EXPLOITING SHARED-MEMORY REUSE THROUGH SOURCE-LEVEL TRANSFORMATION OF CUDA KERNELS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the degree

Master of SCIENCE

by

Swapneela P. Unkule,B.E.

San Marcos,Texas
December 2011

EXPLOITING SHARED-MEMORY REUSE THROUGH SOURCE-LEVEL

TRANSFORMATION OF CUDA KERNELS

Committee Members Approved:

Apan Qasem, Chair

Martin Burtscher

Khosrow Kaikhah

Xiao Chen

Approved:

J. Michael Willoughby
Dean of the Graduate College

FAIR USE AND AUTHORS PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

It is a pleasure to thank many people who made this thesis possible.

First and foremost I offer my sincere gratitude to my advisor Dr. Apan Qasem. His guidance helped me in all the time of research and writing of this thesis. With his enthusiasm, inspiration, and great efforts to explain things clearly and simply, he helped to make this study successful. His wide knowledge and logical way of thinking have been of great value for me. I could not have imagined having a better advisor and mentor.

I wish to express sincere thanks to my committee members Dr. Burtscher, Dr. Kaikhah and Dr. Chen for their encouragement and insightful comments. I appreciate their support. I am grateful to all the members of Computer Science department for assisting me in many different ways. I am indebted to my student colleagues and friends for providing a stimulating and fun environment to learn and grow. I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

My special gratitude is due to my sister, brother-in-law and my dearest niece Nishigandha who were always there to encourage me. I wish to thank my parents. They raised me, supported me, taught me and loved me. To them I dedicate this thesis. Lastly, and most importantly, the one above all of us, the omnipresent God, for answering my prayers and giving me strength. Thank you so much Dear Lord.

This manuscript was submitted on October 28, 2011.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Concern over power consumption and heat dissipation has triggered a fundamental shift in processor design. The trend of packing more transistors into smaller space, driven by Moore's law, has been replaced by the multicore design paradigm, where multiple simplified cores are integrated into a single chip. The multicore design favors on-chip parallelism over increasingly faster processor speed and makes it imperative that parallelism is achieved at multiple levels for achieving scalable high-performance. Since parallelism is not automatic, the shift towards multicore processors also implies that the software has more responsibility in extracting and exploiting the available parallelism.

Along with the multicore processors, the newest emerging technology to support parallelism is GPU computing. GPU computing is the use of graphics processing units to do general purpose scientific and engineering computing. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. GPUs are massively multithreaded many-core chips with hundreds of cores and thousands of concurrent threads. The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability, has made graphics hardware a compelling platform for computationally demanding tasks. As seen from Figure 1, GPU performance growth has been substantially higher that CPU performance growth in the last few years. Also, GPUs have a wide variety of application domains ranging from numeric computing operations, data mining, fluid dynamics, bioinformatics to physical simulations and much more. As the programmability of GPUs increases, the application domains are likely to

increase as well. Although the current power consumption of GPUs as a unit is still high, because of their enormous compute power their FLOPS-per-watt ratio is much superior to conventional CPUs [24]. Moreover, the cost per GFLOP for GPUs is an order of magnitude less than that of mainstream supercomputers. These features have made GPUs a highly attractive platform for the HPC community.

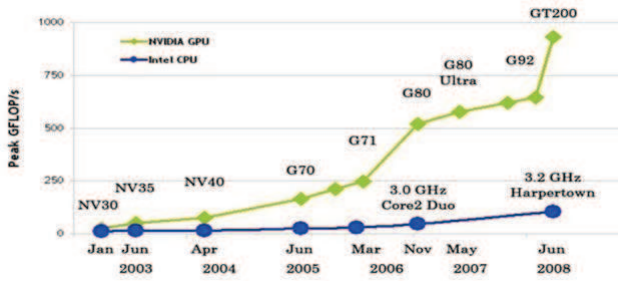

Figure 1: Performance growth of NVIDIA's GPUs vs Intels CPUs [5]

Despite the fact that GPUs have dramatically increased the performance potential of computing systems, achieving a high fraction of peak on these platforms still remains a major challenge. GPUs are much more specialized than conventional CPUs and hence are not applicable to as wide a range of HPC applications. Implementation of any algorithm other than very regular streaming applications requires significant programming effort and careful orchestration of thread granularity parameters to extract a sufficient amount of parallelism. Thus, many of the recent research efforts in GPU computing have focused on implementing and tuning specific applications or classes of algorithms [11, 18, 24, 26, 31]. As a result, the body of literature on general strategies for optimizing for GPUs is fairly thin. Although many CPU-centric optimizations can be effective for GPUs, at least in principle, to date there have been few studies that have attempted to do a careful evaluation of CPU optimizations that can be used on GPU [19, 27].

To fully realize the power of general purpose computation on GPUs, two key issues need to be considered. First how to parallelize an application into concurrent work

items and distribute the workloads in a hierarchy of thread blocks and threads; and second, how to efficiently utilize the GPU memory hierarchy, given its dominant impact on performance. The memory subsystem on the GPU is structured somewhat differently than its CPU counterpart. Effectively utilizing the memory system on the GPU is particularly challenging because of the division of memory into various subspaces including global, local, constant, shared and texture memory. Improved memory performance depends not only on exploiting locality at higher levels of memory but also the placement of data in the different subspaces [30].

GPU programming gives the programmer flexibility in deciding the number of threads running per block and setting up a limit on the amount of registers and/or shared memory used in a given kernel. For making full use of GPU capabilities it is necessary to strike the right balance between each thread's resource usage and number of simultaneously active threads.

This research investigates software strategies for better utilization of memory resources on GPUs. Specifically, we focus on how controlling thread granularity can lead to better utilization of the memory hierarchy and improved performance. Each multiprocessor contains one set of local 32-bit registers with size up to 32kB and shared memory up to 48kB. Since a multiprocessor's registers and shared memory are split among all the threads of block, the number of blocks a multiprocessor can process at once depends on the number of registers and the amount of shared memory required per block for a given kernel. Therefore, to achieve high performance on GPUs, it is necessary to understand the vital role of on-chip resources and their interplay with other components of the architecture. Reducing the number of threads running in a block implies each thread has access to a larger number of registers and is less likely to incur bank conflicts while accessing shared memory. Bank conflicts occur when multiple requests are made for data from the same bank. On the other hand, having fewer threads per block will result in lower occupancy. Occupancy on GPUs

is defined as the ratio of the number of active *warps* and the maximum allowed warps on a multiprocessor and is a good indicator of the amount of achieved parallelism. In general, it is advisable to run large number of threads in order to achieve higher performance. This is because a large number of threads leads to higher occupancy which opens up more opportunities for latency hiding, resulting in higher throughput. However, it has also been shown that running fewer threads per block, although leading to lower occupancy, can improve overall performance [30]. This performance improvement is attributed to fewer accesses to shared memory and better utilization of registers per thread. In other studies, it has been shown that increased register pressure per thread can lead to lower occupancy and huge slowdowns in performance because of accesses to global memory [7, 29].

Register and shared memory usage is directly influenced by thread granularity or the amount of work done per thread. Thread granularity of CUDA programs can be controlled by a source-level transformation called *thread coarsening*. Thread coarsening can be used to determine the amount of work done per thread and help in exploiting inter-thread data reuse.

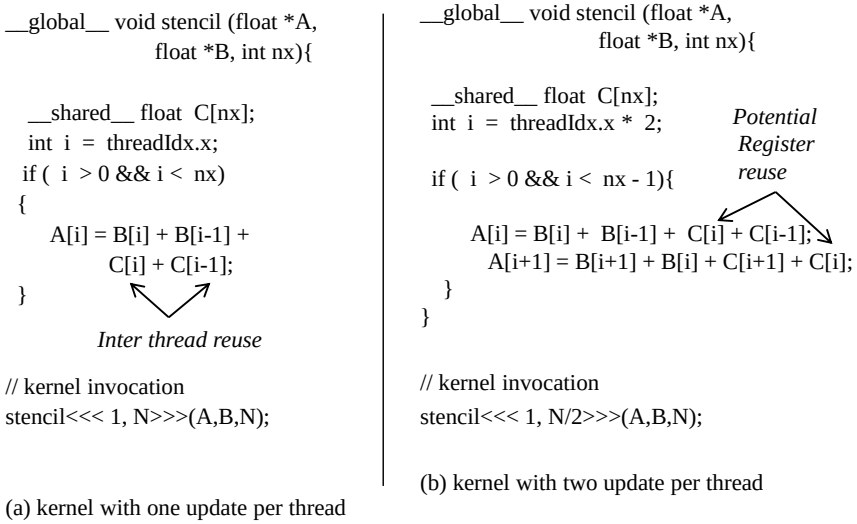

Figure 2: Inter thread locality in stencil computation

We use a simple example to illustrate the interaction between data locality and thread granularity and discuss its performance implications. Consider the CUDA kernel shown in Figure 2(a). In this kernel, all threads are organized in *one* single-dimensional thread block and each thread computes one element of array A, based on elements in B and C. Arrays A and B are allocated in global memory whereas C is allocated in shared memory. Because the computation is based on neighboring elements in B and C, the kernel exhibits temporal reuse in both shared and global memory. Values in B and C, accessed in thread i are reused by thread $i+1$. Although the register pool on a multiprocessor is shared among threads in a warp, the distribution of registers occurs before thread execution and, hence during execution a thread cannot access registers that belong to a co-running thread. This implies that inter-thread data reuse will remain unexploited in the version of code shown in Figure 2(a). In Figure 2(b), the kernel is transformed to perform two updates per thread and is invoked with half as many threads as the original version, thereby increasing thread granularity. This variant converts inter-thread reuse of data elements in B and C into *intra*-thread reuse, allowing the compiler to allocate the values $B[i]$ and $C[i]$ into registers of thread i, leading to better register reuse. Thus, the coarsening transformation shown in Figure 2(b) can potentially reduce shared memory traffic by 25%. In the CUDA programming model, however, employing a fewer number of threads for the same computation generally implies a lower warp occupancy. For the kernel in Figure 2, if we assume 8 registers per thread and 512 threads per block, then for a GPU with Compute Capability (CC) 2.0, we would have a 100% occupancy. The occupancy remains at a 100% when we reduce thread count to 256. However, it falls to just 67% when the thread count is reduced to 128 because only 8 blocks can be mapped to an Streaming Multiprocessor simultaneously. Thus, for this kernel, executing two updates per thread will almost inevitably lead to performance gains but any further coarsening will have to be weighed against the cost of the reduced occupancy. The other performance consideration in this context is register pressure. Increasing thread granularity can potentially increase the number of required registers. This can not only lead to spills and cause more accesses to global memory but also

have an impact on occupancy. If we assume a register count increase of two per coarsening factor for the example kernel then, for a factor of 12, the number of required registers is 32, which drops the occupancy down to 33%. Therefore, all three factors inter-thread data locality, register pressure and occupancy, need to be considered for profitable thread coarsening.

To understand the impact of thread coarsening on CUDA performance, this research develops CREST, a source-to-source transformation that automatically applies thread coarsening to CUDA programs. CREST takes a CUDA file as an input which is passed through kernel extractor which extracts the kernel, along with the thread block size and the coarsening factor provided as a pragma. I have develop the analysis required to determine the legality and profitability of coarsening. I also develop an analytical model to estimate the register pressure for a thread at the source level. I have provided experimental results on a Fermi GPU to evaluate the effectiveness of our proposed strategy.

CHAPTER 2: BACKGROUND

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA Corp. It is designed to explore computing on GPUs with a simple user interface, eliminating the need to understand the complex graphics interface. CUDA is accessible to software developers through variants of industry standard programming languages. Using CUDA, the latest GPUs become accessible for computation like CPUs. CUDA C programming involves running code on two different platforms: a host system that relies on one or more CPUs to perform calculations, and the device, which is one or more CUDA-enabled NVIDIA GPUs. Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads. In response to this difference, CUDA extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. CUDA threads are parallel portions of

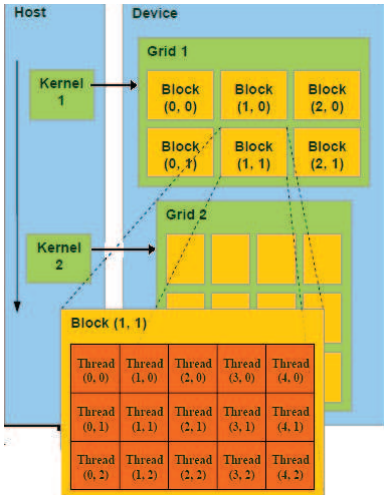

Figure 3: CUDA thread hierarchy [12]

an application that are executed on the device as kernels. One kernel is executed at a time and many threads execute each kernel. The C runtime for CUDA handles kernel loading and preparing kernels before they are launched. CUDA context management, kernel configuration, and parameter passing are all performed by the CUDA runtime. Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. The kernel functions typically generate a large number of threads to exploit data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid. At the top level of the hierarchy, a grid is organized as a one or two dimensional array of blocks as shown in Figure 3. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. A thread block is a batch of threads that can co-operate with each other. At the bottom level of the hierarchy, all blocks of a grid are organized into an upto three dimensional array of threads. Each thread and block have an ID that they use to compute memory addresses and make control decisions.

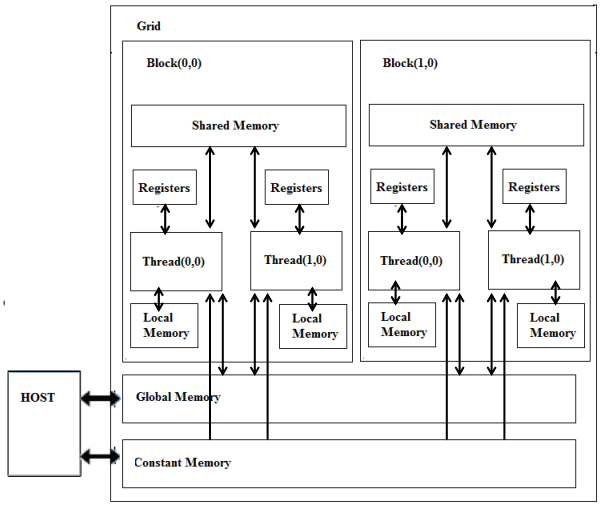

Figure 4: CUDA memory hierarchy

CUDA threads may access data from multiple memory spaces during their execution as shown in Figure 4. Each thread has a private local memory in DRAM. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. Threads within a

block cooperate via shared memory. Threads in different blocks cooperate through global memory.The fastest memory available on GPU is the register space. Registers are accessible for read and write, local to each thread. The number of registers assigned to a thread is rounded up to a multiple of four [1]. Table 1 summarizes different memory access patterns.

Table 1: Memory access pattern

Memory	Location	Cached	Access	Scope
Register	On-Chip	No	Read-Write	One thread
Local	Off-Chip	No*	Read-Write	One thread
Shared	On-Chip	N/A	Read-Write	All threads in a block
Global	Off-Chip	No*	Read-Write	All threads + host
Constant	Off-Chip	Yes	Read-Only	All threads + host

* cached in L1 on Fermi

Any source file containing CUDA language extensions must be compiled with NVCC. NVCC is a compiler driver that simplifies the process of compiling CUDA code. It provides simple and familiar command line options and executes them by invoking a collection of tools that implement the different compilation stages.

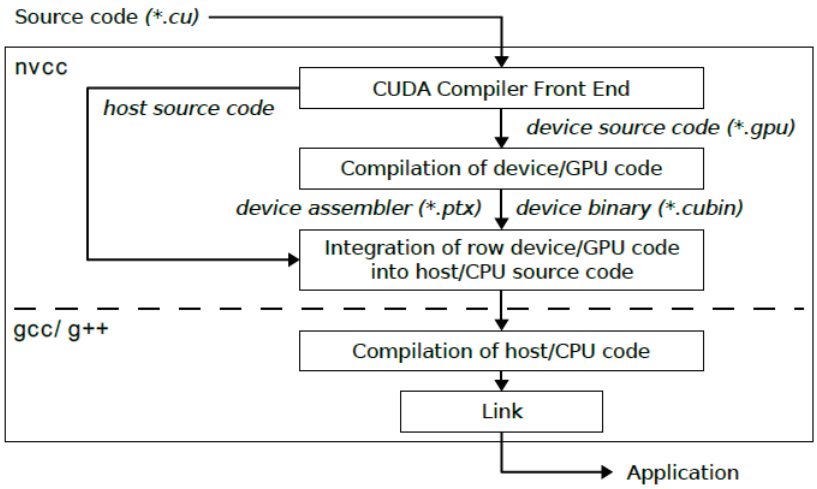

Figure 5: Compilation flow with NVCC [9]

Figure 5 shows NVCC's basic workflow, which consists of separating device code from host code and compiling the device code into a binary form or cubin object. The generated host code is output either as C code that is left to be compiled using another tool or as object code directly by invoking the host compiler during the last compilation stage.

CHAPTER 3: RELATED WORK

Because general-purpose computing on GPUs is a fairly new idea and the technology is still maturing, much of the software-based performance improvement strategies have been limited to manual optimization. Ryoo et al. [25] present a general framework for optimizing applications on GPUs. Their proposed strategies includes utilizing many threads to hide latency, and using local memories to alleviate pressure on global memory bandwidth. Govindaraju et al. develop new FFT algorithms for the GPUs and hand optimize the kernels to achieve impressive performance gains over the CPU-based implementation [16]. The key transformation used in their work was the combining of transpose operations with FFT computation. Demmel and Volkov [30] manually optimize the matrix multiplication kernel and produce a variant that is 60% faster than the autotuned version in CUBLAS 1.1. Among the optimization strategies discussed in this work are the use of shorter vectors at program level and the utilization of the register file as the primary on-chip storage space.

There has been some work in combining automatic and semi-automatic tuning approaches with GPU code optimization. Murthy et al.have developed a semi-automatic, compile time approach for identifying suitable unroll factors for selected loops in GPU programs [21]. The framework statically estimates execution cycle count of a given CUDA loop and uses the information to select optimal unroll factors. Liu et al. [34] propose a GPU adaptive optimization framework (GADAPT) for automatic prediction of near-optimal configuration of parameters that affect GPU performance. They take unoptimized CUDA code as input and traverse an optimization search space to determine optimal parameters to transform the unoptimized input CUDA code into optimized CUDA code. Choi et al. present a

model-driven framework for automated performance tuning of sparse matrix-vector multiply (SpMV) on systems accelerated by GPU [11]. Their framework yields huge speedups for SpMV for the class of matrices with dense block substructure, such as those arising in finite element method applications. Williams et al. have also applied model-based autotuning techniques to sparse matrix computation that have yielded significant performance gains over CPU-based autotuned kernels [32]. Nukada and Matsuoko also provide a highly-optimized 3D-FFT kernel [23]. Work on autotuning general applications on the GPU is somewhat limited. Govindaraju et al. propose autotuning techniques for improving memory performance for some scientific applications [15] and Datta et al. apply autotuning to optimize stencil kernels for the GPU [14]. The MAGMA project has focused on autotuning dense linear algebra kernels for the GPU, successfully transcending the ATLAS model to achieve as much as a factor of 20 speedup on some kernels [22]. Grauer-Gray and Cavazos present an autotuning strategy for utilizing the register and shared memory space for belief propagation algorithms [17].

Automatic approaches to code transformation has been mainly focused on automatically translating C code into efficient parallel CUDA kernels. Baskaran et al. present an automatic code transformation system (PLUTO) that generates parallel CUDA code from sequential C code, for programs with affine references [8]. The performance of the automatically generated CUDA code is close to hand-optimized CUDA code and considerably better than the benchmarks' performance on a multicore CPU. Lee et al. [20] take a similar approach and develop a compiler framework for automatic translation from OpenMP to CUDA. The system handles both regular and irregular programs, parallelized using OpenMP primitives. Work sharing constructs in OpenMP are translated into distribution of work across threads in CUDA. However, the system does not optimize data access costs for access in global memory and also does not make use of on-chip shared memory.

With some different approach compared to the above work a novel optimizing

compiler for general purpose computation on GPU is developed [33], which addresses two major challenges, effective utilization of the GPU memory hierarchy and judicious management of parallelism. This compiler accepts a naive GPU kernel function as an input. Then it analyzes the code, identifies its memory access patterns, and generates both the optimized kernel and the kernel invocation parameters. To generate the optimized kernel, the compiler performs optimizations like thread/thread-block merge to enable data reuse, grouping of memory access to vector data accesses, converting non-coalesced memory accesses into coalesced accesses, data prefetching and eliminating partition camping. Similar to this work, a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPUs is presented by Manikandan et al. [7]. This work focuses on three significant performance influencing factors: efficient global memory access, shared memory access and reduction of the dynamic instruction count. The optimization techniques used include memory coalescing as well as model-driven empirical search for optimal tile size and unroll factors,thereby providing a new approach for compiler optimization on GPU.

The work presented in this thesis distinguishes itself from earlier work in two ways. First the focus here is on automatic compiler methods rather than manual optimization techniques. Second, the approach supports direct optimization of CUDA source rather than C or OpenMP variants. Our research is similar to the compiler developed for generating optimized kernels. I am using the thread coarsening transformation for exploiting inter-thread locality. I do not claim that this approach is superior to the approaches proposed previously. Rather, my framework can be used in conjunction with many of the strategies mentioned in this section.

CHAPTER 4: OVERVIEW OF FRAMEWORK

Figure 6 gives an overview of the code restructuring and tuning framework of CUDA kernel, develop for this thesis. The framework leverages several existing tools, including `nvcc` for compiling CUDA kernels and cudaprof for collection of performance metrics. The kernel extractor and code restructurer have been developed from scratch. We describe these tools next.

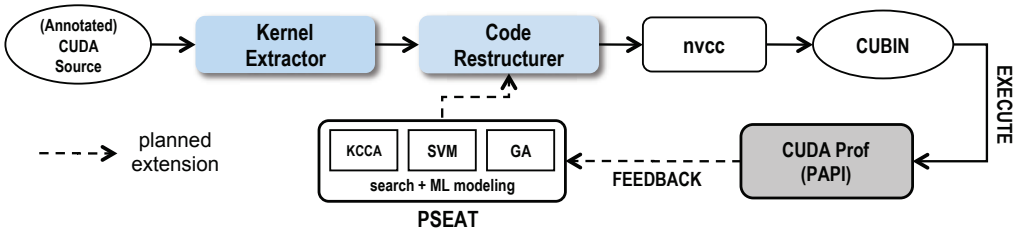

Figure 6: Overview of CREST

4.1 Kernel Extraction

To facilitate analysis, a standalone Perl script is used to extract the kernel from the CUDA source file before parsing. This simplifies the parser by setting aside everything external to the kernel being analyzed. The extraction is performed on a line-by-line basis, using regular expressions to detect the kernel-specific portion of the source file. The kernel is extracted into a separate file for further processing and everything else is held in temporary files for later reassembly. The kernel extraction is designed to be independent of the succeeding phases and can be used in other applications where CUDA kernels need to be examined.

4.2 Code Restructuring with CREST

At the heart of the framework is a source-to-source code transformation tool (CREST) that analyses the CUDA kernels and implements the thread coarsening transformation, among others. A key feature of this tool is that it provides fine-grain control over optimizations through the use of source code directives. With CREST, transformations can be applied at the kernel level with parameters provided by the user. There are three parameters required for the coarsening transformation. The first parameter is the file name. The second parameter is the coarsening factor and the third parameter is the block size as shown in Figure 7. The coarsening factor and block size specified using pragma values are extracted by the front-end and then supplied as input parameters to the main coarsening routine.

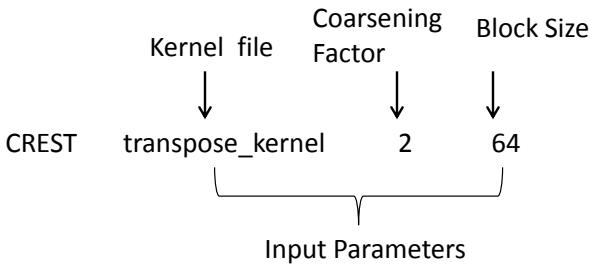

Figure 7: CREST usage

4.3 Performance Analysis

Many performance metrics used for tuning applications on CPUs are also relevant for GPU-based tuning. There are several metrics, however, that hold special significance on GPUs and there are some metrics that are not as pertinent. For instance, register spill count may have less significance on GPUs, since typically the total number of registers available on the GPU is much more than the number of registers available on the CPU. On the other hand, because GPUs rely on having a large number of active warps for latency hiding, metrics such as occupancy have special significance. On GPU platforms, the relationship between code transformations and different aspects of performance is yet to be established.

Currently, we are manually observing the different performance metrics values measured by the CUDA profiler and we use them in deciding the right coarsening factor that will produce improved performance over the baseline. After every transformation. we execute the kernel and check the cuda_profile.log file created by CUDA profiler. Then we manually compare the execution time required by the kernel for different coarsening factors and select the one with the lowest execution time. In the future the performance metrics will be fed directly into PSEAT to allow for autotuning of coarsening factors.

CHAPTER 5: USER-GUIDED THREAD COARSENING

In this section, we describe the thread coarsening transformation and discuss the analysis needed to apply this transformation safely and profitably.

5.1 Notation and Terminology

We introduce the following notation and terminology to describe our transformation framework.

N	number of *simple* high-level statements in kernel
T	number of threads in thread block
s_i	i^{th} static statement in kernel
$synch_i$	`syncthreads` primitive, executed as i^{th} statement in kernel
$s_i \succ s_j$	data dependence from s_i to s_j
$S_{(i,p)}$	a statement *instance*: i^{th} statement in kernel, executed by p^{th} thread
$S_{(i,p)} \succ S_{(j,q)}$	a dependence between statement instances $S_{(i,p)}$ and $S_{(j,q)}$, where $S_{(i,p)}$ is the *source* and $S_{(j,q)}$ is the *sink* of the dependence
C_x	coarsening factor along dimension x

5.2 Dependence Analysis

The goal of our dependence analysis framework is to determine if there is a dependence between two statement *instances*. We require that the input to our framework is a CUDA program that is legally parallelized for execution on a GPU. We make a assumption that, in the absence of `syncthreads()` primitives (i.e., barrier synchronization), statement instances that belong to different threads (or thread blocks) are independent. Here we are not taking into account the lock step execution of threads in same warp. Thus, given this framework we can make the following claim:

if $\nexists$ *syncthreads() primitives in the kernel body then*

$$\nexists S_{(i,p)} \succ S_{(j,q)}, \forall i, j \in \{N\} \text{ and } \forall p, q \in \{T\}$$

To detect and estimate inter-thread data locality, the analyzer needs to consider *read-read* reuse of data, which may or may not occur between statement instances, regardless of the parallel configuration. For this reason, we extend our dependence framework to handle input dependencies. Our framework handles the following two cases of dependence between two statement instances:

(i) $\exists\ S_{(i,p)} \succ S_{(j,q)}$*, iff* $S_{(i,p)}$ *and* $S_{(j,q)}$ *access the same memory location*

(ii) $\exists\ S_{(i,p)} \succ S_{(j,q)}$*, iff* $\exists\ synch_k$ *such that* $k < j$ *or* $i < k$

Conventional dependence analysis [6] can be applied to statements within the body of a kernel to determine if statements access the same memory location. For CUDA kernels, one issue that complicates the analysis is that memory accesses can be dependent on the value of the *thread ID*. For this reason, a subscript analysis of the source may show two statements as accessing the same memory location e.g. the references a[i] and a[i] but, if i depends upon thread id then they would access different locations. To handle this situation we take the following strategy we first identify all statements in the kernel that are dependent on thread ID values; we expand index expressions to replace subscripts

with thread ID values (using scalar renaming [13]) and then apply the subscript test on the expanded expressions. Once all data dependencies have been identified, our dependence analyzer makes another pass to identify dependencies that arise from the presence of `syncthreads()`. This final pass mainly involves checking for the existence of condition (ii) mentioned above.

5.3 Safety Analysis

For simplicity, we only describe the analysis necessary to safely apply thread coarsening along the innermost dimension, x. The same principles can be applied, in a relatively straightforward manner, for coarsening along the y dimension and also for increasing thread block granularity (i.e., fusing two thread blocks).

Two factors determine the legality of the coarsening transformation. One is the relationship between the coarsening factor C_x and the number of threads in the original kernel T, and the other is the presence of coarsening preventing dependencies. For coarsening to be legal, there have to be enough threads available in the original configuration to satisfy the coarsening factor. If the coarsening factor is larger than the original thread count, extra work will be performed in each thread, violating the semantics. Also, when C_x does not evenly divide T, special handling of the remaining threads is necessary, which complicates the transformation and is likely to have a negative impact on overall performance. For this reason, we enforce the constraint that C_x evenly divides T for coarsening to be legal. Thus, the first legality constraint for coarsening is as follows:

$$T \; mod \; C_x = 0 \tag{1}$$

A dependence between two statement instances will cause coarsening to be illegal if, as a result of coarsening, the direction of the dependence is reversed. We refer to such a dependence as a *coarsening preventing dependence (cpd)* and derive the following

conditions under which a *cpd* will *not* occur when the coarsening factor is C_x.

$$\nexists\, S_{(i,p)} \succ S_{(j,p-q)},\ where\ i,j \in \{N\},\ p \in \{C_x+1,\ldots,T\},\ q \in \{1,\ldots,C_x\} \quad (2)$$

or

$$\forall S_{(i,p)} \succ S_{(j,p-q)},\ where\ i,j \in \{N\},\ \ p \in \{C_x+1,\ldots,T\},\ q \in \{1,\ldots,C_x\}$$

$$\nexists\, S_{(k,p)} \succ S_{(j,p-q)},\ where\ k \in \{j+1,\ldots,N\} \quad (3)$$

Constraint (2) describes the situation where we have no dependence between statement instances within the coarsening range. Note, we are only concerned about dependencies that emanate from a *higher* numbered thread. For the coarsening transformation, dependencies that emanate from a *lower* numbered thread are irrelevant, since, by default, in the merged code body after transformation, all statement instances in p get executed after the last statement in q, where $p > q$. Thus, all such dependencies will be preserved automatically. Constraint (3) considers the case where there is a dependence within the coarsening range but we can avoid violating this dependence if, in the merged thread body, we can move the source statement instance above the sink of the dependence.

5.4 Detecting Inter-thread Locality

A CUDA kernel exhibits inter-thread data locality if two threads in the same thread block accesses the same location, either in shared memory or global memory. Generally, scalars are allocated to registers within each thread and hence coarsening does not help with reuse of such values. Thus, we focus on array references, which are typically *not* allocated to registers by the compiler. Also, on the Fermi chip, it is possible for two threads to access a memory locations that map to the same cache line.

Given this framework, an array reference in the kernel can only exhibit either *self-temporal* or *group-temporal* inter-thread data reuse. Self-temporal reuse can only oc-

cur if no subscript in the array reference depends on any of the *thread ID* variables. If the subscripts are not dependent on thread ID variables it implies that, all threads in the thread block will access the same memory location for that reference. Thus, identifying self-temporal reuse is simply a matter of inspecting each array reference and determining if the subscript values are independent of thread ID values.

To compute group-temporal reuse, we introduce the notion of thread independent dependence. There is a thread independent dependence between two references if it can be established that there is a dependence between the two references when the entire kernel executes as a single thread (i.e., executes sequentially). The advantage of using thread independent dependencies is that their existence can be determined by using conventional dependence tests. Once group-temporal reuse has been established between two references M_1 and M_2 in a thread independent way, we determine if the locality translates to inter-thread locality when the task is decomposed into threads. For inter-thread reuse to exist, at least one subscript in either reference has to be dependent on the thread ID value. This implies that, although M_1 and M_2 access the same memory location, the access may occur from two different threads. We formally, define the presence of inter-thread reuse as follows.

There is inter-thread data reuse in kernel K if

(i) there exists an array reference A with subscripts $i_0, \ldots, i_n$ in K such that no $i \in \{i_0, \ldots, i_n\}$ is a function of the thread ID value

(ii) there exists thread independent dependence between array reference M_1 and M_2, and at least one subscript in M_1 or M_2 is an affine function of the thread ID

5.5 Code Transformation

The goal of the coarsening transformation is to restructure the kernel to perform more *work* in each thread. In essence, for a coarsening factor CF ($CF > 1$), we want thread i to execute statements in thread $(i + 1)$ through $(i + CF - 1)$. This can be achieved by introducing a loop in the kernel body that iterates CF times. Of course, the main challenge is in determining what statements are included in the body of the loop and how the memory references need to be adjusted to affect the change. Figure 8 shows parts of a CUDA kernel in its original and thread coarsened form. Here, BS represents block size while CF represents the coarsening factor. In the coarsened version, a loop is added around the core computation to execute CF times. The variable $\_\_i$ is used to store the value of the current thread id. This variable is incremented by BS/CF during each iteration of the loop, ensuring that the correct location in array as is accessed.

```
__global__ void kernel1(int *in) {

    __shared__ float as[];
     . . . .
    sum = as[threadIdx.x] + 1;
    . . . .
}
```

(a) before

```
__global__ void kernel1(int *in){
    __shared__ float as[];
    int __i = threadIdx.x;
    for(int _k=0;_k<CF; _k++, __i+=(BS/CF)){
          sum = as[__i] + 1;
           . . . .
     }
}
```

(b) after

Figure 8: Simple thread coarsening

As mentioned in Section 5.3, the presence of `__syncthreads()`, which acts as a barrier synchronization, complicates the coarsening transformation. We identify three separate cases related to `__syncthreads()` that need to be handled by our algorithm:

(i) `__syncthreads()` *is not present in the kernel:*

This is the simple case that corresponds to constraint (2), derived in Section 5.3. So we assume that there are no dependencies between statement in different threads. Therefore, in this case, we only need to insert the loop and adjust the memory references (as shown in Figure 8).

```
__global__ void kernel2(int *in)
{
as[threadIdx.x] = in[index];
__syncthreads ();
sum = as[threadIdx.x] +
            as[threadIdx.x+2 ];
. . . .
}
```

(a) before

```
__global__ void kernel2(int *in){
 int __i = threadIdx.x;
   for(int _k = 0; _k < CF;
     _k++,__i+=(BS/CF)){
          as[__i] = in[index];
   }
 __syncthreads ();
   __i = threadIdx.x;
   for(int _k = 0; _k < CF;
          _k++,__i+=(BS/CF)){
          sum = as[__i] + as[__i+2]
          . . . .
 }
}
```

(b) after

Figure 9: Thread coarsening in the presence of `syncthreads()`

(ii) `__syncthreads()` *is present but is not control-dependent on any loop:*

Figure 9 depicts the scenario where a `__syncthreads()` primitive is present in the kernel but the primitive does not appear inside a loop. In this case, to increase thread granularity, we can insert the loop and then *distribute* it around the `__syncthreads()` statement. The distribution ensures that the barrier synchronization is preserved, as it forces all statements controlled by the synchronization directive to execute *before* the `__syncthreads()` statement. The value of $\_\_i$ needs to be reinitialized at the point of distribution to ensure correct memory reference by statements in the second loop. The case where there is a dependence from the part of the code above the `__syncthreads()` to the part following it and the value in the dependence is a function of the thread ID needs special handling. In this situation, we need to change the scalar variable to an array with size equal to CF while distributing the loop. This allows for all values computed in the top loop to be saved and thus preserves the dependence across the two loops.

(iii) `__syncthreads()` *is present and is control-dependent on some loop:*

```
__global__ void kernel3 (float* A) {

    __shared__ float as[] , sum;
    . . . .
  for (int i = 0; i <size; i++) {
      int v1 = as[threadIdx.x] ;
      __syncthreads();
      sum += as[i];
    }
 . . . .
}
```

(a) before

```
__global__ void kernel3 (float* A){

__shared__ float as[],sum;
. . . .
for (int i = 0; i <size; i++){
 int v1 = as[threadIdx.x] ;
   int v2 = as[threadIdx.x+(BS/CF)];
   __syncthreads();
 sum += as[i];
 }
. . . .
}
```

(b) after

Figure 10: Thread coarsening in the presence of `syncthreads()` in loops

If a synchronization primitive appears inside a loop in the kernel, then loop distribution generally results in an illegal transformation. In such a case, we perform (implicitly) an unroll of the loop by the coarsening factor. Figure 10 illustrates this transformation.

Our coarsening algorithm accounts for all three cases mentioned above. The current implementation will detect the third case but the unrolling of the coarsening loop has to be performed by hand. The algorithm for thread coarsening is shown in Algorithm 1.

Algorithm 1 Thread Coarsening Transformation

find all the occurrence of `__syncthreads()` in the kernel and store in a list
if none of the occurrence of `__syncthreads()` are inside a loop OR list is empty **then**
  get the first reference of the thread Id.
  declare a new variable (i) and assign thread Id value to it.
  add the variable declaration before the first reference of thread Id.
  replace all the occurrences of thread Id with this new variable (i).
  from block size and coarsening factor calculate the increment value (inc) for thread Id.
  create a for loop with iterations equal to coarsening factor and increment the value of i by inc with each iteration.
  add this for loop after the variable declaration i.
  if list is not empty **then**
    repeat
      take reference of next occurrence of `__syncthreads()`
      limit the for loop body before this reference.
      after the reference of `__syncthreads()` reinitialize variable i to thread Id value.
      start a new for loop after reinitialization of i variable.
      check for occurrences of `__syncthreads()` in rest of the code.
    until end of list
  else
    expand loop body untill last statement of kernel and return.
  end if
else
  return from function with error signal.
end if

5.6 Estimating Register Pressure

Both the PTX analyzer [2] and the CUDA profiler [5] can provide fairly accurate per-thread register utilization information. Nevertheless, because we apply the code transformations on CUDA source, we require a method to estimate register pressure at the source level. To this end, we developed a register pressure estimation algorithm based on the strategy proposed by Carr and Kennedy [10]. An outline of our algorithm is given in Algorithm 2. Our approach operates on the source code AST. The basic idea is to identify references and variables that will be stored in registers. If there are multiple references with register level reuse, we predict that the compiler will coalesce them into a single register. Also, we are not considering live ranges for the register.

Algorithm 2 Estimating Register Pressure

```
Create AST of the kernel.
repeat
    Read each node in AST.
    if node is of type variable declaration then
        Add the variable name to the global array declared for storing all distinct variables in kernel.
        if variable name and data type match with any of the kernel input variables then
            set the scope as global.
        else if variable name is preceded by 'shared' keyword then
            set scope as shared.
        else if variable name is preceded by 'constant' keyword then
            set scope as constant.
        else if variable name is preceded by 'local' keyword then
            set scope as local.
        else
            set the scope as register.
        end if
    else if node type is assignment then
        if RHS has only one term and that term is a variable then
            save the reference of position of RHS variable in global array with LHS variable.
        end if
    end if
until end of AST
```

CHAPTER 6: EXPERIMENTAL RESULTS

6.1 Experimental Setup

All experiments are performed on a Tesla C2050 NVIDIA Fermi GPU. The card has compute capability 2.0 and consists of 448 cores divided among 14 multiprocessors. The number of 32-bit registers allocated to each multiprocessor is 32,768 and the amount of shared memory available per block is 16 KB OF 48KB. For register estimation experiments we have also used a GeForce 9800GT GPU. The card has compute capability 1.1. The GPU consists of 112 cores divided among 14 multiprocessors. The number of 32-bit registers allocated to each multiprocessor is 8192, and the amount of shared memory available per block is 16KB. All CUDA programs are compiled with `nvcc` version 3.2, C programs are compiled with GCC 4.1.2. The CUDA kernels used in transformation experiments are described below:

Table 2: List of kernels used for experiments

Kernel	Description	Source
stencil	computes sum of neighboring elements within a block	Hand coded
reduce	computes min, max	SC CUDA Tutorial [4]
lintext	demonstrates use of textures bound to pitch linear memory	CUDA SDK 3.2
surfacewrite	demonstrates use of texture fetches in CUDA	CUDA SDK 3.2
transpose	performs transpose of a single-precision matrix	CUDA SDK 3.2

We used the CUDA profiler to measure the performance parameters of each kernel. After transforming a kernel with the required coarsening factor, we compiled it using nvcc and then executed it by turning on the CUDA_PROFILE flag. This flag creates a log file after execution of a program that records the values of parameters listed in the CUDA profile config file. After applying the thread coarsening transformation, some of these kernels did not pass through the front end and we manually had to make some changes to the code.

6.2 Impact on Register Pressure

Although current GPU platforms provide a large number of registers per block [5], it has been shown that for some kernels, ineffective use of the register space can cause significant loss in performance [30].

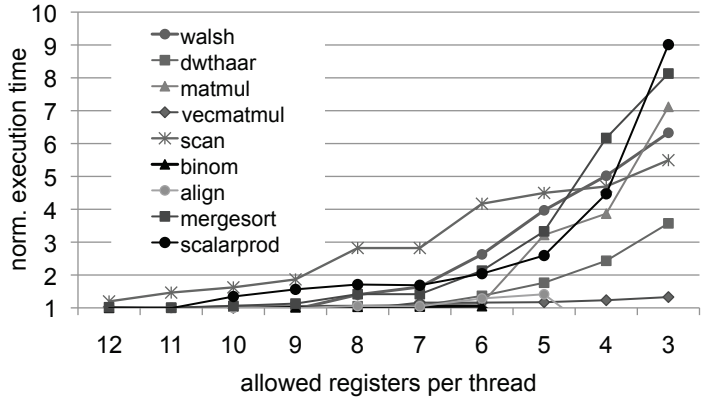

Figure 11: Performance sensitivity to register pressure

To understand how *register pressure* (the ratio between required and available registers) can affect performance, we conducted a simple experiment with a select set of kernels from the CUDA SDK [3]. Since the number of required registers in a thread cannot be modified arbitrarily, we used the *maxregcount* flag in NVIDIA's `nvcc` compiler to control the number of allocated registers, and thereby the register pressure, in each kernel. Figure 11 shows normalized execution times for nine kernels as the register pressure is increased progressively by decreasing the available registers. We observe that, except for matrix-vector multiply (`vecmatmul`), the performance of all kernels is significantly impacted by changes in register pressure. Closer inspection reveals that most of this performance loss is

due to additional accesses to local or shared memory, which can be curbed through better register reuse.

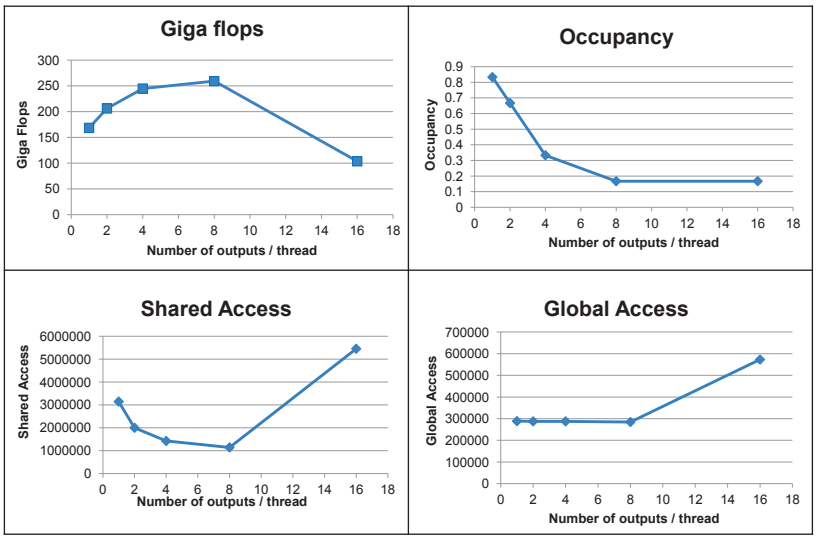

Figure 12: Performance characteristics of matrixmul

Another experiment that we conducted was thread coarsening of the matrix multiplication kernel from the CUDA SDK with few changes in the kernel. The coarsening was done by hand because in this kernel the call to __syncthread is control dependent on a loop, which violates the legality condition of our framework. Figure 12 shows the corresponding results obtained for coarsening factors ranging form 2 to 16. Threads in a block are executed in a group of 32. Hence usually the block size selected is multiple of 32. We required that the Coarsening Factor (CF) chosen should satisfy the equation (BS mod CF = 0) where BS is the block size. So the coarsening factor used in the experiments are multiples of 2. We see that the performance increases up to a CF of 8 and decreases thereafter. The performance increases from 168 Gflops to 259 Gflops, while the number of registers used per thread increased from 21 to 63 for a coarsening factor of 8 over baseline. These results are consistent with earlier work done by Volkov et al. [28].

6.3 Performance Potential

We conducted an experiment to gauge the effectiveness of the proposed strategy under *ideal* circumstances. To this end, we construct a synthetic benchmark `synth` with a high degree of inter-thread reuse. `synth` uses an array with $512K$ elements that is divided into 1024 blocks of size 512. Each thread running in a block computes the sum of all 512 elements residing in the block stored in shared memory. Threads running on different blocks access different elements.

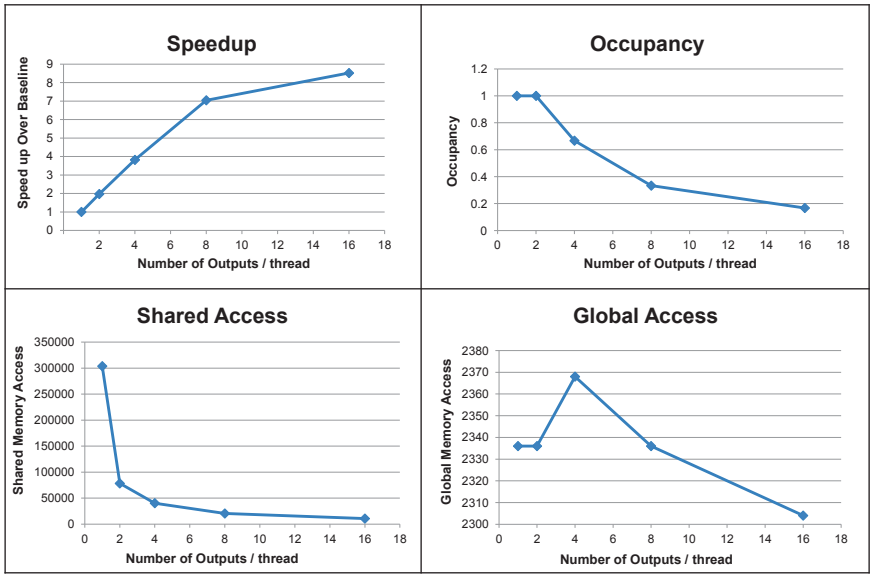

Figure 13: Performance characteristics of `synth`

We ran `synth` with varying coarsening factors from 2 to 16. Figure 13 shows the overall performance, occupancy, and shared and global memory access for `synth` as the coarsening factor is varied. The best speedup of 8.5 is obtained at a coarsening factor of 16. This performance gain is directly attributed to reduced shared and global memory traffic and an total increase of 8 registers per thread for the coarsening factor 16 as compared to baseline. The shared memory traffic never increases as a result of increasing the thread granularity, however, there is a spike in global memory access when going from factor 2 to 4. This spike may be explained by the reduction in occupancy from 1 to 0.67. Thus, the spike indicates a situation where the reduced number of threads causes spills.

Interestingly, the maximum performance is achieved at the lowest occupancy levels (16%), which emphasizes the need for considering factors other than occupancy when optimizing code for GPUs.

6.4 Multi-dimensional Coarsening

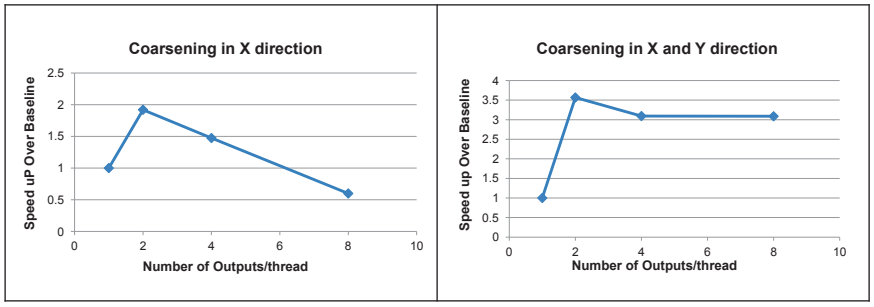

Figure 14: Multi-dimensional coarsening with `transpose`

We evaluate the effects of coarsening along multiple dimensions with `transpose`. The chart on the left in Figure 14 the shows performance of `transpose` for different coarsening factors along the X dimension, while the chart on the right depicts the performance when coarsening is done along both the X and Y dimensions manually. The corresponding transpose kernels are shown in Figure 15. Clearly, for `transpose`, it is more profitable to coarsen along both dimensions, which results in a speedup of 3.5 over the baseline version.

As shown in the transpose kernel the index_out and index_in variables are used for loading and storing different elements from the array. The values of these variables depend on the thread Id values in the X as well as the Y direction. So when we perform coarsening along the X and Y direction, we obtain a larger increase in performance as compared to coarsening in a single direction.We also note that performance degrades (below the baseline) for larger coarsening factors when coarsening along the X dimension. This indicates that there is not enough inter-thread locality along this dimension to outweigh the costs of lower occupancy. Therefore, it is important to consider data locality in other dimensions

```
__global__ void transpose(float *odata float* idata,
                          int width, int height, int n)
{
 int __i = threadIdx.x;

 for(int __j=0;__j<2;__j++,__i+=8){

     int xIndex = blockIdx.x * TILE_DIM + __i;
     int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

     int index_in  = xIndex + width * yIndex;
     int index_out = yIndex + height * xIndex;

     for (int r=0; r < n; r++) {
         for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
             odata[index_out+i] = idata[index_in+i*width];
         }
     }
 }
```

(a) X direction

```
__global__ void transpose(float *odata float* idata,
                          int width, int height, int n)
{
 int __i = threadIdx.x;
 int __k = threadIdx.y;
 for(int __j=0;__j<2;__j++,__i+=8){

     int xIndex = blockIdx.x * TILE_DIM + __i;
     int yIndex = blockIdx.y * TILE_DIM + __k;

     int index_in  = xIndex + width * yIndex;
     int index_out = yIndex + height * xIndex;

     for (int r=0; r < n; r++) {
         for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
             odata[index_out+i] = idata[index_in+i*width];
         }
     }
 }
```

(b) X and Y Direction

Figure 15: Multi-dimensional coarsening with `transpose` kernel

when coarsening.

6.5 Performance Sensitivity

Although thread coarsening helps in improving performance, it is not guaranteed that performance will always increase. Figure 16 shows the speedup observed for the `reduce` kernel with different coarsening factors.

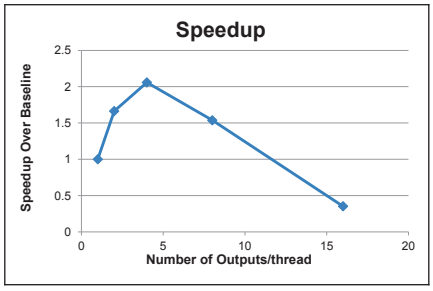

Figure 16: Performance sensitivity for `reduce`

We observe that performance increases when the kernel is coarsened by factors of 2 and 4 but beyond that it starts decreasing rapidly, with factor 16 almost doubling the execution time. On further inspection, we found that this decrease in performance is mainly due to an increase in global memory and shared memory accesses. This indicates that perhaps not enough registers were available to exploit the exposed data reuse. The other factor that contributes to the loss is the lower occupancy.

6.6 Overall Performance

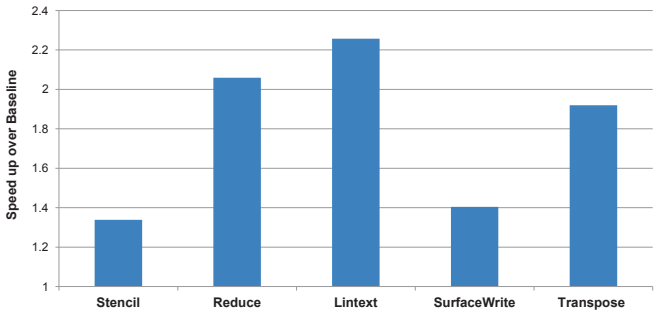

Figure 17: Performance improvement by thread coarsening

Figure 17 shows the speedup obtained for each kernel in our test suite using our strategy. We only selected kernels that had some amount of inter-thread locality. Therefore, it is not too surprising that we observe performance improvement on all five kernels. The more interesting aspect of these results is that not all coarsening factors yielded good performance for all kernels. In fact, for some coarsening factors the performance degraded significantly.

6.7 Register Estimates

Table 3: Register estimation results

Kernel	CREST	CC 1.1	CC 2.0
matrixmul	17	13	21
asyncAPI	2	2	3
concurrentKernel	10	7	8
synth	7	3	12
transpose	9	6	5

In order to test the accuracy of our register pressure algorithm, we applied the algorithm on some kernels and compared the predicted register value with the actual registers used at execution time by passing these kernels through the CUDA profiler. After applying the algorithm on the kernel a file is created that gives a summary about the different

variables used, their probable storage location and finally the estimated number of registers that will be used by that kernel. All kernels used here are baseline kernels and there is no transformation performed on it. Table 3 shows the estimated number of registers per thread in comparison with output produced by the CUDA profiler on compute capability 1.1 (CC 1.1) and 2.0 (CC 2.0) devices.

In case of thread coarsening transformation, we are not seeing a very large increase in the number of registers after coarsening the kernel. However in other transformations like loop unroll, unroll and jam we might see a significant increase in the amount of register for the transformed kernel. So in future we can use the register estimate result in order to select the optimal factor for these transformations.

CHAPTER 7: CONCLUSION AND FUTURE WORK

This thesis describes an automatic approach for controlling thread granularity in GPU kernels. We develop the analysis required to apply the coarsening transformation safely and profitably. The dependence analyzer presented can serve as a framework for implementing a range of memory hierarchy transformations on the GPU. The model for register pressure estimation can be used in developing compiler heuristics. The experimental results depict increased overall performance for kernels that exhibit inter-thread data locality that outweighs the costs of lower occupancy by improving register reuse and reduce memory traffic. These results are preliminary and more extensive experimentation is needed to evaluate the true effectiveness of the proposed method. Nevertheless, the results emphasize the need for considering factors other than occupancy when optimizing code for GPUs.

In the future we aim to combine the register pressure estimation with the thread coarsening model. Also we plan to provide autotuning support for CREST to perform coarsening with different factors and select the optimal factor based on execution time feedback.

BIBLIOGRAPHY

[1] Cuda compiler driver nvcc. http://moss.csc.ncsu.edu/ mueller/cluster/nvidia/2.0/nvcc_2.0.pdf.

[2] CUDA PTX ISA. http://www.nvidia.com/content/CUDAptxisa1.4.pdf.

[3] GPU Computing SDK. http://developer.nvidia.com.

[4] Kernel for min-max and reduction. http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/.

[5] *CUDA Programming Guide, Version 3.0.* NVIDIA, 2010.

[6] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[7] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing, ICS 08*, 2008.

[8] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In R. Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2010.

[9] P. E. Bernard, C. Berthelot, and G. Sauvebois. Integrating hpc and gpu processors. http://www.cse.scitech.ac.uk/disco/mew19/Presentations/BULL_Pierre-EricBernard.pdf.

[10] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[11] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 115–126, New York, NY, USA, 2010. ACM.

[12] Z. Cyril. Tutorial cuda. http://people.maths.ox.ac.uk/ gilesm/hpc/NVIDIA/.

[13] R. Cytron and J. Ferrante. What's in a name? -or- the value of renaming for parallelism detection and storage allocation. In *ICPP'87*, pages 19–27, 1987.

[14] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[15] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM.

[16] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[17] S. Grauer-Gray and J. Cavazos. Optimizing and auto-tuning belief propagation on the gpu. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 121–135, 2011.

[18] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *ACM/IEEE Conf. Supercomputing (SC), Portland, OR, USA, November 2009*, 2009.

[19] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Hardware and software prefetching mechanisms for gpgpu applications. In *IEEE/ACM Intl. Symp. Microarchitecture (MICRO), December 2010*, 2010.

[20] S. Lee, S.-J. 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*, 2009.

[21] G. Murthy, M. Ravishankar, M. Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *IEEE International Symposium on Parallel Distributed Processing*, 2010.

[22] R. Nath, S. Tomov, and J. Dongarra. Accelerating gpu kernels for dense linear algebra. In *In Proceedings of 9th International Meeting on High Performance Computing for Computational Science (VECPAR'10)*, 2010.

[23] A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM.

[24] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-d fft kernel for gpus using cuda. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC 2008*, 2008.

[25] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[26] S. S. Stone, J. Haldar, S. C. Tsao, W. Hwu, Z. Liang, and B. P. Sutton. Accelerating advanced mri reconstructions on gpus. In *Proceedings of the 5th conference on Computing frontiers, CF 08, pages 261272, New York, NY, USA*, 2008.

[27] I. J. Sung, J. A. Stratton, and W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT 10*, 2010.

[28] V. Volkov. Better performance at lower occupancy. http://www.cs.berkeley.edu/ volkov/volkov10-GTC.pdf, 2010.

[29] V. Volkov and J. Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. In *Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley*, 2008.

[30] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[31] C. Wei-Fan, D. Michael, H. Todd, P. Tyler, T. Kevin, H. Mary, W. Phil, and G. James. Gpu acceleration of the generalized interpolation material point method. 2009.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35(3):178–194, 2009.

[33] J. K. H. Z. Yi Yang, Ping Xiang. Gpugpu compiler for memory optimization and parallel management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010.

[34] L. Yixun, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009.

VITA

Swapneela Padmakar Unkule, the daughter of Padmakar Unkule and Manisha Unkule was born on August 5, 1986 in Pune, Maharashtra, India. She received the degree bachelor of engineering (B.E) in Electronics and Tele-Communication from Cummins College of Engineering Pune, Maharashtra, India in 2008. Following her bachelors degree she joined Infosys Technologies Ltd, one of the top IT company in India as a System Engineer. After working for year and a half she entered in Masters of Science (M.S) degree program at Texas State University-San Marcos in January 2010. During her masters she was employed as grader, research assistant, and graduate instructional assistant with Texas State University-San Marcos. During her degree she was awarded with Southwest Research Institute scholarship for 2011. For her good academic record she, received the membership of Alpha Chi National College Honor Society.

Permanent Email Address: swapneela_unkule@yahoo.com

This thesis was typed by Swapneela P. Unkule.