Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality^{*}

Swapneela Unkule, Christopher Shaltz, and Apan Qasem

Texas State University, San Marcos, TX 78666, USA

Abstract. Hundreds of cores per chip and support for fine-grain multithreading have made GPUs a central player in today's HPC world. For many applications, however, achieving a high fraction of peak on current GPUs, still requires significant programmer effort. A key consideration for optimizing GPU code is determining a suitable amount of work to be performed by each thread. Thread granularity not only has a direct impact on occupancy but can also influence data locality at the register and shared-memory levels. This paper describes a software framework to analyze dependencies in parallel GPU threads and perform source-level restructuring to obtain GPU kernels with varying thread granularity. The framework supports specification of coarsening factors through source-code annotation and also implements a heuristic based on estimated register pressure that automatically recommends coarsening factors for improved memory performance. We present preliminary experimental results on a select set of CUDA kernels. The results show that the proposed strategy is generally able to select profitable coarsening factors. More importantly, the results demonstrate a clear need for automatic control of thread granularity at the software level for achieving higher performance.

1 Introduction

Increased programmability and inclusion of higher precision arithmetic hardware has opened the doors for general-purpose computing on the GPU. With hundreds of cores and extremely fine-grain multithreading, GPUs offer massive amounts of on-chip parallelism. However, much of the raw computational power of GPUs today, remain unrealized. Consider the recent release of the top500 list, where three of the top five systems are configured with GPUs. Yet, none of these three systems achieve more than 55% of peak floating-point computation, falling significantly short of the 67% average for the entire list [4]. Furthermore, with the exception of very regular streaming algorithms, successful porting of HPC applications to GPUs has required significant amount of manual tuning [22,28,17]. Thus, harnessing the computation power of these emerging platforms merits the need for software tools that can reason about program behavior, automatically restructure code to yield better performance and reduce programmer effort.

^{*} This research was supported by IBM through a Faculty Award and by Nvidia Corporation through an equipment grant.

A key consideration in producing high-performing GPU code is effectively managing both the thread and memory hierarchies. In the CUDA programming model, threads are grouped together in *blocks* and thread blocks are grouped into *grids*. Thus, the programmer needs to deal with multi-level parallelism by ensuring that not only the individual threads but the thread blocks can execute in parallel. On the other hand, effectively utilizing the memory hierarchy on the GPU is challenging because it is structured differently than its CPU counterpart and is divided into various subspaces including global, local, constant, cache, shared and texture memory. The element that further complicates the issue is the sharing of memory resources at various levels in the thread hierarchy. For example, unlike CPUs, the total number of registers is divided among threads in the same thread block. Similarly, although each thread block has access to its own shared memory, global memory is shared by all thread blocks in a grid. Thus, the placement and reuse of data (both intra and inter thread reuse) at various levels of the memory hierarchy and the decomposition of tasks are inter-related and needs to be carefully orchestrated by the programmer to obtain the desired performance.

Fig. 1. Performance sensitivity to register pressure of selected CUDA SDK kernels

This work focuses on the register and shared-memory subspaces of the memory hierarchy and proposes a software-based strategy that enables automatic adjustment of thread granularity for exploiting inter-thread data locality at the shared memory level. 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 [24]. 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 [2]. 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. Fig. 1 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 other kernels are significantly affected by changes in register pressure. Closer inspection revealed that most of this performance loss was to due additional accesses to local or shared memory which could have been curbed through better register reuse. One of the main factors that determines register pressure is thread granularity. Generally, a coarser granularity implies more work is done per thread with a higher demand for registers, while a finer granularity indicates lower demands. However, there are two factors that complicate this relationship between register pressure and thread granularity. First, if sibling threads in a thread block exhibit data locality either at the shared or global memory levels, then this locality can potentially be exploited at the register level by fusing multiple threads and increasing granularity. Although this will increase register pressure, the saved memory references can lead to improved performance. The second complicating issue is that thread granularity is also intimately tied with occupancy. Since GPU codes generally tend to perform better with higher occupancy, when increasing granularity for improved register reuse, care must be taken to not reduce occupancy to the point where it hurts performance. Therefore, automatic thread coarsening requires careful consideration of these two factors, and striking the right balance between occupancy and register reuse. Our proposed approach aims to achieve this goal.

Our strategy supports both automatic and semi-automatic methods of thread coarsening. In the semi-automatic mode, the programmer specifies the coarsening factor as a source-level pragma, whereas in the automatic method thread granularity is determined using a compiler heuristic. In both cases, we verify the legality of the coarsening transformation. To this end, we develop a generalized dependence analysis framework that allows fast analysis of CUDA programs. The analysis is based on the observation that the CUDA programming model enforces certain restrictions on thread execution, which are not enforced on multithreaded code for CPUs. For example, only barrier synchronization is supported among threads within a given thread block while no synchronization is allowed *between* thread blocks. These constraints allow for the construction of a simpler dependence framework and enables us to verify the legality of certain code transformations that would not be feasible on CPUs. Although, applied only to thread coarsening in this paper, the dependence framework has wider applicability and can be used to determine legality and profitability of a range of transformations including loop fusion and fission, unroll-and-jam and loop interchange. To facilitate fully automatic thread coarsening, we analyze the threads to detect the presence of inter-thread data locality at the shared memory level and then apply a heuristic that uses estimates of thread occupancy and register pressure as constraints to select a profitable thread granularity. Thus, the main contributions of this paper are

- a framework to provide compile-time support for both automatic and semi-automatic thread coarsening of CUDA kernels
- a generalized dependence analyzer for fast implementation of compiler optimizations of GPU codes

- an analytical model to estimate register pressure in kernels, on a per-thread basis

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 provides an example, illustrating the coarsening transformation. Section 4 describes our dependence analysis framework, the thread coarsening algorithm and the analytical model for estimating register pressure. Section 5 presents an overview of our optimization framework. Preliminary experimental results appear in Section 6 and finally we conclude in Section 7.

2 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. [23] present a general framework for optimizing applications on GPUs. Their proposed strategies include 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 [17]. The key transformation used in their work was the combining of transpose operations with FFT computation. Demmel and Volkov [24] manually optimize the matrix multiplication kernel and produce a variant that is 60% faster than the auto-tuned 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 [15]. The framework statically estimates execution cycle count of a given CUDA loop, and uses the information to select optimal unroll factors. Liu et al. [27] 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 [12]. 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 [25]. Nukada and Matsuoko also provide a highly-optimized 3D-FFT kernel [21]. 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 [16]

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 [20]. Grauer-Gray and Cavazos present an autotuning strategy for utilizing the register and shared memory space for belief propagation algorithms [18].

Automatic approaches to code transformation has been mainly focused on automatically translating C code to 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. [19] 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.

The work presented in this paper 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. We do not claim that this approach is superior to the approaches proposed previously but rather that framework can be used in conjunction with many of the strategies mentioned in this section.

3 An Example

In this section, 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 Fig. 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 Fig. 2(a).

In Fig. 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

Fig. 2. Inter thread locality in stencil computation

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 reuse¹. Thus, the coarsening transformation shown in Fig. 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 Fig. 2, if we assume 8 registers per thread and 512 threads per block, then for a GPU with 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. 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 not only can 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.

4 Automatic Thread Coarsening

In this section, we describe the main code transformation in our proposed approach and discuss the analyses needed to apply this transformation safely and profitably.

¹ Our experience showed that `nvcc` does not always perform scalar replacement and hence we included this transformation in our framework

4.1 Notation and Terminology

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

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

4.2 Dependence Analysis

The goal of our dependence analysis framework is to determine if there is a dependence between two statement *instances*. To achieve this, we augment a *data dependence graph* (DDG) to represent thread and statement instances and maintain data access information across execution of multiple threads. We assume that the input to our framework is a CUDA program that is legally parallelized for execution on a GPU. This assumption simplifies some of the analysis since it implies, that in the absence of **syncthreads()** primitives (i.e., barrier synchronization), statement instances that belong to different threads (or threads blocks) are dependence-free. Thus, given this framework we can make the following claim:

if $\nexists$ syncthreads() primitives in the kernel body then
 $\nexists S_{(i,p)} \succ S_{(j,q)}, \forall i, j \in 1, \dots, N \text{ and } \forall p, q \in 1, \dots, T$

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 parallel configuration. For this reason, we extend our dependence framework to handle input dependences. 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 $i < k < j$

Conventional dependence analysis [7] 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 *thread ID*. Hence, although a subscript analysis of the source may show two statements as accessing the same memory location, in reality they would access different memory 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; 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 dependences have been identified, our dependence analyzer makes another pass through the DDG to identify dependences that arise from the presence of `synctreads()`. This final pass mainly involves checking for the existence of condition (ii), mentioned above.

4.3 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 adopted, 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 dependences. 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 then 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:

$$T \bmod C_x = 0 \quad (1)$$

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 .

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

or

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

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

Constraint (2) describes the situation where we have no dependence between statement instances within the coarsening range. Note, we are only concerned

about dependences that emanate from a *higher* numbered thread. For the coarsening transformation, dependences that emanate from a *lower* numbered thread is irrelevant, since, by default, all statement instances in p get executed after the last statement in q , where $p > q$. Thus, all such dependences 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. In our framework, we use a variant of the partial redundancy elimination (PRE) algorithm [9] that performs this task, once threads have been coarsened.

4.4 Profitability Analysis

There are two main profitability considerations for the thread coarsening transformation. First, we need to ensure that there is enough inter-thread data locality in the kernel to make coarsening worthwhile. Second, we need to determine if coarsening will cause an excessive increase in register pressure.

Detecting Inter-thread Locality. A CUDA kernel exhibits inter-thread data locality if two threads in the same thread block access 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 two memory locations that map to the same cache line. However, we do not explicitly consider the case of spatial reuse in this paper.

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 occur 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 for that reference, all threads in the thread block will access the same memory location. 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 sequentially. The advantage of using thread independent dependences 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

Algorithm 1. Estimating Register Pressure

```

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 any of the kernel input variable then
      set the scope as global
    end if
    if variable name is preceded by __shared__ keyword then
      set scope as shared 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 index of RHS variable with LHS variable
    end if
  else if statement type is comparison then
    for all variables present in the statement increment read count
  end if
until end of AST

```

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 $i_0, \dots, i_n$ in K such that no $i \in \{i_0, \dots, i_n\}$ 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*

Estimating Register Pressure. Both the PTX analyzer [1] 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 [11]. An outline of our algorithm is given in Algorithm 1. Our strategy operates on the source code AST and the DDG built by our framework. The basic idea is to identify references that exhibit register level reuse. If there are multiple references with register level reuse then we predict that the compiler will coalesce them into a single register. One limitation of this approach is that it does not consider the possibility of any other transformation being applied to the kernel other than thread coarsening. We plan to extend this model in future to address this issue.

4.5 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

<pre> __global__ void kernell(int *in) { __shared__ float as[]; sum = as[threadIdx.x] + 1; } </pre>	<pre> __global__ void kernell(int *in){ __shared__ float as[]; int __i = threadIdx.x; for(int __k=0; __k<CF; __k++, __i+= (BS/CF)) { sum = as[__i] + 1; } } </pre>
(a) before	(b) after

Fig. 3. Simple Thread Coarsening

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. Fig. 3 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 in each iteration.

As mentioned in Section 4.3, 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 4.3. This implies there are no dependences between statement in different threads. Therefore, in this case, we only need to insert the loop and adjust the memory references (as shown in Fig. 3).
- (ii) `__syncthreads()` is present but is not control-dependent on any loop: Fig. 4 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 `__syncthreads()` to the part following it and the value in the dependence

<pre> __global__ void kernel2(int *in) { as[threadIdx.x] = in[index]; __syncthreads (); sum = as[threadIdx.x] + as[threadIdx.x+2]; } </pre>	<pre> __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] } } </pre>
(a) before	(b) after

Fig. 4. Thread Coarsening in the presence of `__syncthreads()`

is a function of thread ID, needs special handling. In this situation, while distributing the loop we need to change the scalar variable to an array with size equal to CF . 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:* If a synchronization primitive appears inside a loop in the kernel then loop distribution results in an illegal transformation. In such a case, we perform (implicitly) an unroll of the loop by the coarsening factor. Fig. 5 illustrates this transformation.

<pre> __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]; } } </pre>	<pre> __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]; } } </pre>
(a) before	(b) after

Fig. 5. Thread Coarsening in the presence of `__syncthreads()` in loops

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 2.

Algorithm 2. Thread Coarsening Transformation

```

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
find all the occurrence of __syncthreads() in the kernel and store in a list
if list is not empty then
  if none of the occurrence of __syncthreads() are inside a loop 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
    return from function with error signal
  end if
else
  expand loop body till last statement of kernel and return
end if
  
```

5 CREST : CUDA Kernel Restructuring and AutoTuning

Fig. 6 gives an overview of CREST, our code restructuring and tuning framework for CUDA kernels. The framework leverages several existing tools including `nvcc` for code generation, HPCToolkit [6] and PAPI [10] for collection of performance metrics, and PSEAT [26] for online search algorithms. The rest of the components have been developed from scratch.

Fig. 6. Overview of CREST

5.1 Kernel Extraction

To facilitate analysis, a stand alone 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 analysed. The extraction is performed on a line by line basis, using Perl 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 could be used in any application where CUDA kernels are to be examined.

5.2 Code Restructurer

At the heart of the framework is a source-to-source code transformation tool that analyzes dependences in CUDA kernels and implements a range of optimizations, some of which, to our knowledge, are currently not supported by any source-to-source transformer or by `nvcc`. These transformations include unroll-and-jam, scalar replacement, multi-level loop fusion and distribution, loop interchange, and thread coarsening. More specialized transformations such as array padding for eliminating conflicts in cache (on Fermi) and shared memory banks, array contraction for orchestrating placement of data, and iteration space splicing for reducing conditionals in kernels are currently being included in the tool. In addition to providing support for high-level code transformations, the code restructurer also implements several GPU specific heuristics for profitable application of these transformations. For example, when applying tiling on the GPU, the framework selects a tile size (and shape) that aims to improve intra-thread locality and reduce the number of synchronizations.

Another useful feature of the code restructurer is its support for fine-grain control over optimizations through source code annotation. Fig. 7 shows example directives for thread coarsening and unroll-and-jam, embedded in the *3D Jacobi* kernel. A directive is simply a comment line that specifies a particular transformation and one or more optional parameter values. In the matrix transpose code, the directive specifies coarsening of threads within a block, along the x dimension by a factor of two, and unroll-and-jam of the outer loop by a factor of 4. These directives can be associated with any loop, data structure, or kernel, providing explicit control over the scope of the optimization. This level of fine-grain control over transformations is generally not available in compilers. `nvcc` exposes a few control knobs to the user (e.g., `-maxregcount`). For comprehensive tuning of CUDA kernels, however, this merely scratches the surface.

5.3 Autotuning Support

Autotuning of kernels and applications has emerged as a dominant strategy in the HPC domain. As architectures and applications grow in scale and complexity, the need for autotuning is likely to be even more pronounced. With this picture in mind, we have designed our framework with support for autotuning. Currently, `cuDAProf` is used to collect performance feedback from CUDA kernel execution. This feedback is sent to a search engine, PSEAT, that implements a variety of online (genetic algorithm, simulated annealing), and offline (support vector machines, kernel component analysis) search methods. Our future plans include developing an interface with PAPI 4.2 to collect better diagnostic feedback to make the tuning process more efficient.

```

// pragma specifies action, dimension and factor

# coarsen x 2
__global__ void jacobi(float *un, float *u, ) {

    unsigned int j = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int k = blockIdx.y * BLOCK_DIM + threadIdx.y;

# uj 4
    for (unsigned int i = 1; i <= N; i++)
        for (; j <= N; k++)
            for (; k <= N; k++)
                Un[i, j, k] = (U[i-1, j, k] + U[i+1, j, k]
                    + U[i, j-1, k] + U[i, j+1, k]
                    + U[i, j, k-1] + U[i, j, k+1] ) * c
}

```

Fig. 7. Source code directives in CREST

Table 1. Description of CUDA kernels used in experimental evaluation

Kernel	Description	Source
stencil	computes sum of neighbouring elements within a Hand coded block	
reduce	computes min, max, sum	SC CUDA Tutorial [3]
lntext	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

6 Experimental Results

Experimental Setup. All experiments are performed on a Tesla C2050 NVIDIA Fermi GPU. The card has a compute capability of 2.0 and consists of 448 cores divided among 14 multiprocessors. Number of 32-bit registers allocated to each multiprocessor is 32K, while the amount of shared memory available per block is 48K. All CUDA programs are compiled with `nvcc` version 3.2, and all C programs are compiled with GCC 4.1.2. CUDA kernels used in our experiments are described in Table 1.

Performance Potential. We first conduct 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 which is divided into 1024 blocks of size 512. Each thread running on a block, computes the sum of all 512 elements residing in the block which are stored in shared memory. Threads running on different blocks all access different elements.

We ran **synth** with varying coarsening factors from 2 to 16. Fig. 8 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

Fig. 8. Performance characteristics of `synth` for varying coarsening factors

coarsening factor of 16. This performance gain is directly attributed to reduced shared and global memory traffic. 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 increased register pressure. Our model estimates an increase of 8 registers per thread for each coarsening factor. Thus, the spike indicates a situation where the combination of register pressure and thread block size 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.

Fig. 9. Multi-dimensional coarsening with `transpose`

Fig. 10. Performance sensitivity to coarsening factors for **reduce**

Multi-dimensional Coarsening. We evaluate the effects of coarsening along multiple dimensions with **transpose**. The chart on the left in Fig. 9 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 X and Y dimensions. Clearly, for **transpose**, it is more profitable to coarsen along both X and Y dimensions, which obtains a speedup of 3.5 over the baseline version. 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 when coarsening.

Performance Sensitivity. Although thread coarsening helps in improving performance, it is not guaranteed that performance will always increase. Fig. 10 shows the speedup observed for **reduce** with different coarsening factors. 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 more than doubling the execution time. On further inspection, we found that this decrease in performance is mainly due to 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 contributed to the loss is the lower occupancy. For **reduce**, our model was able to rightly predict that the optimal coarsening factor is 4.

Overall Performance. Fig. 11 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. However, our proposed strategy was able to weed out the bad values and balance the cost of lower occupancy against the benefits of reduced memory traffic to pick suitable coarsening factors that resulted in significant performance gains.

Fig. 11. Performance improvement from thread coarsening

7 Conclusions

This paper presented an automatic approach for controlling thread granularity in GPU kernels. We described the analysis required to apply the coarsening transformation, safely and profitably. The dependence analyzer presented in this paper 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 and also in the realm of autotuning. The experimental results suggest that the proposed approach can select suitable coarsening factors to improve register reuse, reduce shared memory traffic and increase overall performance for kernels that exhibit inter-thread data locality. These results are preliminary and more extensive experimentation is needed to evaluate the true effectiveness of the proposed method. Nevertheless, these results reiterate the need for an automatic strategy for controlling thread granularity for improved performance of GPU code.

Acknowledgement. We would like to thank the reviewers for helping us improve the quality of the final version of this paper. We also thank Dr. Martin Burtcher for allowing us compute time on his GPUs.

References

1. CUDA PTX ISA, <http://www.nvidia.com/content/CUDAptxisa1.4.pdf>
2. GPU Computing SDK, <http://developer.nvidia.com>
3. Kernel for min-max and reduction, <http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/>
4. Top 500 Supercomputer Sites, <http://www.top500.org>
5. CUDA Programming Guide, Version 3.0. NVIDIA (2010)
6. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22(6), 685–701 (2010)

7. Allen, R., Kennedy, K.: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann (2002)
8. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta, R. (ed.) *CC 2010*. LNCS, vol. 6011, pp. 244–263. Springer, Heidelberg (2010)
9. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994* (1994)
10. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: *ACM/IEEE 2000 Conference, Supercomputing* (November 2000)
11. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems* 16(6), 1768–1810 (1994)
12. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: *PPoPP 2010: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 115–126. ACM, New York (2010)
13. Cytron, R., Ferrante, J.: What’s in a name? -or- the value of renaming for parallelism detection and storage allocation. In: *ICPP 1987*, pp. 19–27 (1987)
14. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: *SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12. IEEE Press, Piscataway (2008)
15. Murthy, G., Ravishankar, M., Sadayappan, M.B., Optimal, P.: loop unrolling for gpgpu programs. In: *IEEE International Symposium on Parallel Distributed Processing* (2010)
16. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A memory model for scientific algorithms on graphics processors. In: *SC 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, p. 89. ACM, New York (2006)
17. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete fourier transforms on graphics processors. In: *SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12. IEEE Press, Piscataway (2008)
18. Grauer-Gray, S., Cavazos, J.: Optimizing and Auto-tuning Belief Propagation on the GPU. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) *LCPC 2010*. LNCS, vol. 6548, pp. 121–135. Springer, Heidelberg (2011)
19. Lee, S., Min, S.J., Eigenmann, R.: 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)
20. Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU Kernels for Dense Linear Algebra. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) *VECPAR 2010*. LNCS, vol. 6449, pp. 83–92. Springer, Heidelberg (2011)
21. Nukada, A., Matsuoka, S.: Auto-tuning 3-d FFT library for CUDA GPUs. In: *SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10. ACM, New York (2009)

22. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010)
23. Ryoo, S., Rodrigues, C.I., Bagsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.M.W.: 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)
24. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: *SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008)
25. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.* 35(3), 178–194 (2009)
26. Yi, Q., Qasem, A.: Exploring the Optimization Space of Dense Linear Algebra Kernels. In: Amaral, J.N. (ed.) *LCPC 2008. LNCS*, vol. 5335, pp. 343–355. Springer, Heidelberg (2008)
27. Yixun, L., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (2009)
28. Zhuo, Y., Wu, X.L., Haldar, J.P., Hwu, W.M., Liang, Z.P., Sutton, B.P.: Accelerating iterative field-compensated mr image reconstruction on GPUs. In: *Proceedings of the 2010 IEEE International Conference on Biomedical Imaging: From Nano to Macro, ISBI 2010* (2010)