in GPU code. The first tries to exploit the data reuse across different GPU threads by bringing data shared by multiple threads into the GPUs shared memory. The second coalesces memory accesses that are not coalesced because they do not meet alignment requirements. It accomplishes this by bringing the required data into shared memory using multiple coalesced accesses.

We have prototyped the proposed system using the Octave [23] system. Our experiments using this implementation show a geometric mean speedup of 12X on the GeForce 8800 GTS and 29.2X on the Tesla S1070 over baseline MATLAB execution. Experiments also reveal that our method provides up to 10X speedup over hand written GPUmat versions of the benchmarks. Our method also provides a speedup of 5.3X on the GeForce 8800 GTS and 13.8X on the Tesla S1070 compared to compiled MATLAB code running on the CPU.

6.2 Future Work

As described in Section 2.4, the current compiler makes certain assumptions about the input program. An interesting direction for future work would be to try and remove these constraints

so that a larger class of programs can be compiled. One restriction that can be relaxed is the requirement that arrays are not indexed past their end. There has been past work by Chauhan et al that addresses this problem [17]. It is likely that their work can be adapted and used in the GPU context. Also, as described in Section 3.1.3, we currently use a simple mechanism to infer types of variables. In order to enable the compilation of more general programs, our type system and type inference mechanism need to be extended to support symbolic types [37]. Extending the type system would have implications on other parts of the compiler like kernel identification and these would need to be suitably modified as well.

The dynamic nature of MATLAB makes the problem of statically compiling MATLAB code a hard one. However, more type information can be inferred at run-time. Therefore, we believe that building an execution environment that uses a combination of static compilation and dynamic compilation will be an interesting direction to pursue. With this in mind, we have designed the algorithms described in this thesis to be as simple as possible so that they are applicable in a dynamic setting as well. However, several additional details need to be worked out in order to successfully design and build such a system.

Another direction of future work would be improving the mapping and scheduling algorithm proposed in Section 3.2.3. It would be useful to have a global scheduling algorithm that can take GPU memory constraints into account. Designing an algorithm to solve this problem would be an interesting direction to pursue.

Bibliography

[1] S. V. Adve and K. Gharachorloo. *Shared Memory Consistency Models : A Tutorial*. IEEE Computer, pp 66–76, December 1996.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 1st Edition (October 2001).

[3] A. V. Aho, Ravi Sethi, J. D. Ullman, M. S. Lam. *Compilers: Principles, Techniques, & Tools*. Pearson Education, 2009.

[4] G. Almasi, D. Padua. *MaJIC: Compiling MATLAB for Speed and Responsiveness*. In the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02), pp 294–303, Berlin, Germany, June 2002.

[5] Advanced Micro Devices Inc., *AMD Compute Abstraction Layer Programming Guide*.

[6] ATI Technologies, *http://ati.amd.com/products/index.html*

[7] ATI Technologies, *http://fusion.amd.com*

[8] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, W. W. Hwu. *An Adaptive Performance Modeling Tool for GPU Architectures*. In the Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), pp 105–114, Bangalore, India, February 2010.

[9] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers 1988.

[10] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan. *A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs*. In the 22nd Annual International Conference on Supercomputing (ICS '08), pp 225–234, Island of Kos, Greece, June 2008.

[11] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan. *Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories*. In the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), pp 1–10, Salt Lake City, UT, USA, February 2008.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan. *A Practical Automatic Polyhedral Parallelizer and Locality Optimizer*. In the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08), pp 101–113, Tucson, AZ, USA, June 2008.

[13] P. Briggs, K. D. Cooper, and L. Torczon. *Improvements to Graph Coloring Register Allocation*. ACM Transactions on Programming Languages and Systems, pp 428-455, May 1994.

[14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: Stream Computing on Graphics Hardware*. ACM Transactions on Graphics, Vol. 23, pp. 777–786, 2004.

[15] B. Catanzaro, M. Garland, and K. Keutzer. *Copperhead: Compiling an Embedded Data Parallel Language*. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11), pp 47–56, San Antonio, TX, USA, February 2011.

[16] G. Chaitin. *Register Allocation and Spilling Via Graph Coloring*. SIGPLAN Not. 39, 4, pp 66–74, April 2004.

[17] A. Chauhan and K. Kennedy, *Slice-hoisting for Array-size Inference in MATLAB*, In the 16th International Workshop on Languages and Compilers for Parallel Computing 2003 (LCPC '03), pp 495–508, College Station, TX, USA, October 2003.

[18] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. *Automatic Type-Driven Library Generation for Telescoping Languages*. In the 2003 ACM/IEEE Conference on Supercomputing (SC '03), pp 51–57, Phoenix, AZ, USA, November 2003.

[19] M. Chevalier-Boisvert, L. Hendren, C. Verbrugge. *Optimizing MATLAB Through Just-In-Time Specialization*. In the 2010 International Conference on Compiler Construction (CC '10), pp 46–65, Paphos, Cyprus, March 2010.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd Edition (September 2001).

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In the ACM Transactions on Programming Languages and Systems, pp 451–490, October 1991.

[22] L. De Rose, D. Padua. *Techniques for the Translation of MATLAB Programs into Fortran 90*. In the ACM Transactions on Programming Languages and Systems, pp 286–323, March 1999.

[23] J. W. Eaton. *GNU Octave Manual*, Network Theory Limited, 2002.

[24] P. Feautrier. *Some Efficient Solutions to the Affine Scheduling Problem: One-Dimensional Time*. International Journal of Parallel Programming, pp 313–348, October 1992.

[25] P. Feautrier. *Some Efficient Solutions to the Affine Scheduling Problem: Multi Dimensional Time*. International Journal of Parallel Programming, pp 389–420, December 1992.

[26] R. Fernando, M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional (2003).

[27] W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. *Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow*. In the Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40), pp 407–420, Chicago, IL, USA, December 2007.

[28] GPUmat Home Page. *http://gp-you.org/*

[29] The Global Power Community, *http://www.power.org/*

[30] D. Hackenberg, D. Molka, and W. E. Nagel. *Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems*. In the Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42), pp 413-422, New York, NY, USA, December 2009.

[31] M. Haldar et. al. *MATCH Virtual Machine: An Adaptive Run-Time System to Execute MATLAB in Parallel*. In the 2000 International Conference on Parallel Processing (ICPP '00), pp 145–155, Toronto, Canada, August 2000.

[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, Fourth Edition.

[33] J. Hensley. *AMD CTM Overview*. In ACM SIGGRAPH 2007 courses (SIGGRAPH '07).

[34] S. Hong and H. Kim. *An Analytical Model for a GPU Architecture with Memory-Level and Thread-level Parallelism Awareness*. In the Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09), pp 152–163, Austin, TX, USA, June 2009.

[35] Jacket Home Page. *http://www.accelereyes.com/*

[36] P. Joisha, P. Banerjee. *Static Array Storage Optimization in MATLAB*. In the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03), pp 258–268, San Diego, California, USA, June 2003.

[37] P. Joisha, P. Banerjee. *An Algebraic Array Shape Inference System for MATLAB*. ACM Transactions on Programming Languages and Systems, pp 848–907, September 2006.

[38] P. Joisha, P. Banerjee. *A Translator System for the MATLAB Language*, Research Articles on Software Practices and Experience '07, pp 535–578, April 2007.

[39] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. J. Shippy. *Introduction to the Cell Multiprocessor*. In the IBM Journal of Research and Development, pp 589-604, July 2005.

[40] K. Kennedy, K. S. McKinley. *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution*. In the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC '93), pp 301–320, Portland, OR, USA, August 1993.

[41] R. Khoury, B. Burgstaller, B. Scholz, *Accelerating the Execution of Matrix Languages on the Cell Broadband Engine Architecture*. IEEE Transactions on Parallel and Distributed Systems, pp 7–21, January 2011.

[42] The Khronos Group, *http://www.khronos.org/opencl/*.

[43] C. Kim, D. Burger, S.W. Keckler, *Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches*. IEEE Micro, Volume 23, Number 6, pp. 99- 107, November 2003.

[44] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. *COMIC: A Coherent Shared Memory Interface for CellBE*. In the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08), pp 303–314, Toronto, Canada, September 2008.

[45] S. Lee, S. Min, and R. Eigenmann. *OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization*. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09), pp 101–110, Raleigh, NC, USA, February 2009.

[46] C. G. Lekkerkerker, and J. C. Boland. *Representation of a Finite Graph by a Set of Intervals on the Real Line*. Fund. Math. 51, pp 45–64, 1962.

[47] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. IEEE Micro, March 2008.

[48] Mathworks Home Page. *http://www.mathworks.com/*

[49] Microsoft Corporation, *.NET Framework Reference Information*. URL:http://msdn.microsoft.com/en-us/library/t7yeed4c%28v=vs.80%29.aspx.

[50] Microsoft Corporation, *Direct3D 11 Reference*. URL:http://msdn.microsoft.com/en-us/library/windows/desktop/ff476147(v=vs.85).aspx

[51] NVIDIA Corp, *NVIDIA CUDA: Compute Unified Device Architecture: Programming Guide*, Version 3.0, 2010.

[52] NVIDIA Corp, Fermi Home Page, *http://www.nvidia.com/object/fermi_architecture.html*

[53] The OpenMP Architecture Review Board, *OpenMP Application Program Interface*. Version 3.1, July 2011.

[54] A. V. Oppenheim, and R. W. Schafer. *Digital Signal Processing*. Prentice Hall, January 1975.

[55] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, T. J. Purcell. *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, Volume 26, Issue 1, pp 80–113, March 2007.

[56] MATLAB Parallel Computing Toolboc. *http://www.mathworks.in/products/parallel-computing/*

[57] M. Poletto and V. Sarkar. *Linear Scan Register Allocation*. In the ACM Transactions on Programming Languages and Systems, pp 895–913, September 1999.

[58] Python Programming Language - Official Website, *http://www.python.org/*.

[59] The Global Power Community, *http://www.power.org/*

[60] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. *Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA*. In the Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), pp 73–82, Salt Lake City, UT, USA, February 2008.

[61] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. *Program Optimization Space Pruning for a Multithreaded GPU*. In the Proceedings of the sixth annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08), pp 195–204, Boston, MA, USA, April 2008.

[62] K. Sano, O. Pell, W. Luk, and S. Yamamoto. *FPGA-based Streaming Computation for Lattice Boltzmann Method*. In the 2007 International Conference on Field-Programmable Technology (FPT 2007), pp 233–236, Kitakyushu, December 2007.

[63] K. Sano, T. Iizuka, and S. Yamamoto. *Systolic Architecture for Computational Fluid Dynamics on FPGAs*. Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), pp 107–116, Napa, CA, USA, April 2007.

[64] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. *Larrabee: A Many-Core x86 Architecture for Visual Computing*. ACM Trans. Graph. 27, 3, Article 18, pp 18:1–18:15, August 2008.

[65] S. Seo, J. Lee and Z. Sura.*Design and Implementation of Software-Managed Caches for Multicores with Local Memory*, In the 15th International Symposium on High Performance Computer Architecture (HPCA 2009), pp 55–66, Raleigh, NC, USA, February 2009.

[66] C. Shei, P. Ratnalikar and A. Chauhan. *Automating GPU computing in MATLAB*. Proceedings of the International Conference on Supercomputing (ICS '11), pp 245–254, Tucson, Arizona, USA, June 2011.

[67] D. Shreiner, M. Woo, J. Neider, T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional, 6th edition, August 2007.

[68] S. K. Singhai, K. S. Mckinley. *A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality*, Computer Journal, 1997.

[69] D. Tarditi, S. Puri, J. Oglesby. *Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses*. In the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp 325–335, San Jose, CA, USA, October 2006.

[70] W. Thies, M. Karczmarek, and S. P. Amarasinghe. *StreamIt: A Language for Streaming Applications*. In the Proceedings of the 11th International Conference on Compiler Construction (CC '02), pp 179–196, Grenoble, France, April 2002.

[71] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil. *Software Pipelined Execution of Stream Programs on GPUs*. In the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09), pp 200–209, Seattle, WA, USA, March 2009.

[72] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil. *Synergistic Execution of Stream Programs on Multicores with Accelerators*. In the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09), pp 99–108, Dublin, Ireland, June 2009.

[73] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu.*CUDA-Lite: Reducing GPU Programming Complexity*. In Languages and Compilers for Parallel Computing 2008 (LCPC '08), pp 1–15, Edmonton, Canada, July 2008.

[74] V. Volkov, J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. In the 2008 ACM/IEEE Conference on Supercomputing (SC '08), pp 31:1–31:11 , Austin, TX, USA, November 2008.

[75] D. B. West. *Introduction to Graph Theory*. Englewood Cliffs, NJ: Prentice-Hall 2000.

[76] L. Wu, C. Weaver, and T. Austin. *CryptoManiac: A Fast Flexible Architecture for Secure Communication*. In the 28th annual International Symposium on Computer Architecture (ISCA '01), pp 110–119, Goteborg, Sweden, June 2001.

[77] Y. Yang, P. Xiang, J. Kong, H. Zhou. *A GPGPU Compiler for Memory Optimization and Parallelism Management*. In the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '10), pp 86-97, Toronto, Canada, June 2010.