Int J Parallel Prog (2013) 41:768–781
DOI 10.1007/s10766-012-0228-3

The Implementation of a High Performance GPGPU Compiler

Yi Yang · Huiyang Zhou

Received: 6 January 2012 / Accepted: 25 October 2012 / Published online: 9 November 2012

Abstract In this paper we present our experience in developing an optimizing compiler for general purpose computation on graphics processing units (GPGPU) based on the Cetus compiler framework. The input to our compiler is a naïve GPU kernel procedure, which is functionally correct but without any consideration for performance optimization. Our compiler applies a set of optimization techniques to the naive kernel and generates the optimized GPU kernel. Our compiler supports optimizations for GPU kernels using either global memory or texture memory. The implementation of our compiler is facilitated with a source-to-source compiler infrastructure, Cetus. The code transformation in the Cetus compiler framework is called a pass. We classify all the passes used in our work into two categories: functional passes and optimization passes. The functional passes translate input kernels into desired intermediate representation, which clearly represents memory access patterns and thread configurations. A series of optimization passes improve the performance of the kernels by adapting them to the target GPGPU architecture. Our experiments show that the optimized code achieves very high performance, either superior or very close to highly fine-tuned libraries.

Keywords GPU · Compiler · Optimization · Vectorization · OpenCL

1 Introduction

The high computational power and memory access bandwidth of state-of-art graphics processing units (GPUs) have made them appealing for high performance computing. However, it is quite challenging to develop high performance GPGPU code

Y. Yang (✉) · H. Zhou
Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA
e-mail: yyang14@ncsu.edu; hzhou@ncsu.edu

as application developers need to know how to utilize the low-level GPU hardware resources effectively. We advocate using the compiler to facilitate the development of GPGPU programs and to relieve the application developers of low-level hardware-specific performance optimizations. In this paper, we present our experiences in developing an optimizing GPGPU compiler, which takes naive GPU kernels as inputs and outputs highly optimized kernel code.

Our compiler works as follows. The input to the compiler is a naïve GPU kernel function. The main focus of such a naïve kernel function is on identifying the inherent data-level parallelism or task-level parallelism from application algorithms and ensuring the functional correctness rather than optimizing the GPGPU code performance. A typical example of such a kernel is to compute one output pixel per thread in image processing applications or an element in the output matrix/vector in linear algebra computations. Our compiler parses the input naïve kernel into intermediate representation and applies sets of functional passes, which facilitate the analysis of the off-chip memory access patterns and the thread hierarchy configurations of the naïve kernel. Then, the compiler optimizes the code through a series of optimization passes. First, the memory accesses are optimized using vectorization and coalescing to achieve high data access bandwidth. Second, data dependencies are analyzed and possible data sharing among threads and thread blocks is identified. Thread merge and/or thread-block merge are then used to leverage memory reuse through the register file and the on-chip shared memory. Third, to avoid partition camping [8], the compiler checks the thread blocks' memory accesses and the thread block dimensions. Then, the compiler either inserts an address offset or remaps the block identifiers (ids), if there exists partition camping. Additionally, data prefetching is also supported in our compiler. Because the GPUs have different hardware support for texture memory and global memory, our compiler applies different optimization passes based on the characteristics of the two memory types.

In summary, our work makes the following contributions: (1) We propose an optimizing source-to-source compiler for GPGPU kernels based on the Cetus framework. (2) We develop both functional compiler passes to analyze the GPU kernels and optimization passes to improve performances of the GPU kernels. The functional passes provide effective ways to analyze the GPU kernels and are independent of optimization passes. Therefore, our framework provides a good platform to introduce more optimization techniques in the future. (3) The programs optimized using our compiler achieve very high performance.

The remainder of the paper is organized as follows. In Sect. 2, we present a brief background on the GPGPU architecture and highlight the key requirements for high performance GPU computing. In Sect. 3, we present the details of the development of our proposed optimizing compiler using the Cetus framework. The experimental methodology and results are presented in the Sect. 4. Finally, Sect. 5 concludes our paper.

2 GPGPU Architecture

State-of-the-art GPUs use many-core architectures. The on-chip processor cores are organized in a hierarchical manner. A GPU has a number of streaming multiprocessors

(SMs) (NVIDIA GPUs) or SIMD engines (AMD GPUs), as shown in Fig. 1a. Every SM can support multiple threads running concurrently. Threads in GPUs follow the single-program multiple-data (SPMD) program execution model. We promote this simplified view of GPU architecture for application developers to write GPU programs that are functionally correct. However, it is highly challenging to optimize such programs to achieve high performance without the knowledge of the detailed hardware features. Figure 1b shows a more detailed view of an NVIDIA GPU. The AMD GPUs have very similar structures with different parameters. In each SM/SIMD engine, there are multiple streaming processors/cores. Every SM or SIMD has a number of registers, which is private to each thread, and shared memory, which is visible to a thread block/group. The off-chip memory includes both the global memory and texture memory, which have very long latency and can be used by all the threads. The memory request to off-chip memory is through multiple memory controllers to different

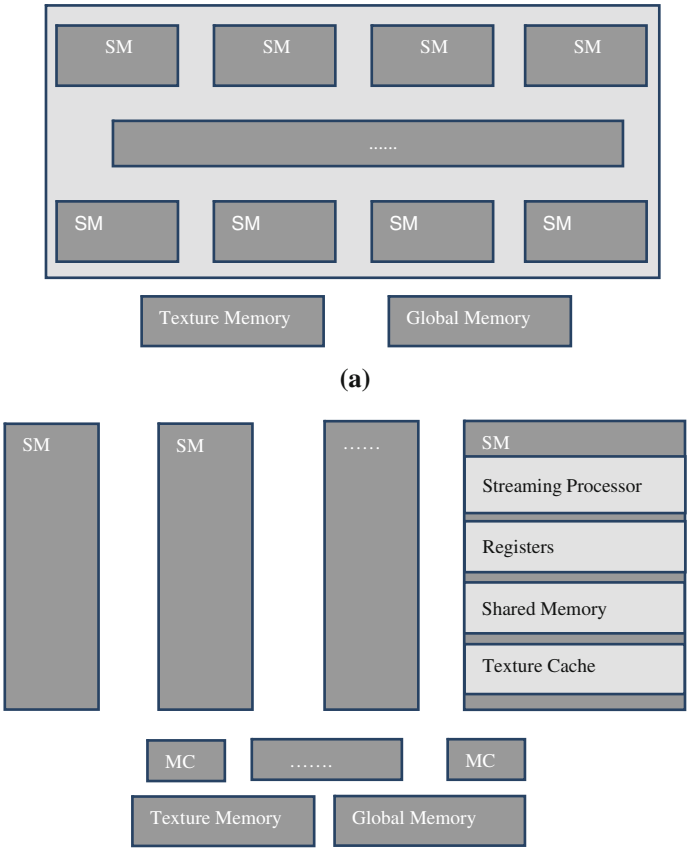

Fig. 1 The GPU architecture **a** A simplified view of GPGPU architecture, **b** A more detailed view of GPGPU architecture

memory partitions and multiple partitions can fulfill the memory requests concurrently. Between the global and texture memory, global memory is more flexible because it is both writeable and readable, while the texture memory is either write-only or read-only. On the other hand, most GPUs have hardware texture caches, which significantly improve the throughput of texture memory. In comparison, only NVIDIA GTX 480 GPUs have L1 caches for global memory. The threads in each SM/SIMD are organized in thread blocks/groups. Within a thread block/group, the threads can communicate data through fast on-chip shared memory. Each block/group has multiple warps/wavefronts, in which the threads are executed in the single-instruction multiple-data (SIMD) manner.

To achieve high performance for GPU applications, many studies [1,3,5,9,10,12] have been conducted to identify and address the performance issues on GPU applications. Compared to these prior works, our compiler focuses on memory optimizations and parallelism management. We summarize the key aspects as follows.

(a) The off-chip GPU memory includes texture memory and global memory. To achieve high memory bandwidth, four issues need to be addressed. (1) Memory coalescing, which leads to strict requirements on the memory requests from the threads in a half warp [6]. (2) Vectorization as GPU memory, especially the texture memory/caches, provides higher bandwidth for vector data types [13]. (3) Eliminating partition camping, meaning that memory requests need to be evenly distributed to different memory partitions [8]. (4) Utilizing caches to reduce memory traffic. The caches include both software manage shared memory [6], hardware texture caches, and L1 caches for global memory.
(b) Parallelism management. The workloads of a thread or a thread block have important performance impacts. If we increase the workload of a thread or a thread block, there are more opportunities to leverage data sharing or data reuse. The trade off, however is that the thread-level parallelism can be reduced.

3 The Implementation of a High Performance GPGPU Compiler

Since the focus of our compiler is memory optimizations and parallelism management, we can leverage the vendors' compiler for classic code optimizations such as dead code elimination and register allocations. Therefore, we choose to implement our compiler as a source-to-source code optimizer and the overall structure of our proposed compiler is shown in Fig. 2.

Our compiler leverages MCUDA [11], which adds CUDA language support to Cetus [2] and translates the kernel code into intermediate representation. Our compiler adds additional passes to translate the intermediate representation to our GPU intermediate representation and applies GPU optimizations on the kernel. We classify the passes into two categories: the functional passes and GPU optimization passes. The functional passes do not improve the performance of the kernels. Instead, they are needed for the preprocessor, GPU optimization passes and the postprocessor. As shown in Fig. 2, the preprocessor includes multiple functional passes which translate the CUDA intermediate representation to the GPU intermediate representation. The GPU intermediate representation includes the information on memory access patterns,

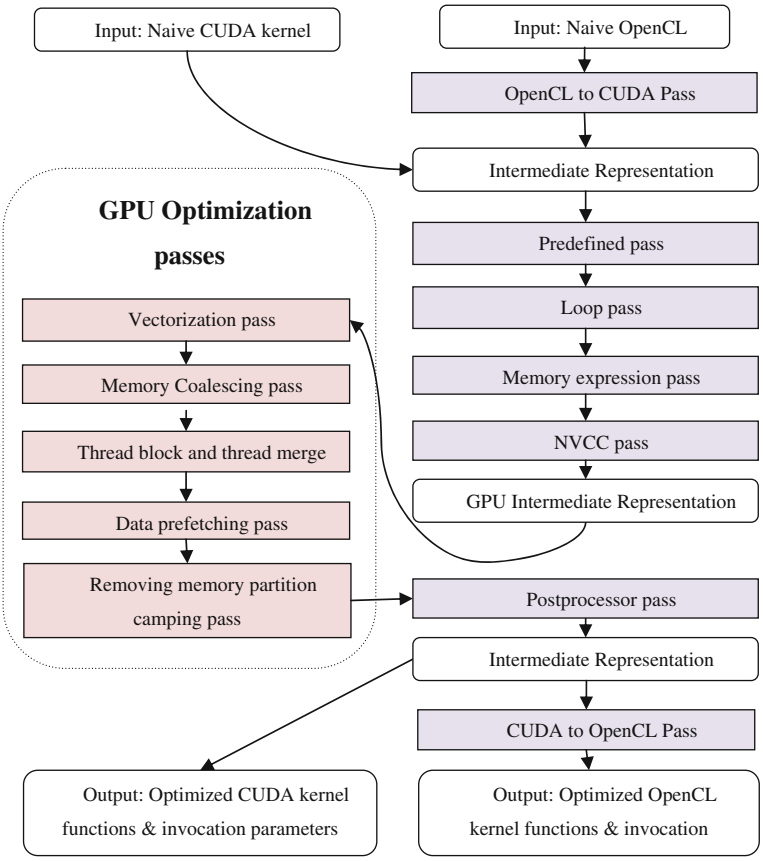

Fig. 2 The framework of the proposed compiler

loop structures, thread configurations and thread block dimensions. Then the compiler applies five GPU optimization passes on the GPU intermediate representation. Finally the postprocessor translates the GPU intermediate representation back to the CUDA intermediate representation and outputs the high performance kernels in either CUDA [6] or OpenCL [7].

To better illustrate our compiler implementation, we also include the class diagram in the Fig. 3. As shown in Fig. 3, most of the classes inherit from the Pass class. The child classes of Pass on the left hand side are GPU optimization passes and the child classes of Pass on the right hand side are functional passes. Every pass takes a GProcedure object as the input, applies code transformation on it and generates a new GProcedure object as the output for the next pass. Next we present the details of our compiler passes as well as their corresponding class implementation.

First, the functional passes are summarized as follows. The GPU kernel, matrix vector multiplication (MV), is used as a case study to illustrate the compilation steps. The naïve implementation of MV is shown in Fig. 4.

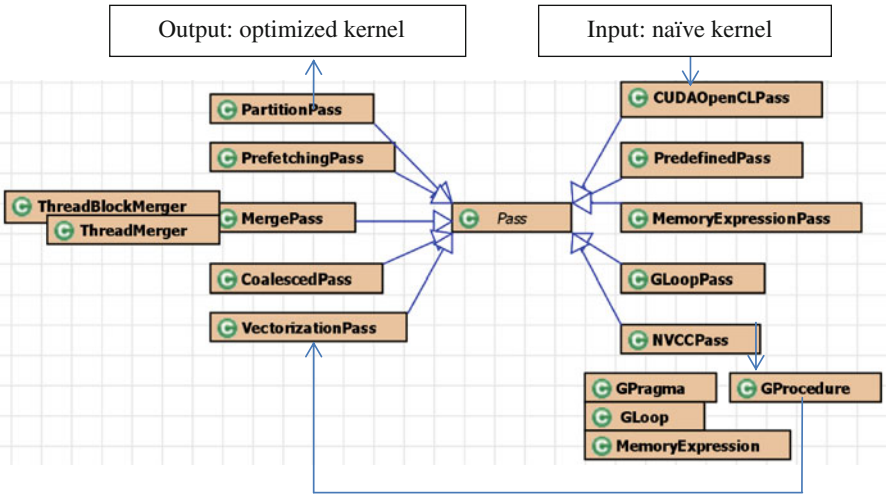

Fig. 3 The class diagram of our implementation of the compiler

```
#define A(y,x) A[(y)* width+(x)]
#define globalDimY 1
__global__ void mv_naive(float *A, float *B, float *C, int width) {
   float sum = 0;
   for (int i=0; i< width; i=i+1) {
      float a;
      float b;
      a = A(idx, i);
      b = B[i];
      sum += a*b;
   }
   C[idx] = sum;
}
```

Fig. 4 Naive implementation of MV

(1) OpenCL to CUDA pass (CUDAOpenCLPass). Since our compiler uses the intermediate representation following the CUDA style, we convert OpenCL code into CUDA code in this pass to facilitate code optimizations. Because the naive version of MV is CUDA code, this step is bypassed.
(2) Predefined pass (PredefinedPass). To simplify the compiler implementation, we use unified variables to express the internal variables of CUDA or OpenCL. For example, the variable 'idx' in our compiler is the same as (blockIdx.x*blockDim.x + threadIdx.x) in the CUDA code or get_global_id(0) in the OpenCL code. The compiler adds macro like "#define idx (blockIdx.x*blockDimX+threadIdx.x)" for CUDA kernels and "#define idx get_global_id(0)" for OpenCL code to express such correspondence. Furthermore, while Cetus uses Procedure as the object for

the kernel procedure, our compiler adds additional attributes to the Procedure to capture the distinctive features of GPU programs. For example, for the naive kernel, the compiler considers that the thread block dimension is (1, 1) by adding two macros "#define blockDimX 1" and "#define blockDimY 1" to the kernel procedures automatically unless the application developers manually set these values. Because the kernel procedure of MV has only one dimension, the attribute "globalDimY" is set to 1 for the naive version as shown in Fig. 4. The compiler does not change "globalDimY" when it performs optimizations. When the CPU code invokes the GPU programs, it utilizes these parameters.

(3) Loop pass (GLoopPass). Our compiler introduces additional functions to the original loop object in Cetus. It identifies all the loops, which include the global memory accesses, and analyses the impacts of the loops on these global memory accesses. It also simplifies loop transformation when the compiler applies the optimization passes. In the MV kernel, the variable "*i*" is a loop iterator.

(4) Memory expression pass (MemoryExpressionPass). First, the compiler identifies all the global memory arrays from the parameters of the kernel procedure declaration such as "float* A", "float* B' in MV kernel. Second, the compiler tries to convert the global memory accesses into two-dimensional memory accesses in the intermediate representation if possible. The reason is that the CUDA global memory can only present multi-dimensional data as a one-dimensional array. Such conversion is helpful to our optimization passes to determine data sharing/reuse. In the case of MV as shown in the Fig. 4, the access A[(idx)*width+(i)] is mapped to A(idx, i). There are several reasons for such mappings: (1) this macro definition provides correct grammar for vendors' compilers so that the vendors' compilers can accept the kernels as inputs without modification, while accesses such as A[idx][i] is incorrect because the global memory array is one dimension; (2) such an expression can clearly express the data accesses of the algorithm and it is convenient for our compiler to generate the texture memory version from the global memory version. The reason is that the compiler only needs to change the procedure declaration and the macro to access different types of memory. A developer can also use one dimension accesses such as A[i*width+threadId.x], for which the compiler will recognize as one-dimension array accesses rather than two-dimensional ones. This may result in missing opportunities when we analyze data reuse along the Y-direction. As shown in Fig. 5, there are three memory access types used in CUDA and OpenCL programs. But in our compiler intermediate representation, the only expression is A(y, x) as shown in the Fig. 4 and the compiler can apply different optimizations without dealing different memory access types. (3) The compiler decouples the indices of global memory accesses into a combination of constant indices, predefined indices, loop indices, and unresolved indices. For example, in the global memory access 'a[idx][i+5]', '5' is identified as a constant index, idx is identified as a predefined index and i is identified as a loop index assuming that the memory access is within a loop with i as the index variable. With these indices, the compiler knows that the access 'a[idx][i+5]' has the same address for the threads along the Y direction because it does not have 'idy' or other indices, which have different values for threads along the Y direction. In the MV kernel, the expression A(idx, i) is a two-dimension array access. Its Y

```
(a) float a = read_imagef(imgA, sampler, (int2)(y, x))
(b) float a = A[y*width+x];
(c) float a = tex2D(texA, x, y);
```

Fig. 5 Different memory access types: **a** Texture memory in OpenCL; **b** Global memory access in OpenCL and CUDA; **c** Texture memory access in CUDA

```
ptxas info    : Compiling entry function '_Z17mv_naive_fPfjj' for 'sm_20'
ptxas info    : Used 6 registers, 48 bytes cmem[0]
```

Fig. 6 Sample of NVCC compiler output

dimension has a predefined index "idx" and its X dimension has a loop index "i". The compiler handles the texture memory using the similar approach.

(5) NVCC pass (NVCCPass). In order to perform optimizations effectively, reducing the search space of optimization parameters in particular, our compiler needs to know the accurate register usage and shared memory usage of the kernels. To do so, our compiler invokes the vendor's compiler to compile the kernel to obtain such resource usage information. The compiler adds these two attributes to the kernel procedure. The Fig. 6 shows the output from NVCC compiler for the CUDA kernel 'mv_naive' and the kernel needs 6 registers and no shared memory is used.

(6) Postprecessor pass. This pass invokes other passes to translate the GPU intermediate representation back to the CUDA intermediate representation for the final output of the kernels. For example, the compiler uses A[idx][i] to present the memory accesses to array A as intermediate representation when it applies optimization passes. For the final output, it needs to be converted to A(idx, i) and mapped to A[(idx)*width+(i)].
(7) CUDA to OpenCL pass (CUDAOpenCLPass). If OpenCL kernels are preferred as the output, this pass is used to translate the CUDA intermediate representation into OpenCL.

The class implementation of these functional passes is shown in Fig. 3. One GPU kernel function is presented as a GProcedure object, which includes the GLoop, GPragma, and MemoryExpression objects. The GLoop object is the intermediate representation of the loops in the kernel, and the GPragma object represents information about thread block configuration. The MemoryExpression object is the intermediate representation of the memory accesses. Then, the compiler applies GPU optimization passes to the kernel function (GProcedure) to improve the performance of the kernel. The implementation of the GPU optimization passes are as follows. The algorithm of each optimization step is discussed in detail in [14]. Here, we focus on how the optimizations are implemented as a sequence of compiler passes.

(1) Vectorization pass (VectorizationPass). Because the data types of memory accesses may have significant impact on bandwidth utilization, the compiler checks data

```
#define A(y,x) read_imagef(A, imageSampler, (int2)(y,x))
#define B(y,x) read_imagef(B, imageSampler, (int2)(y,x))
#define globalDimY 1
__kernel void mv_vec(__read_only image2d_t    A,
                     __read_only image2d_t B,      __global float* C, int width) {
  float sum = 0;
  for (int i=0; i< width/4; i=i+1) {
    float a;
    float b;
    a = A(idx, i);
    b = B(0, i);
    sum += a.x*b.x;
    sum += a.y*b.y;
    sum += a.z*b.z;
    sum += a.w*b.w;
  }
  C[idx] = sum;
}
```

Fig. 7 Code after vectorization for MV using texture memory

accesses in a kernel procedure to see whether they can be grouped into a vector type data access. If the global memory is to be used on NVIDIA GPUs, the Vectorization pass is bypassed as vectorization does not improve memory access bandwidth on NVIDIA GPUs. On the other hand, if the texture memory is to be used, using vector data type is critical for the performance and the code after vectorization for MV is shown in Fig. 7.

(2) Memory Coalescing pass (CoalescedPass). GPGPU requires the threads follow very strict patterns to achieve high memory bandwidth. The compiler detects the memory access pattern and converts non-coalesced global memory accesses to coalesced ones. Figure 8 shows the MV code after the Memory Coalescing pass. Because the accesses for both array A and array B are based on loop iterator "i", they are not coalesced accesses. The compiler unrolls the loop, loads the data into shared memory and then accesses the data from shared memory.

(3) Thread block merge and thread merge pass (MergePass). There are two ways to reduce memory accesses: reuse data either in shared memory or in registers. The thread block merge will combine the threads in multiple thread blocks into one thread block so that the data in the shared memory can be reused by more threads. The other way to improve the data reuse is thread merge, which combine multiple threads into one thread. After the thread merge, not only the data in the shared memory, but also the data in the registers can get better reuse. When the compiler applies the thread block merge, the workload of each thread block increases and the reuses in shared memory can be increased; when the compiler applies the thread merge, the workload of each thread increases and the reuses in registers is increased. In this pass, thread-block merge determines the workload for each thread

```
for (i=0; i<width; i=(i+16)) {
  __shared__ float shared2[16];
  __shared__ float shared1[16][17];
  shared2[(0+tidx)]=B[i+tidx];
  for (l=0; l<16; l=(l+1))
    shared1[(0+l)][tidx]= A[((idx-tidx)+l)][(i+tidx)];
  __syncthreads();
  for (int k=0; k<16; k=(k+1)){
    sum+=(shared1[tidx][k]*shared2[k]);
  }
__syncthreads();
}
C[idx] = sum;
```

Fig. 8 Code after memory coalescing for MV

block while thread merge decides the workload for each thread. After applying the thread (block) merge, the compiler collects the usage of the register and shared memory using the NVCC pass to calculate the number of threads and number of thread blocks supported in each SM based on the hardware specification. This way, the compiler can choose the version with sufficient number of threads. Because the merge pass has these two options, we have ThreadMerger and ThreadBlockMerger as two merger classes.

(4) Data prefetching pass (PrefetchingPass). Data prefetching is a well-known technique to overlap memory access latency with computation. For instance, many-thread aware prefetching for GPUs is proposed in [4]. In our compiler, the prefetching technique is used to prefetch data for the future iterations in a loop. Because our compiler uses the register and shared memory aggressively, prefetching into registers or shared memory may hurt thread-level parallelism. Therefore, this pass has limited performance impact. Using prefetch instructions eliminates this problem as the prefetched data are not bound to registers or shared memory resources.

(5) Removing memory partition camping pass (PartitionPass). Because GPGPU prefers threads to distribute global memory accesses to different partitions of off-chip memory, our compiler applies several code transformations to eliminate memory partition camping. Figure 9 shows the code after removing partition camping by giving different partition offset for different thread blocks. The bidx is the block id of the thread block and one partition is 256 bytes.

While our compiler supports these GPU optimization passes, the compiler will choose different passes for global memory and texture memory accesses. The compiler will always apply vectorization pass for texture memory because of the high bandwidth of vector types for texture memory, but it only applies vectorization for global memory on AMD GPUs because the NVIDIA GPUs do not require vector access for global memory to achieve good bandwidth. The memory coalescing pass is ignored for texture memory. The reason is that the data of the non-coalesced memory access can be

```
int offset = bidx*64;
for (i= offset; i<width+ offset; i=(i+16)) {
   __shared__ float shared2[16];
   __shared__ float shared1[16][17];
   int i1 = i% width;
   shared2[(0+tidx)]=B[i1+tidx];
   for (l=0; l<16; l=(l+1))
      shared1[(0+l)][tidx]= A[((idx-tidx)+l)][(i1+tidx)];
   __syncthreads();
   for (int k=0; k<16; k=(k+1)){
      sum+=(shared1[tidx][k]*shared2[k]);
   }
__syncthreads();
}
C[idx] = sum;
```

Fig. 9 Code after removing partition camping for MV

cached in the texture cache and memory coalescing does not have significant impact on the texture memory. The texture cache also can reduce the benefit of the shared memory for data reuses. Since the purpose of thread block merge is to better utilize the shared memory, thread block merge is not necessary if the kernels use texture memory. Thread merge, in comparison, is effective for both texture memory and global memory. Similarly to coalescing pass, the benefit of removing memory partition camping pass is also small for texture memory.

4 Performance Evaluation

To evaluate our compiler framework for GPUs, we used NVIDIA GTX 480 GPUs with CUDA SDK 3.2 and a 64-bit bit Red Hat enterprise Linux 5.4 operating system. For AMD/ATI HD5850 GPUs, we used AMD/ATI Stream SDK 2.3 on a 32-bit Windows 7 operating system. Our compiler source code, the naïve kernels, and the optimized kernels are available at [15].

In Figs. 10 and 11 we report the performance gains of our optimized code over the naïve kernels on GTX 480 and HD5870, respectively. The global memory is used in this experiment. From the figures, we can see that the compiler significantly improves the performance of various naïve kernels using the proposed optimizations: 3.2X on GTX 480, 4.9X on HD 5870 on average using the geometric mean. The optimized MV achieves a 12.4X speedup on GTX 480 and a 36.4X speedup on HD 5870.

We also report the speedups of the benchmarks using texture memory as the storage of the data on both NVIDIA GTX 480 GPU and AMD HD 5870 GPU in Figs. 12 and 13, respectively. As we can see from the figures, the optimized kernels by our compiler achieves 2.5 times on GTX 480 and 7.2 times on HD 5870 over the naive kernels on average with the geometric mean. Because the benchmarks reduction and strsm need

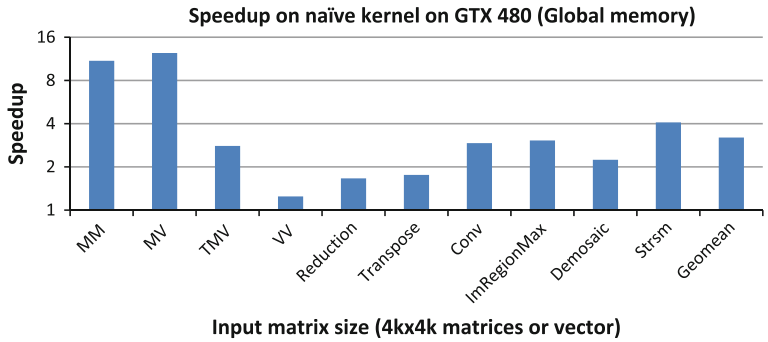

Fig. 10 The speedups of the optimized kernels over the naive ones on GTX 480 (Global Memory)

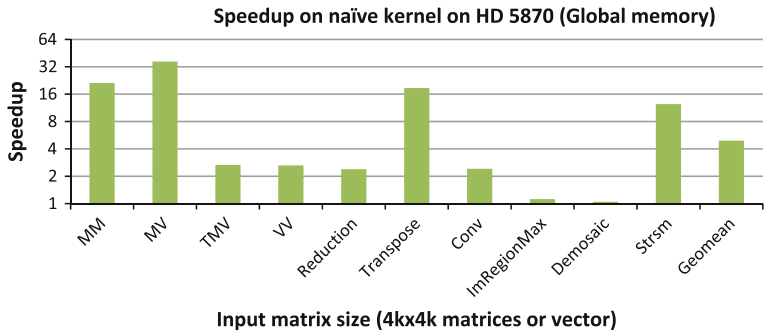

Fig. 11 The speedups of the optimized kernels over the naive ones on HD 5870 (Global Memory)

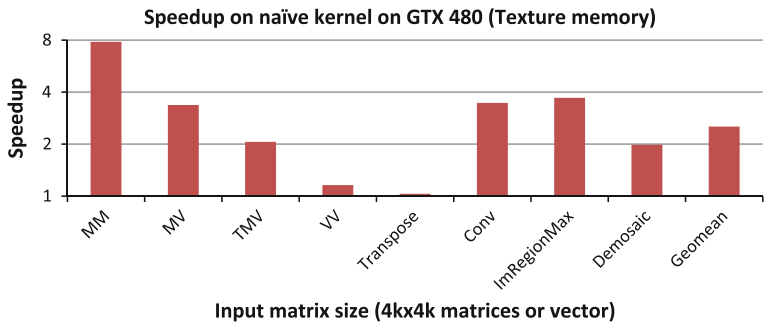

Fig. 12 The speedups of the optimized kernels over the naïve kernels on GTX 480 (Texture Memory)

to read from and write to the same memory and the texture can be read only or write only, the texture memory is not a viable option for these two benchmarks.

To illustrate the effectiveness of our compiler, we also compare the performance of optimized kernels by our compiler to the NVIDIA CUBLAS library. Our compiler outperforms CUBLAS 3.2 for 4k by 4k inputs on GTX 480, with an average of 17.5 % for MV, MM, TMV, VV, Reduction and Strsm [13].

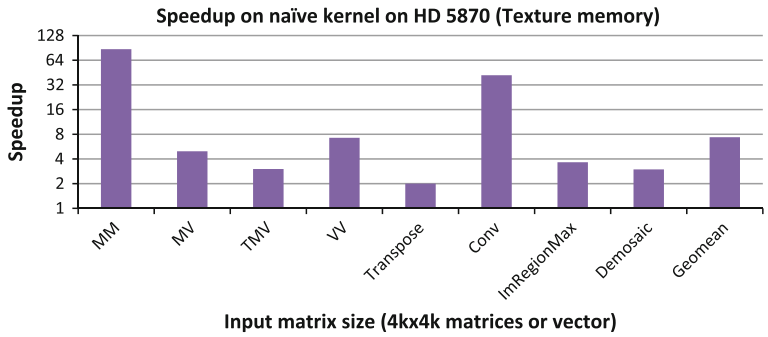

Fig. 13 The speedups of the optimized kernels over the naïve kernels on HD 5870 (Texture Memory)

5 Conclusions

In this paper, we present our experience in developing a compiler framework to optimize GPGPU programs using the Cetus infrastructure. A set of compiler techniques is implemented in our compiler to improve GPU memory usage and to judiciously distribute workloads among threads and thread blocks. Our experimental results show that the optimized code achieves very high performance, often superior to manually optimized programs. As a source-to-source compiler framework, Cetus enables us to implement code optimizations on high level language without the details of low level language like assembly. Optimizations at the high level language can be effective for different low level implementations. To facilitate further development of our GPGPU compiler, we expect Cetus to include the OpenCL and CUDA support internally or some extension interfaces for parallel programming languages. We are also interested in static single assignment, which can simplify data dependency analysis.

Acknowledgement This work is supported by the National Science Foundation, CAREER award CCF-0968667.

References

1. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.W.: An adaptive performance modling tool for GPU architectures. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2010)
2. Lee, S.-I., Johnson, T., Eigenmann, R.: Cetus—an extensible compiler infrastructure for source-to-source transformation. In: Proceedings of Workshops on Languages and Compilers for Parallel Computing (2003)
3. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2009)
4. Lee, J., Lakshminarayana, N.B., Kim, H., Vuduc, R.: Many-thread aware prefetching mechanisms for gpgpu applications. IEEE/ACM International Symposium on Microarchitecture (2010)
5. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU programs optimization. In: Proceedings of IEEE International Parallel and Distributed Processing, Symposium (2009)
6. NVIDIA CUDA C Programming Guide 3.1. (2010)
7. OpenCL. http://www.khronos.org/opencl/
8. Ruetsch, G., Micikevicius, P.: Optimize Matrix Transpose in CUDA. NVIDIA (2009)

9. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S., Stratton, J.A., Hwu,W.W.: Optimization space pruning for a multi-threaded GPU. International Symposium on Code Generation and Optimization (2008)
10. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
11. Stratton, J.A., Stone, S.S., Hwu, W.W.: MCUDA: An Efficient Implementation of CUDA Kernels on Multicores. IMPACT Technical Report IMPACT-08-01, UIUC, Feb (2008)
12. Ueng, S., Lathara, M., Baghsorkhi, S.S., Hwu, W.W.: CUDA-lite: Reducing GPU programming complexity. In: Proceedings of Workshops on Languages and Compilers for Parallel Computing (2008)
13. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. ACM SIGPLAN conference on Programming Language Design and Implementation (2010)
14. Yang, Y., Xiang, P., Kong, J., Mantor, M., Zhou, H.: A unified optimizing compiler framework for different GPGPU architectures. In: ACM Transactions on Architecture and Code, Optimization (2012)
15. Yang, Y., Zhou, H.: http://code.google.com/p/gpgpucompiler/