A GPGPU Compiler for Memory Optimization and Parallelism Management

Yi Yang
Dept. of ECE
North Carolina State University
yyang14@ncsu.edu

Ping Xiang
School of EECS
Univ. of Central Florida
xp@knights.ucf.edu

Jingfei Kong
School of EECS, UCF
Univ. of Central Florida
jfkong@cs.ucf.edu

Huiyang Zhou
Dept. of ECE
North Carolina State University
hzhou@ncsu.edu

Abstract

This paper presents a novel optimizing compiler for general purpose computation on graphics processing units (GPGPU). It addresses two major challenges of developing high performance GPGPU programs: effective utilization of GPU memory hierarchy and judicious management of parallelism.

The input to our compiler is a naïve GPU kernel function, which is functionally correct but without any consideration for performance optimization. The compiler analyzes the code, identifies its memory access patterns, and generates both the optimized kernel and the kernel invocation parameters. Our optimization process includes vectorization and memory coalescing for memory bandwidth enhancement, tiling and unrolling for data reuse and parallelism management, and thread block remapping or address-offset insertion for partition-camping elimination. The experiments on a set of scientific and media processing algorithms show that our optimized code achieves very high performance, either superior or very close to the highly fine-tuned library, NVIDIA CUBLAS 2.2, and up to 128 times speedups over the naive versions. Another distinguishing feature of our compiler is the understandability of the optimized code, which is useful for performance analysis and algorithm refinement.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – Compilers, Optimization

General Terms Performance, Experimentation, Languages

Keywords GPGPU; Compiler;

1. Introduction

The high computational power and affordability of state-of-art graphics processing units (GPU) have made them the first widely accessible parallel computers with teraflops capability. To fully realize the power of general purpose computation on graphics processing units (GPGPU), two key issues need to be considered carefully: (1) how to parallelize an application into concurrent work items and distribute the workloads in a hierarchy of thread blocks and threads; and (2) how to efficiently utilize the GPU memory hierarchy, given its dominant impact on performance. As these two issues usually coupled together and finding an optimal tradeoff between different levels of parallelism and memory optimizations requires detailed understanding of GPU hardware, developing high performance GPGPU programs remains challenging for application developers. Furthermore, GPU hardware architectures are evolving rapidly, which makes the code developed and tuned for one generation (e.g., NVIDIA GTX 8800) less optimal for the next one (e.g., NVIDIA GTX280). Our envisioned solution to these problems is to let application developers identify fine-grain thread-level parallelism and/or data-level parallelism and to use an optimizing compiler to perform memory and parallelism optimizations. This way, we leverage the algorithm-level expertise of application developers and at the same time relieve them of low-level hardware-specific performance optimizations.

Our compiler works as follows. The input is a naïve GPU kernel function, which is functionally correct but does not include any device-specific performance optimizations. Such a kernel function represents the user-identified fine-grain work item that can run concurrently. A typical example of a fine-grain work item is the computation of a single data element in the output domain. The compiler analyzes the naïve kernel, checks the off-chip memory access patterns, and optimizes the memory accesses through vectorization and coalescing to achieve high data access bandwidth. Then the compiler analyzes data dependencies and identifies possible data sharing across threads and thread blocks. Based on data sharing patterns, the compiler intelligently merges threads and/or thread-blocks to improve memory reuse through the register file and the on-chip shared memory. These merges provide a novel way to achieve loop tiling and unrolling by aggregating fine-grain work items into threads and thread blocks. Additionally, the compiler schedules the code to enable data prefetching so as to overlap computation with memory access latencies. To avoid partition camping [12] (i.e., to distribute memory traffic evenly across memory partitions), thread blocks are checked for their memory accesses and depending on the thread block dimensions, either an address offset is inserted or the block identifiers (ids) are remapped, if necessary. The compiler also performs hardware-specific tuning based on hardware parameters such as the register file size, the shared memory size, and the number of cores in the target GPU.

Besides the aggressive compiler optimizations, another distinguishing feature of our compiler is that the optimized code is reason-ably understandable compared to the code generated using algebraic frameworks such as polyhedral models [11]. As a result, it is relatively easy to reason about the optimized code generated by our compiler, which facilitates algorithm-level exploration.

In our experiments, we used the compiler to optimize 10 scientific and image processing functions. The experimental results on NVIDIA 8800 GTX and NVIDIA GTX 280 GPUs show that our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'10 June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019-3/10/06…$10.00.

optimized code can achieve very high performance, either superior or very close to the NVIDIA CUBLAS 2.2 library and up to 128X over the naïve implementation.

In summary, our work makes the following contributions. (1) We propose a compiler for GPGPU programming that enables the application developers to focus on algorithm-level issues rather than low-level hardware-specific performance optimizations. (2) We propose a set of new compiler optimization techniques to improve memory access bandwidth, to effectively leverage on-chip memory resource (register file and shared memory) for data sharing, and to eliminate partition conflicts. (3) We show that the proposed optimizing compiler is highly effective and the programs optimized by our compiler achieve very high performance, often superior to manually optimized codes.

The remainder of the paper is organized as follows. In Section 2, we present a brief background on the NVIDIA CUDA programming model [19] and highlight key requirements for high performance GPU computation. In Section 3, we present our proposed optimizing compiler in detail. Section 4 explores the design space of our propose optimizations. A case study of matrix multiplication is presented in the Section 5 to illustrate the compilation process. The experimental methodology and results are presented in the Section 6. In Section 7, we highlight the limitations of the proposed compiler. Related work is discussed in Section 8. Finally, Section 9 concludes our paper and discusses future work.

2. Background

State-of-the-art GPUs employ many-core architectures. The on-chip processors cores are organized in a hierarchical manner. In the NVIDIA G80/GT200 architecture, a GPU has a number of streaming multiprocessors (SMs) (16 SMs in an NVIDIA GTX 8800 and 30 SMs in an NVIDIA GTX 280) and each SM contains 8 streaming processors (SPs). The on-chip memory resource includes register files (32kB per SM in GTX 8800 and 64kB per SM in GTX 280), shared memory (16kB per SM), and caches with undisclosed sizes for different memory regions. To hide the long off-chip memory access latency, a high number of threads are supported to run concurrently. These threads follow the single-program multiple-data (SPMD) program execution model. They are grouped in 32-thread warps with each warp being executed in the single-instruction multiple-data (SIMD) manner. According to the CUDA programming guide [19], each warp contains threads of consecutive, increasing thread ids. In a typical 2D/3D execution domain, the threads in a warp (if not at the boundary) have increasing thread ids along the X direction, and the same thread ids along the Y and Z directions.

In the CUDA programming model, the code to be executed by GPUs is the kernel functions. All the threads will run the same kernel code with different thread ids to determine their workloads. The software architecture also defines the concept of a thread block as an aggregation of threads which must be executed in the same SM and the threads in the same thread block can communicate with each other through the shared memory on the SM.

Next, we summarize the key aspects for high performance GPGPU code as they are the main focus of our proposed compiler optimizations.

a) Off-chip memory access bandwidth. To utilize the off-chip memory bandwidth efficiently, memory accesses need to be coalesced and each data item may need to be a vector type, depending on specific GPU hardware. Memory coalescing refers to the requirement that the accesses from 16 consecutive threads in a warp (i.e., a half warp) can be coalesced into a single contiguous, aligned memory access [19]. In this paper, we refer to such a coalesced contiguous, aligned region as a coalesced segment. If each memory access is of the type 'float', each segment starts from an address which is a multiple of 64 bytes, and has the size of 64 bytes. The memory bandwidth utilization may be significantly improved when each of coalesced memory accesses is of a vector data type, such as float2 (a vector of two float numbers) and float4 (a vector of four float numbers). For ATI/AMD HD 5870, the sustained bandwidth reaches 71GB/s, 98GB/s, and 101GB/s when accessing 128MB data using the float, float2, and float4 data types, respectively. In comparison, for the same data transmission on NVIDIA GTX 280, the sustained bandwidth is 98GB/s, 101GB/s, and 79GB/s using the float, float2, and float4 data types, respectively.
b) Shared memory. The common usage of shared memory is a software-managed cache for memory reuse. Although it has low access latencies, shared memory is slower than register files and has certain overheads beyond access latency. First it needs to be synchronized to ensure proper access order among the threads in a thread block. Second, the shared memory in NVIDIA GPUs has 16 banks, and bank conflicts can impair the performance.
c) Balanced resource usage. As multiple threads in the same thread block and multiple thread blocks compete for limited resources in an SM, including the register file, the shared memory, and the number of the thread contexts being supported in hardware, we need to carefully balance parallelism and memory optimizations.
d) Off-chip memory partitions. In current GPUs, off-chip memory is divided into multiple partitions. There are 6 and 8 partitions in GTX8800 and GTX280, respectively, and the partition width is 256 bytes. To use the partitions effectively, the memory traffic should be evenly distributed among all the partitions. Otherwise, the requests may be queued up at some partitions while others are idle. This is referred to as partition camping [12] or partition conflicts, which are similar to bank conflicts at shared memory but incur much higher performance penalties. Since concurrent memory requests are issued on a per half-warp basis from all active thread blocks, partition conflicts happen across different thread blocks.

Note that the key performance issues listed above are not unique to current GPUs. Future many-core architectures will probably use similar approaches to achieve high memory bandwidth (i.e., coalescing, multiple memory partitions) and to reduce memory access latency (i.e., on-chip software managed cache or shared memory). So, the proposed compiler optimizations are expected to be relevant beyond the scope of GPGPU.

3. An Optimizing GPGPU Compiler

Our proposed compiler framework is shown in Figure 1. The input to our compiler is a naïve GPU kernel function, which is functionally correct, but does not include any device-specific performance optimizations. For many scientific computing and media processing functions, the naïve version is simply the code to compute one element/pixel in the output matrix/image. Typically such code is straightforward to extract from the sequential CPU code. One common example is the loop body from a heavily executed loop. In Figures 2a and 2b, we show the sample naïve kernel functions for the matrix multiplication (mm) and matrix-vector multiplication (mv) algorithms, respectively. Each computes one element at the position (idx, idy).

In Figure 2, 'idx' and 'idy' are the position/coordinate of the element in the output matrix. In the CUDA programming model, 'idy' can be viewed as the absolute thread id along the Y direction,

which is equal to (blockIdx.y*blockDimy + threadIdx.y) in the CUDA code. Correspondingly, 'idx' is the absolute thread id along the X direction, which equal to (blockIdx.x*blockDimx + threadIdx.x). In comparison, the CUDA predefined 'threadIdx.x' and 'threadIdx.y' are the relative thread position/coordinate within a thread block and we refer to them as 'tidx' and 'tidy' for short. Both tidx and tidy are independent of the thread block ids.

As can be seen from the two examples, the naïve kernel functions don't have any shared memory usage and do not require thread block partition. In other words, we may simply assume every block only has one thread. All the arrays are initially in the off-chip global memory.

For applications which require synchronization among computing different output pixels, e.g., reduction operations, a global sync function is supported in the naïve kernel.

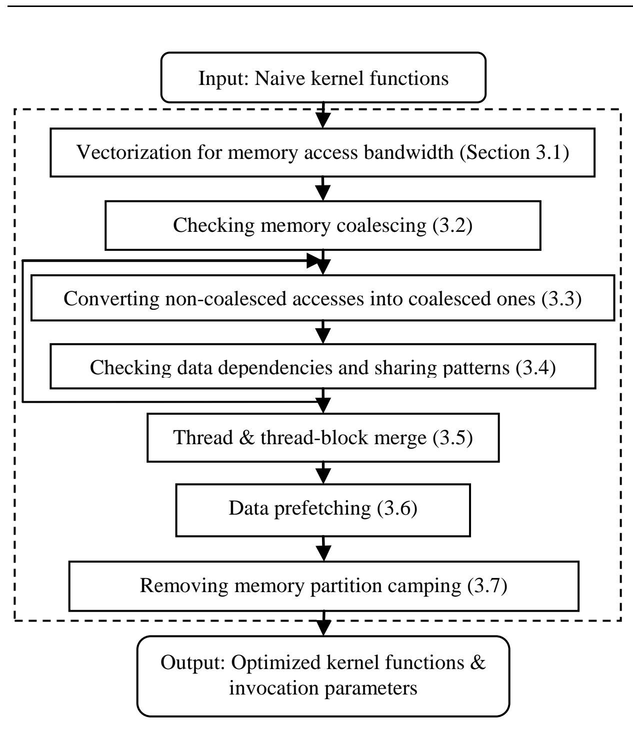

Figure 1. The framework of the proposed compiler.

To facilitate compiler optimizations, the following (optional) information can be conveyed using the '#pragma' interface: the size of the input and output dimensions, and the output variable names. The latter can be used to eliminate global memory writes to temporary variables when they are moved to shared memory.

Given the naïve kernel function, the compiler takes the following steps to generate the optimized kernel code. First, depending on the targeted GPUs, the compiler attempts to group memory accesses into vector data accesses. Second, the off-chip memory accesses are checked to see whether they satisfy the requirements for memory coalescing. If not, the code will be converted to coalesced memory accesses using shared memory as temporary storage. Third, the compiler analyzes data dependencies and sharing patterns to determine how the data are shared among the neighboring thread blocks. Based on data sharing patterns, the compiler merges both threads (i.e., combining several threads in different thread blocks into one) to enable the data reuse through registers and thread blocks (i.e., combining several blocks into one) to increase data reuse through shared memory. The data reuse information is also used to disable certain memory coalescing transformations when there is little or no data reuse. After thread/thread-block merge, the compiler schedules the code to perform data prefetching. Then, the compiler checks the memory accesses from different thread blocks for partition camping and either inserts address offsets or remaps thread block ids, if necessary. Finally, the compiler generates the optimized kernel and the parameters (i.e., the thread grid & block dimensions) to invoke the kernel function.

The optimization process described above can also be used as a generic methodology to guide manual optimizations of GPGPU programs. As a result, our optimized code is reasonably understandable, as will be seen in the remainder of Section 3.

3.1 Vectorization of Memory Accesses

As discussed in Section 2, the data type of memory accesses may have significant impact on bandwidth utilization. Therefore, the compiler first checks data accesses inside the kernel function to see whether they can be grouped in a vector type data access. Since different GPUs feature significantly different requirements on vector types for bandwidth utilization, the compiler follows different rules to adjust the aggressiveness of vectorization. In this paper, we focus on CUDA and NVIDIA GPUs, in which a vector of two floats (i.e. float2) is the preferred data type but the bandwidth improvement over the float type is less than 3%. Therefore, we use the following strict rule: if there is a pair of accesses to the same array with the indices: 2*idx+N and 2*idx+N+1, where N is an even number, the compiler generates a float2 variable *f2* with array offset as idx+N/2 and replaces the original array accesses with *f2.x* and *f2.y*. This rule is essentially designed for applications using complex numbers when the real part is stored next to the imaginary part of each data element. Note that this vectorization of data accesses is simpler than classical vectorization.

For AMD/ATI GPUs, due to the much more profound impact on bandwidth, the compiler is more aggressive and also groups data accesses from neighboring threads along the X direction into float2/float4 data types. The tradeoff of vectorization of data accesses is that if the vector data accesses are not coalesced (Section 3.2) and the compiler converts them into coalesced ones (Section 3.3) through shared memory, there may be bank conflicts. For AMD/ATI GPUs, the benefits of vectorization far overweigh the penalties of shared memory bank conflicts. For NVIDIA GPUs, however, the benefits from vectorization are limited. Therefore, the compiler skips these additional steps to vectorize data accesses.

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=a[idy][i]*b[i][idx];
c[idy][idx] = sum;
```

(a) A naïve kernel for matrix multiplication

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=a[idx][i]*b[i];
c[idx] = sum;
```

(b) A naïve kernel for matrix-vector multiplication

Figure 2. Examples of naive kernel functions.

3.2 Checking Memory Coalescing

As discussed in Section 2, GPGPU employs the SPMD model and the threads in a single warp execute the kernel function in the SIMD mode. Therefore, in order to determine whether off-chip memory accesses can be coalesced, we need to compute the addresses of each memory access in the kernel function for different threads. As arrays are the most common data structure in scientific and media processing, we consider four types of array indices and affine transformations of these indices:

1. *Constant index*: the constant value is used in an array index, for example, the constant integer '5' in '*a*[idy][i+5]'.
2. *Predefined index*: the predefined numbers, such as absolute thread ids, idx, idy, and relative thread ids, tidx (i.e., threadIdx.x), tidy (i.e., threadIdx.y), are used as an array index. For example, 'idy' in '*a*[idy][i+5]'.
3. *Loop index*: a loop iterator variable is used as an array index, for example, 'i' in '*b*[i][idx]' in Figure 2a.
4. *Unresolved index*: an array index is used, which is not one of the first three types. For example, an indirect access '*a*[x]' where 'x' is a value loaded from memory. As our compiler cannot determine the addresses of such indices, we simply skip them without checking whether they can be coalesced.

Among the four types of indices, the addresses corresponding to the first two are fixed for a given thread. For the third, however, we need to check different values of the loop iterator. Assuming that a loop index starts from *S* with increment *Incr*, then we need to check the index addresses from the first *16* iterations: *S*, *S*+*Incr*, *S*+2**Incr*, to *S*+15**Incr*. The reason is that the same behavior repeats for remaining iterations in terms of whether the access can be coalesced as the difference in addresses is a multiple of 16.

After determining the types of array indices in the kernel function, for each memory access instruction, the compiler computes the addresses from the 16 consecutive threads in the same warp (i.e., a half warp) to see whether they can be coalesced. As discussed in Section 2, if we assume the array type of 'float', the coalesced accesses will form a coalesced *segment*, which starts from an address, whose value is a multiple of 64, and has the size of 64 bytes. Among the addresses from the 16 threads, we refer to the smallest one as the 'base address'. The differences between the base address and the addresses from the subsequent 15 threads are referred to as 'offsets'. To satisfy the coalescing requirement, the base address needs to be a multiple of 64 and offsets need to be 1 to 15 words. The following two rules are used to handle common array accesses.

For an index to a multi-dimensional array, e.g., '*A*[z][y][x]', the index to the higher-order dimensions, e.g., the '*y*' and '*z*' dimensions, should remain the same for all the 16 threads in the half warp. Otherwise, for example, if the predefined index '*idx*' (the thread id along the '*x*' direction) is used in an index to the '*y*' dimension in a multi-dimension array 'A[][idx][0]', the accesses from the 16 threads will be 'A[][0][0]', 'A[][1][0]', 'A[][2][0]', etc., and are not coalesced.

When a loop index is used in the kernel function, the compiler computes the base address and the offsets for each possible value of the loop iterator. For example, for the address 'a[idy][i]' in Figure 2a, the base address is '&a[idy][0]' when the iterator 'i' is 0; '&a[idy][1]' when 'i' is 1, etc. The offsets are all zeros as the addresses do not change for different threads in the same half warp. As both the base addresses and the offsets do not meet the condition, the array access 'a[idy][i]' is not coalesced. For the array access 'b[i][idx]' in Figure 2a, the base address is '&b[0][0]' when 'i' is 0; '&b[1][0]' when 'i' is 1, etc.. The offsets are from 1 word to 15 words. Thus, the array access 'b[i][idx]' is coalesced as long as each row of array b is aligned to the multiple of 16 words. For the array access 'b[idx+i]', although the offsets satisfy the condition for every possible 'i', it is not a coalesced access since the base address is not always a multiple of 16 words, e.g., 'b[1]' when 'i' is 1.

3.3 Converting Non-Coalesced Accesses into Coalesced Ones

After the compiler analyzes every array access in the kernel code, the compiler converts the non-coalesced accesses into coalesced ones through shared memory. The observation here is that for each non-coalesced memory access instruction, the compiler can determine the coalesced segments that contain the data required by the non-coalesced memory accesses from the half warp. The compiler then introduces shared-memory array variables, inserts statements (coalesced memory accesses) to initialize the shared memory variables, and replaces the original global memory accesses with shared memory accesses. The thread block size is also set to 16 so that each thread block contains one half warp. The 'syncthreads' function is also inserted to ensure the proper access order.

```
(S0)for (i=0; i<w; i=(i+16)) {
(S1)   __shared__ float shared0[16];
(S2)   shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
(S3)   __syncthreads();
(S4)   for (int k=0; k<16; k=(k+1)) {
(S5)     sum+=shared0[(0+k)]*b[(i+k)][idx]);
(S6)   }
(S7)   __syncthreads();
(S8)}
(S9)c[idy][idx] = sum;
```

(a) The coalesced mm kernel

```
(S0)   for (i=0; i<w; i=(i+16)) {
(S1)     __shared__ float shared2[16];
(S2)     __shared__ float shared1[16][17];
(S3)     shared2[(0+tidx)]=b[i+tidx];
(S4)     for (l=0; l<16; l=(l+1))
(S5)       shared1[(0+l)][tidx]=
               a[((idx-tidx)+l)][(i+tidx)];
(S6)     __syncthreads();
(S7)     for (int k=0; k<16; k=(k+1)){
(S8)       sum+=(shared1[tidx][k]*shared2[k]);
(S9)     }
(S10)    __syncthreads();
(S11) }
(S12) c [idx] = sum;
```

(b) The coalesced mv kernel

Figure 3. Coalesced kernels generated by the compiler.

For array accesses using constant or predefined indices, the process is typically straightforward. For example, the non-coalesced access, 'A[idy][0]', the coalesced segment is 'A[idy][0:15]'. The compiler inserts a shared-memory array variable 'sA[0:15]' and initializes the 'sA[0:15]' with 'A[idy][tidx]', where tidx is relative thread id within the warp. In the case when 'idx' is used in an index to a multi-dimensional array, the compiler may introduce a loop to load the required data for a half warp. For example, for an array access 'A[idx][0]', the required data for a half warp is 'A[(idx-tidx)+(0:15)][0]', where '(idx-tidx)' provides the start address of each thread block, which is the same as the start address of the half warp as each thread block only con-

tains a half warp at this time. The coalesced segments that contains the required data are 'A[(idx-tidx)+(0:15)][0:15]'. In the introduced loop of 16 iterations, a shared memory array is initialized with 'A[(idx-tidx)+*l*][tidx]', where *l* is the iterator of the newly introduced loop. From these examples, it can be seen that not all the data loaded in the shared memory are useful, the compiler will perform data reuse analysis (Section 3.4) to determine whether this transformation is beneficial or not. If it is not, the compiler will skip coalescing transformation on this access. In the special case where an array access involves both 'idx' and 'idy', such as 'A[idx][idy]', the compiler analyzes the feasibility to exchange 'idx' and 'idy' to make it coalesced. This transformation is equivalent to loop interchange on the CPU code.

For array accesses using a loop index, 'A[*m**i+*n*]', where 'i' is the loop iterator and *m* and *n* are constants, the compiler unrolls the loop for 16/(GCD(*m*,16)) times if *m* is less than or equal to 8. If *m* is greater than 8, the coalesced access has little benefit due to limited reuse across different iterations. Then, the compiler groups the accesses from unrolled loops into coalesced ones. For example, for the array access 'A[idy][i]' where 'i' is the loop iterator, the segment 'A[idy][0:15]' contains all the required data for the first 16 iterations. The compiler unrolls the loop for 16 times, introduces shared memory variable sA[0:15] which are initialized with A[idy][tidx+i] (coalesced as the increment of 'i' is 16 after unrolling), and replaces 'A[idy][i]' with 'sA[i]'.

For the naïve kernels in Figure 2, the coalesced versions are shown in Figure 3. The inner loop with the iterator 'k' is a result of unrolling the outer loop with the iterator 'i'. In the naïve kernel in Figure 2a, the access 'a[idy][i]' is not coalesced, which results in loop unrolling as described above. 'b[i][idx]' is coalesced and it transforms to 'b[(i+k)][idx]' due to unrolling for 'a[idy][i]'. In the mv kernel in Figure 2b, both accesses 'a[idx][i]' and 'b[i]' are not coalesced. Converting the access 'b[i]' into coalesced accesses involves a loop unrolling of 16 (=16/GCD(1,16)) times and it becomes 'b[i+tidx]' in Figure 3b. For the access 'a[idx][i]' the loop with the iterator 'l' is introduced and the access is transformed to 'a[(idx-tidx)+l][i+tidx]'. In addition, the compiler may add padding to the shared memory arrays to avoid bank conflicts and padding to input data arrays to ensure that the row size of each array is a multiple of 16 words so as to meet the requirement of memory coalescing.

After memory coalescing, the kernel code generated by our compiler has the following characteristics:

1. Each thread block has 16 consecutive threads (i.e., only a half warp) along the X direction, because 16 threads are needed by hardware to coalesce memory accesses and they communicate with each other through shared memory. The number of threads in each thread block will be expanded during the next optimization phase (Section 3.5) to make sure there are enough threads in each thread block.
2. There are two types of global memory load statements: (a) Global memory to shared memory (G2S): the statements read data from global memory and store them into the shared memory, such as (S2) in Figure 3a. (b) Global memory to register (G2R): the statements read data from global memory and save them to registers. For example, in (S5) in Figure 3a, the global memory access 'b[(i+k)[idx]' loads the data into registers.

3.4 Data Dependencies and Data Sharing

In this step, the compiler detects data dependency and data sharing. Such analysis is similar to those used in analyzing affine array accesses for locality optimization and parallelization [1]. As our compiler has already enforced memory coalescing by associating coalesced segments with each global memory access, the compiler can detect data sharing by comparing whether the address ranges of the segments have overlaps. In the applications that we studied, we found that data sharing happens most frequently among neighboring blocks along the X or Y direction. Therefore, our current compiler implementation mainly focuses on checking data sharing among neighboring thread blocks and also the thread blocks with a fixed stride along the X or Y direction.

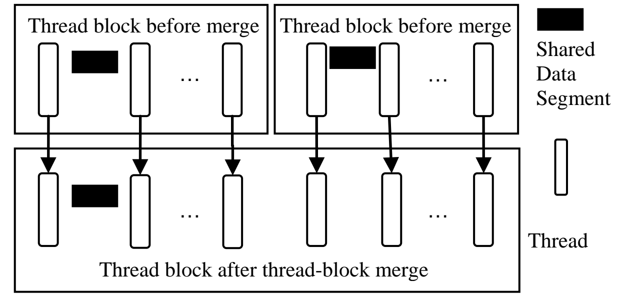

Figure 4. Improve memory reuse by merging neighboring thread blocks.

```
int i = 0;
float sum = 0;
for (i=0; i<w; i=(i+16)) {
  __shared__ float shared0[16];
  if (tidx<16) { /*inserted due to block merge to remove redundant loads */
      shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
  }
  __syncthreads();
  int k;
  for (k=0; k<16; k=(k+1)) {
      sum+=shared0[(0+k)]*b[(i+k)][idx]);
  }
  __syncthreads();
}
c[idy][idx] = sum;
```

Figure 5. The kernel function for matrix multiplication, after merging blocks along the X direction.

The data sharing/reuse information is also used to determine whether the code conversion for memory coalescing is beneficial. As described in Section 3.3, shared memory is used as temporary storage to achieve memory coalescing. The data in the shared memory, however, may not be useful as they are simply loaded from off-chip memory to satisfy the coalescing requirement. For example, the compiler loads A[idy][0:15] in order to convert the access A[idy][0] into a coalesced one. Currently, our complier employs a simple rule to check whether an access needs to be converted: if the loaded data in shared memory have no reuse, it is not converted. A more crafted heuristic may further rank code conversions for different accesses by comparing their shared memory usage and number of data reuses, and then select the most beneficial ones if shared memory is used up. We left such investigation as our future work to refine our compiler framework.

3.5 Thread/Thread-Block Merge to Enhance Memory Reuse

After detecting that there exists data sharing among thread blocks (mainly neighboring blocks), we propose two new techniques to enhance data sharing so as to reduce the number of global memory accesses: merging thread blocks and merging threads. Thread-block merge determines the workload for each thread block while thread merge decides the workload for each thread. These two techniques combined are essentially a way to achieve loop tiling and unrolling by aggregating the fine-grain work items into threads and thread blocks. We first present the two techniques and then discuss how compiler prioritizes one over the other.

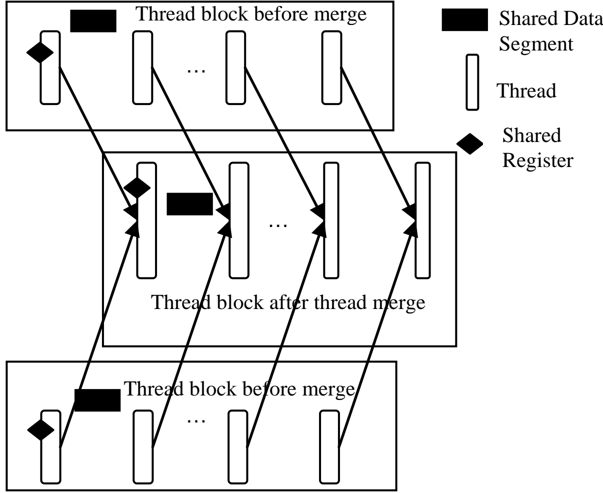

Figure 6. Improve memory reuse by merging threads from neighboring thread blocks.

3.5.1 Thread-block merge

When our compiler determines that multiple thread blocks share some common data, it may choose to merge them into one thread block, as shown in Figure 4.

To illustrate the procedure to merge thread blocks, we show how our compiler combines two neighboring blocks along the X direction into one. First, the compiler re-computes the thread id information within the thread block (i.e., tid). As two thread blocks along the X direction are merged, idx, idy and tidy remain the same while tidx is re-computed as (idx%(N*blockDim.x)), where N is 2 for Figure 4. Second, for the statements that result in data sharing, we add control flow to ensure that the global memory data are loaded only once. For the matrix multiplication example in Figure 3a, the statement S2 in threads from two neighboring thread blocks accesses the same segment. Therefore, we add an 'if (tidx < blockDim.x)' statement to eliminate redundant global memory accesses, as shown in Figure 5. Third, the thread block dimension is resized (blockDim.x = 2*blockDim.x).

As thread-block merge determines the workload for each thread block and all threads in the same thread block reuse data in shared memory, it essentially achieves loop tiling for locality and parallelism optimizations.

3.5.2 Thread merge

The other approach to enhance data sharing is to merge threads from different thread blocks, which combines several threads' workloads into one, as shown in Figure 6. Compared to thread-block merge, after these threads are combined into one, they can share not only shared memory, but also the registers in the register file. Furthermore, some control flow statements and address computation can be reused, thereby further reducing the overall instruction count. The limitation is that an increased workload typically requires a higher number of registers, which may reduce the number of active threads that can fit in the hardware. From the discussion, it can be seen that thread merge achieves the effects of loop unrolling. Note that thread merge also combines multiple thread blocks into one but it does not increase the number of threads in each thread block.

```
int i = 0;
float sum_0 = 0;
……
float sum_31 = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0_0[16];
    ……
    __shared__ float shared0_31[16];
    if (tidx<16)  {
      /* 32 is the number of the threads to
be merged */
        shared0_0[(0+tidx)]=
                a[idy*32+0][((i+tidx)+0)];
        ……
        shared0_31[(0+tidx)]=
                a[idy*32+31][((i+tidx)+0)];
    }
    syncthreads();
    int k;
    for (k=0; k<16; k=(k+1)) {
        float r0 = b[(i+k)][idx])
        sum_0+=shared0[(0+k)]*r0;
        ……
        sum_31+=shared0_31[0+k]*r0;
    }
    __syncthreads();
}
c[idy*32+0][idx] = sum_0;
……
c[idy*32+31][idx] = sum_31;
```

Figure 7. The matrix multiplication kernel after merging 32 threads in 32 adjacent blocks along the Y direction.

To illustrate the procedure to merge threads, we show how our compiler combines *N* neighboring blocks along the Y direction into one. First, the compiler re-computes the thread id information. As we merge threads from two thread blocks along the Y direction, the absolute thread ID along the X direction 'idx' remains the same while the thread ID along the Y direction 'idy' will be changed to idy**N*, idy**N*+1, idy**N*+2…, idy**N*+(*N*-1) for the *N* replicated statements. The thread id information within a thread block remains the same. Second, for the statement that results in data sharing, we need only one copy. Third, for the control flow statement such as loops, we also only need one copy. Fourth, for the remaining statements including data declaration, ALU computation statement and other memory access statements, we replicate them for *N* times. For the matrix multiplication example in Figure 5, the array access 'b[(i+k)][idx]' results in the shared data among the thread blocks along the Y direction (as the access address is not dependent on 'idy'). The compiler merges 32 neighboring blocks along the Y direction using thread merge, as shown in Figure 7.

3.5.3 Selection between thread merge and thread-block merge

As discussed in Section 3.3, the code generated by the compiler after memory coalescing has two types of global memory accesses: global to shared memory (G2S) and global to register (G2R). If data sharing among neighboring blocks is due to a G2S access, the compiler prefers thread-block merge to better utilize the shared memory. When data sharing is from a G2R access, the compiler prefers to merge threads from neighboring blocks due to the reuse of registers. If there are many G2R accesses, which lead to data sharing among different thread blocks, the register file is not large enough to hold all of the reused data. In this case, thread block merge is used and shared memory variables are introduced to hold the shared data. In addition, if a block does not have enough threads, thread-block merge instead of thread merge is also used to increase the number of threads in a block even if there is no data sharing.

```
for (i=0; i<w; i=(i+16)){
   __shared__ float shared0[16];
   shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
   __syncthreads();
   int k;
   for (k=0; k<16; k=(k+1))   {
        sum+=(shared0[(0+k)]*b[(i+k)][idx]);
   }
   __syncthreads();
}
```

(a) Before inserting perfetching

```
/* temp variable */
float tmp = a[idy][((0+tidx)+0)];
for (i=0; i<w; i=(i+16)) {
   __shared__ float shared0[16];
   shared0[(0+tidx)]=tmp;
   __syncthreads();
   if (i+16<w) //bound check
        tmp = a[idy][(((i+16)+tidx)+0)];
   int k;
   for (k=0; k<16; k=(k+1))   {
        sum+=(shared0[(0+k)]*b[(i+k)][idx]);
   }
     __syncthreads();
}
```

(b) After inserting perfetching

Figure 8. A code example to illustrate data prefetching.

3.6 Data Prefetching

Data prefetching is a well-known technique to overlap memory access latency with computation. To do so, the compiler analyzes the memory accesses in a loop and uses a temporary variable to prefetch data for the next iteration before the computation in the current loop. The process is illustrated in Figure 8. The code before insertion of prefetching is in Figure 8a and Figure 8b shows the code after insertion. Besides the temporary variable, additional checking is added to ensure that the prefetching access does not generate unnecessary memory accesses.

The overhead of data prefetching code is the increased register usage due to the temporary variables. If the register can be used for data reuse (e.g., as a result of thread merge), the compiler skips this optimization.

3.7 Eliminating Partition Camping

In this step, the compiler reuses the address access patterns obtained for thread/thread-block merge to see whether they lead to partition camping. As neighboring thread blocks along the X direction are likely to be active at the same time, the compiler focuses on the addresses that involve blockIdx.x or bidx in short. Those accesses without involving bidx either access the same line in the same partition (e.g., A[0]) or access the same partition at different times (e.g., A[bidy][0] based on the assumption that thread blocks with different bidy will execute at different times). The following rules are followed by our compiler.

Partition Camping Detection: If an array access involves bidx, the compiler checks the address stride between the two accesses from the two neighboring blocks (i.e., one with block id bidx and the other with bidx+1). The compiler detects partition camping if the stride is a multiple of (*partition size * number of partitions*). For example, for an array access A[idx], it is equivalent to A[bidx*blockDimx+tidx]. The stride between two neighboring blocks is blockDimx, whose value then decides whether there are partition conflicts (i.e., two concurrent accesses to the same partition).

Partition Camping Elimination: If an access results in partition conflicts, depending on how thread blocks are organized, we use two ways to eliminate partition conflicts:

1. If thread blocks are arranged in one dimension, we add a fixed offset, (*the partition width * bidx*), to the access and update the loop bounds to accommodate the change. For example, in mv, the output is a vector. So the thread blocks are organized in one dimension. The accesses A[idx][i] (or the coalesced version A[((idx-tidx)+l)][(i+tidx)]), from neighboring thread blocks result in partition camping if the width of A is a multiple of (*partition size * number of partitions*), as shown in Figure 9a. With the added offset, the access pattern is changed to Figure 9b, eliminating partition camping.
2. If thread blocks are organized in two or more dimensions, we apply the diagonal block reordering proposed in [12], which essentially changes the workload (or tile) that each thread block is assigned to. The diagonal mapping rule is newbidy = bidx and newbidx = (bidx+bidy)%gridDim.x.

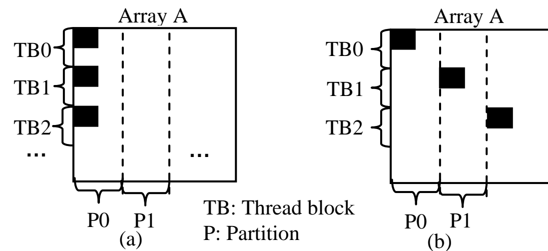

Figure 9. Eliminating partition camping. (a) Accesses to array A resulting in conflicts at partition 0. (b) Adding an offset as (partition size * bidx) eliminates the conflicts. The dark regions represent the memory footprint of A[idx][0] from different thread blocks.

4. Design Space Exploration

4.1 The Number of Threads in A Thread Block

In our compiler algorithm, the number of threads in a thread block is determined by thread/thread-block merge. The CUDA programming guide suggests that one SM should have at least 192 active threads to hide the latency of register read-after-write dependencies. Because our compiler tries to use a number of resources (the shared memory due to thread-block merge and the register file due to thread merge) for better memory reuse, it is possible that the code after thread/thread-block merge requires a large amount of shared memory and registers so that one SM can only support a limited number of thread blocks. To balance the thread-level parallelism and memory reuse, our compiler tries to put 128, 256, or 512 threads into one thread block (equivalent to merging of 8, 16, and 32 blocks), if possible. Also, the compiler varies the degrees of thread merge (i.e., how many threads to be merged into one) across 4, 8, 16, or 32 so as to balance register-based data reuse and thread-level parallelism. As such, the combination of these design parameters creates a design space to explore. As discussed in Section 3.5, merging threads/thread blocks is one way to achieve loop tiling and unrolling. So, exploring such a design space is similar to finding the best tile size and unrolling factors for parallelization and locality enhancement. Due to the non-linear performance effect of those parameters on GPU performance, the compiler generates multiple versions of code and resorts to an empirical search by test running each version to select the one with the best performance. Another way is to use an analytical performance model [7],[15] to predict the performance of each version, but this requires higher accuracy than current models. Moreover, based on our experiments, the optimal version may be dependent upon the size of the input arrays, which implies that unless the compiler knows detailed information of the intended inputs, it is almost inevitable that the compiler must run multiple versions of code in order to find the optimal one.

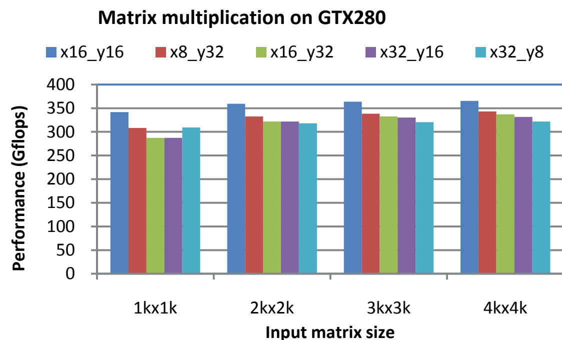

Figure 10. The performance impact (GTX 280) of the number of merged threads/thread blocks. 'x*N*_y*M*' means that *N* thread blocks merged along the X direction and *M* threads merged along the Y direction.

4.2 Hardware Specification

GPU hardware is evolving rapidly. Although different generations of GPU hardware may share similar architecture, e.g., NVIDIA GTX8800 and GTX280, there are significant changes, e.g., the register file size, which may have a strong impact on performance. When there are more registers, more threads can be put into one block or one thread can use more registers for temporary data. As a result, an optimized code tuned for one GPU generation may not be optimal for the next. To solve this problem, our compiler generates different versions of optimized code based on different machine descriptions so that they can be deployed on different GPU platforms.

5. Case Study: Matrix Multiplication

Matrix multiplication (mm) is a commonly used algorithm and there has been continuing effort to improve its performance [18]. The fine tuned implementation in NVIDIA CUBLAS 1.0 has a throughput of 110 GFLOPS when computing the product of two 2kx2k matrices on GTX 8800. In CUBLAS 2.2, a more optimized version is implemented based on the work of Vasily et. al. [18], which can reach 187 GFLOPS. In comparison, the CUDA SDK version has a throughput of 81 GFLOPS. In this section, we use matrix multiplication as an example to illustrate our compilation process.

The naïve kernel, i.e., the input to our compiler, is shown in Figure 2a. In the kernel, there are two input arrays, *a* and *b*, from the global memory. The compiler converts the accesses to array *a* into coalesced ones, as shown in Figure 3a. Based on detected data sharing, the compiler determines that neighboring thread blocks along the X direction can be merged to improve reuse of array *a* and neighboring thread blocks along the Y direction can be merged to improve memory reuse of array *b*. As the access to array *a* is a R2S (read-to-shared memory), the compiler chooses to perform thread-block merge. As the access to array *b* is R2R (read-to-register), the compiler chooses thread merge as discussed in Section 3.5. The next question is then how many thread blocks should be merged along either direction? As discussed in Section 4, the heuristic is to put at least 128 threads in each thread block and to generate different versions of kernel functions depending on the number of threads/thread blocks to be merged. Figure 10 shows the performance effect on GTX 280 of the number of merged threads/thread blocks in either direction. It can be seen that the optimal performance for different sizes of input matrices is achieved with merging 16 thread blocks along the X direction and 16 threads along the Y direction.

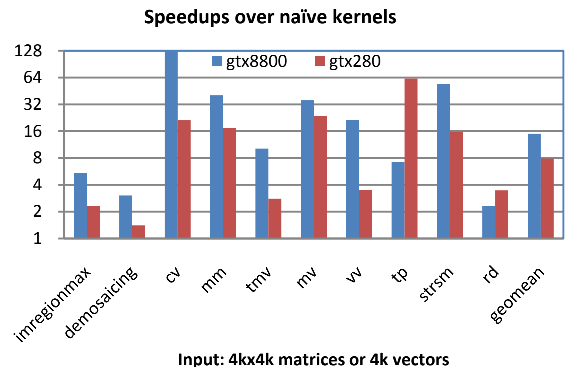

Figure 11. The speedups of the optimized kernels over the naive ones. (the input to reduction is a vector of 16M floats).

6. Experiments

6.1 Experimental Methodology

We implemented the proposed compiler framework in Cetus, a source-to-source compiler infrastructure for C programs [8]. The

CUDA language support in Cetus is ported from MCUDA [16]. The compiler optimizes the naïve kernel functions of the algorithms listed in Table 1, all of which compute a single element at the position (idx, idy). The numbers of lines of code (LOC) of these naïve kernel functions are included in Table 1 to illustrate their programming complexity/simplicity. Among the kernels, '#pragma' is used in the reduction kernel to convey the information of input vector length and the actual output to the compiler. The output of our compiler, i.e., the optimized kernel, is compiled by the CUDA compiler, nvcc, to generate the GPU executable file. In our experiments, we used both NVIDIA GTX8800 and NVIDIA GTX280 GPUs with CUDA SDK 2.2 and a 32-bit CentOS 5.2 operating system. Our compiler code, the naïve kernels, and the optimized kernels are available at [20].

Algorithm	The size of input matrices/vectors	Num. of LOC in the naïve kernel
transpose matrix vector multiplication (tmv)	1kx1k to 4kx4k (1k to 4k vec.)	11
matrix mul. (mm)	1kx1k to 4kx4k	10
matrix-vector mul. (mv)	1kx1k to 4kx4k	11
vector-vector mul. (vv)	1k to 4k	3
reduction (rd)	1-16 million	9
matrix equation solver (strsm)	1kx1k to4kx4k	18
convolution (conv)	4kx4k image, 32x32 kernel	12
matrix transpose (tp)	1kx1k to 8kx8k	11
Reconstruct image (demosaicing)	1kx1k to 4kx4k	27
find the regional maxima (imregionmax)	1kx1k to 4kx4k	26

Table 1. A list of the algorithms optimized with our compiler.

6.2 Experimental Results

In our first experiment, we examine the effectiveness of our compiler optimizations. Figure 11 shows the kernel speedups of the optimized kernels over the naïve ones running on both GTX8800 and GTX 280 GPUs. From the figure, it can be seen that the compiler significantly improves the performance using the proposed optimizations (15.1 times and 7.9 times on average using the geometric mean).

To better understand the achieved performance, we dissect the effect of each step of our compilation process and the results are shown in Figure 12. The performance improvement achieved in each step is an average of all applications using the geometric mean. Since data vectorization is designed to handle complex numbers and all the inputs in this experiment are scalar numbers, this step has no effect. From Figure 12, it can be seen that thread/thread-block merge has the largest impact on performance, which is expected as tiling and unrolling achieved with this optimization is critical for locality and parallelism optimizations. Between the two GPUs, GTX280 benefits less from the optimizations due to its improved baseline performance (i.e., naïve kernels) as the hardware features more cores and higher memory bandwidth. Prefetching shows little impact in our results. The reason is that after thread/thread-block merge, the kernel consumes many registers. When allocating registers for prefetch, either the degree of thread merge must be reduced or the off-chip local memory may have to be used, resulting in degraded performance. Therefore, when registers are used up before prefetching, the prefetching step is skipped by our compiler. Elimination of partition camping shows larger impact on GTX280 than GTX8800. One reason is due to the input data sizes used in our experiments. For example, there is significant partition camping on GTX280 when transposing a 4kx4k matrix as it has 8 partitions and the partition size is 256 bytes. For the same input on GTX8800 which has 6 partitions, the accesses become more evenly distributed and eliminating partition camping has little effect. When transposing a 3kx3k matrix on GTX8800, however, eliminating partition camping results in a 21.5% performance improvement.

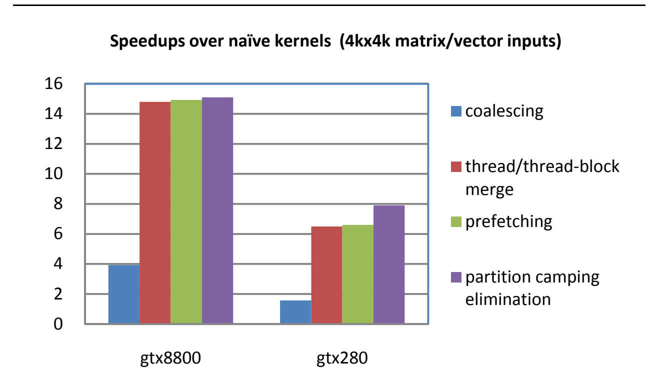

Figure 12. Performance improvement along the compilation process.

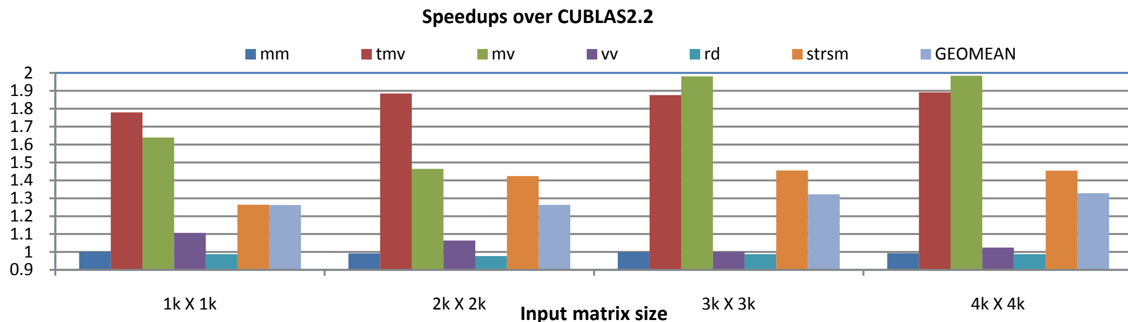

Figure 13. Performance improvement of our optimized kernels over CUBLAS 2.2 implementations on GTX280.

Among the algorithms in Table 1, six are implemented in the CUDA CUBLAS library. In the next experiment, we compare our optimized kernel with the highly tuned CUBLAS v2.2 on GTX 280. Figure 13 shows the performance comparison of the algorithms with different input sizes. From Figure 13, we can see that the kernel optimized by our compiler achieves consistently better performance than CUBLAS 2.2 for transpose matrix vector multiplication (tmv), matrix vector multiplication (mv), vector vector multiplication (vv), and matrix equation solver (strsm) for different input sizes. For matrix multiplication (mm) and reduction (rd), the performance of our optimized code is very close to CUBLAS 2.2 (within 2% difference). On average (based on the geometric mean), our performance improvement over CUBLAS varies from 26% to 33% for different input sizes.

To study the effect of data vectorization, we chose the reduction (rd) algorithm since rd is the only algorithm in our study that has a corresponding version for complex numbers (*CublasScasum*) in CUBLAS. We changed the naïve kernel of rd to process complex numbers by using two float-type variables to read the real (A[2*idx]) and imaginary (A[2*idx+1]) parts of a complex number instead of a single float2 variable. Then, we optimized this naïve kernel with and without the data vectorization step. For different input sizes, we compared the performance of the two optimized kernels (labeled 'optimized_wo_vec' and 'optimized', respectively) and the results are show in Figure 14.

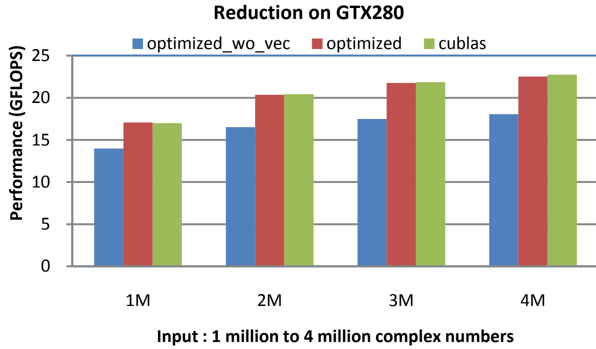

Figure 14. The effect of data vectorization on reduction with complex number inputs.

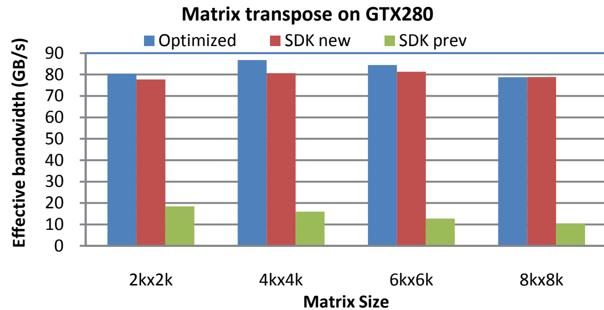

Figure 15. Performance comparison between CUDA SDK and our optimized kernel on matrix transpose.

From Figure 14, we can see that data vectorization significantly improves the performance. One reason is the improved memory bandwidth due to the use of float2 data types as discussed in Section 2. Another reason is the side effect of memory coalescing. Without data vectorization, the compiler recognized that the array accesses to both real and imaginary parts (A[2*idx] and A[2*idx+1]) are not coalesced. So, it uses shared memory as temporary storage to generate coalesced memory accesses as discussed in Section 3.3. In comparison, the accesses in the kernel after data vectorization, A[idx], is coalesced. As a result, the data are directly loaded into registers for computation. Although the compiler uses the shared memory to improve memory reuse for both vectorized and un-vectorized versions, there are more shared memory accesses in the un-vectorized kernel 'optimized_wo_vec' due to code transformation for coalescing. These extra shared memory accesses contribute to the performance differences between the 'optimized_wo_vec' and 'optimized' kernels.

Among all the kernels, transpose (tp) and matrix-vector multiplications (mv) exhibit the partition camping problem. Ruetsch and Micikevicius [12] proposed diagonal block reordering to address the issue with transpose and their implementation is included in the latest CUDA SDK. In Figure 15, we compare the performance of our optimized kernel (labeled 'optimized') with theirs (labeled 'SDK new') and we also include the previous CUDA SDK version for reference (labeled 'SDK prev'). Since tp does not have any floating point operations, the effective bandwidth is used. From Figure 15, it can be seen that although our compiler uses the same approach to eliminate partition camping, the remaining optimizations taken by our compiler result in better performance than the version in the latest SDK.

In mv, the thread blocks are in one dimension. Therefore, diagonal block reordering cannot be applied. Our compiler uses the address offset approach described in Section 3.7 and the results are shown in Figure 16. It can be seen that for different input sizes, even without partition camping elimination, our optimized kernel (labeled 'Opti_PC') already achieves better performance than CUBLAS and eliminating partition camping (labeled 'optimized') further improves the performance.

In summary, our experimental results show that our optimizing compiler generates very high quality code and often achieves superior performance even compared to the manually optimized code in CUDA CUBLAS and SDK.

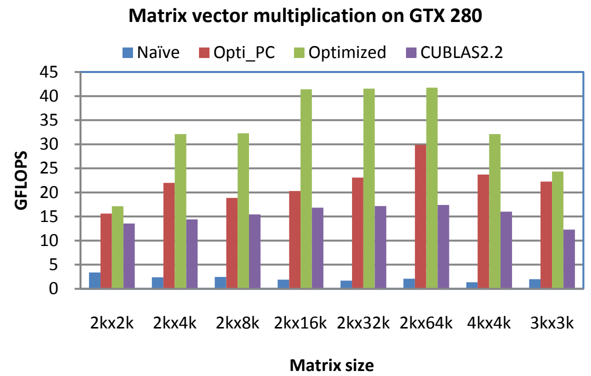

Figure 16. Performance of mv using the naïve kernel, the optimized kernel without partition camping elimination (labeled 'Opti_PC'), the optimized kernel, and CUBLAS.

7. Limitations

Although the proposed compiler can dramatically improve the performance over naïve kernel functions, the fundamental limita-

tion is that it cannot change the algorithm structure. Instead, our compiler can be used to facilitate algorithm-level exploration. The reasons are two-fold. First, developers can leverage our aggressive compiler optimizations so that they do not need to optimize their implementations of each candidate algorithm. Second, the relatively good understandability of our optimized code may give a hint of what algorithms are better suited. Taking 1D fast Fourier transform (FFT) as an example, when the naïve kernel (50 lines of code) simply uses 2-point FFT in each step of the Cooley–Tukey algorithm [4], the throughput is 24 GLOPS for computing the FFT of 2^{20} complex numbers on GTX280. Our compiler optimizes the naïve kernel by merging threads and the resulting kernel computes 8-point FFT in each step, which delivers a throughput of 41 GFLOPS, significantly better than CUFFT 2.2 (26GFLOPS). The compiler generated 8-point FFT version, however, is not as good as a naïve implementation of 8-point FFT (113 lines of code with a throughput of 44 GFLOPS). The reason is that the compiler generated version uses multiple 2-point FFT calculations for an 8-point FFT. On the other hand, as our compiler generated code is reasonably understandable, it serves as a good guideline for algorithm exploration: changing the naïve kernel from 2-point FFT to 8-point FFT, for which the compiler can further optimize the performance to achieve 59 GFLOPS. More elaborate algorithm-level development by Govindaraju et. al. [6] as well as the one used in CUFFT2.3 achieves even higher performance (89 GFLOPS), indicating that our compiler facilitates but cannot replace intelligent algorithm-level exploration.

8. Related Work

CUDA [19] provides a relatively simple programming model to application developers. However, many hardware details are exposed since it is critical to utilize the hardware resources efficiently in order to achieve high performance. Given the non-linear optimization space, optimizing GPGPU programs has been shown to be highly challenging [14]. To relieve this task from developers, there has been some recent work on compiler support for GPGPU optimization. Ryoo et. al. [13] defined performance metrics to prune the optimization spaces. G-ADAPT [10] is a compiler framework to search and predict the best configuration for different input sizes for GPGPU programs. Compared to our proposed approach, this compiler takes the optimized code and aims to adapt the code to different input sizes, while ours optimizes the naïve kernel functions.

One closely related work to ours is the optimizing compiler framework for affine loops by Baskaran et. al. [2][3]. Their compiler uses a polyhedral model to empirically search for best loop transformation parameters, including the loop tiling sizes and unrolling factors. It is reported that their compiler achieves similar performance to CUBLAS1.0 for matrix multiplication and better performance for other kernels. In comparison, our proposed compiler also uses empirical search to determine the best parameters to merge threads/thread blocks. The difference is that we propose a novel way to achieve the effect of loop tiling and loop unrolling. In our proposed approach, we start from the finest-grain work item and aggregate work items together to exploit data reuse through registers and share memory. This approach fits particularly well with GPGPU programming models where work items are typically defined in a 2D/3D grid and aggregating work items usually bears a clear physical meaning in terms of the workload of each thread and each thread block. In addition, we propose explicit rules to check memory coalescing and approaches to covert non-coalesced accesses into coalesced ones. For the applications that we studied, including matrix multiplication, our compiler achieves much better performance (superior or close to CUBLAS 2.2, which is significantly improved over CUBLAS 1.0). In addition, the loop transformed code generated based on polyhedral models is often quite complex [11] while our optimized code has relatively good understandability.

Our compiler shares a common goal with CUDA-lite [17]: the user provides a kernel function which only uses the global memory and the compiler optimizes its memory usage. In CUDA lite, the compiler uses the programmer provided annotation to improve memory coalescing. It also performs loop tiling to utilize shared memory. In comparison, our compiler does not require user annotation. More importantly, our compiler does not only improve memory coalescing but also effectively achieves data sharing with the proposed thread/thread-block merge techniques. In addition, our compiler distinguishes memory reads based on their target, the register or the shared memory, to make best use of either type of resource for data reuse.

One interesting way to automatically generate GPGPU programs is to translate OpenMP programs to CUDA programs [9]. Our proposed compiler is complementary to this work as it can be used to further optimize the CUDA kernel functions generated from OpenMP programs.

9. Conclusions

In this paper, we present a compiler framework to optimize GPGPU programs. A set of novel compiler techniques is proposed to improve GPU memory usage and distribute workload in threads and thread blocks. Our experimental results show that the optimized code achieves very high performance, often superior to manually optimized programs.

In our future work, we plan to extend our compiler to support OpenCL programs so that a single naïve kernel can be optimized for different GPUs from both NVIDIA and AMD/ATI. We are also investigating detailed analytical performance models to simply the effort in design space exploration.

Acknowledgements

We thank the anonymous reviewers and Professor Vivek Sarkar for their valuable comments to improve our paper. This work is supported by an NSF CAREER award CCF-0968667.

References

[1] A. V. Aho, Ravi Sethi, and J. D. Ullman. Compilers, Principles, Techniques, & Tools, Pearson Education, 2007.

[2] 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 *Proc. International Conference on Supercomputing*, 2008.

[3] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[4] J. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series, In *Math. Comput,* 1965.

[5] N. Fujimoto. Fast Matrix-Vector Multiplication on GeForce 8800 GTX. In *Proc. IEEE International Parallel & Distributed Processing Symposium*, 2008

[6] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proc. Supercomputing*, 2008.

[7] S. Hong and H. Kim. An analytical model for GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. International Symposium on Computer Architecture*, 2009.

[8] S.-I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proc. Workshops on Languages and Compilers for Parallel Computing,* 2003

[9] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009

[10] Y. Liu, E. Z. Zhang, amd X. Shen. A Cross-Input Adaptive Framework for GPU Programs Optimization. In *Proc. IEEE International Parallel & Distributed Processing Symposium*, 2009.

[11] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral mode: part I, on dimensional time. In *Proc. International Symposium on Code Generation and Optimization,* 2007

[12] G. Ruetsch and P. Micikevicius. Optimize matrix transpose in CUDA. NVIDIA, 2009.

[13] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Optimization space pruning for a multithreaded GPU. In *Proc. International Symposium on Code Generation and Optimization,* 2008.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[15] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modling tool for GPU architectures. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[16] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA:An efficient implementation of CUDA kernels on multicores. *IMPACT Technical Report IMPACT-08-01*, UIUC, Feb. 2008.

[17] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming Complexity, In *Proc. Workshops on Languages and Compilers for Parallel Computing,* 2008

[18] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. Supercomputing,* 2008.

[19] NVIDIA CUDA Programming Guide, Version 2.1, 2008

[20] http://code.google.com/p/gpgpucompiler/