DEVELOPING A HIGH PERFORMANCE GPGPU COMPILER USING CETUS

YI YANG, North Carolina State University

HUIYANG ZHOU, North Carolina State University

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. The implementation of our compiler is facilitated with the Cetus infrastructure. 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 can clearly represent memory access patterns and thread configurations. The CUDA language support pass is derived from MCUDA. A series of optimization passes improve the performance of the kernels by adapting the kernels to the GPGPU architecture. Our experiments show that the optimized code achieves very high performance, either superior or very close to highly fine-tuned libraries.

1. INTRODUCTION

The high computational power and memory access bandwidth of state-of-art graphics processing units (GPU) have made them appealing for high performance computing. However, it's big challenge to develop high performance GPGPU code as application developers need to know how to utilize the GPU hardware resources effectively. We present our GPGPU compiler as a solution, which takes naive GPU kernels as inputs and generates optimized kernels to relieve the application developers of low-level hardware-specific performance optimizations.

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). Each SM/SIMD engine contains multiple streaming processors / cores. Threads in GPUs follow the single-program multiple-data (SPMD) program execution model and they 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. Every SM or SIMD has a number of registers, which is private to each thread, shared memory, which is visible to a thread block/group, and global memory, which is used by all the threads. To full utilize the resource on GPUs, two issues need to be considered: (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. We develop our compiler based on Cetus [1] to address these two issues.

Our compiler achieves the following goals. (1) It enables the application developers to focus on algorithm-level issues rather than low-level hardware-specific performance optimizations. (2) It includes 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) It is highly effective and the programs optimized by our compiler achieve very high performance, often superior to manually optimized codes.

2. IMPLEMENTATION

Our compiler leverages MCUDA [2] which adds CUDA language support to Cetus [1] and translates the kernel code into intermediate representation with CUDA support. Our compiler adds additional passes to translate the intermediate representation to our GPU intermediate representation and applies GPU optimization passes 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 the Figure 1, 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, loop structures, thread configurations and thread block dimensions. Then the compiler applies

Our compiler leverages MCUDA [2] which adds CUDA language support to Cetus [1] and translates the kernel code into intermediate representation with CUDA support. Our compiler adds additional passes to translate the intermediate representation to our GPU intermediate representation and applies GPU optimization passes 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

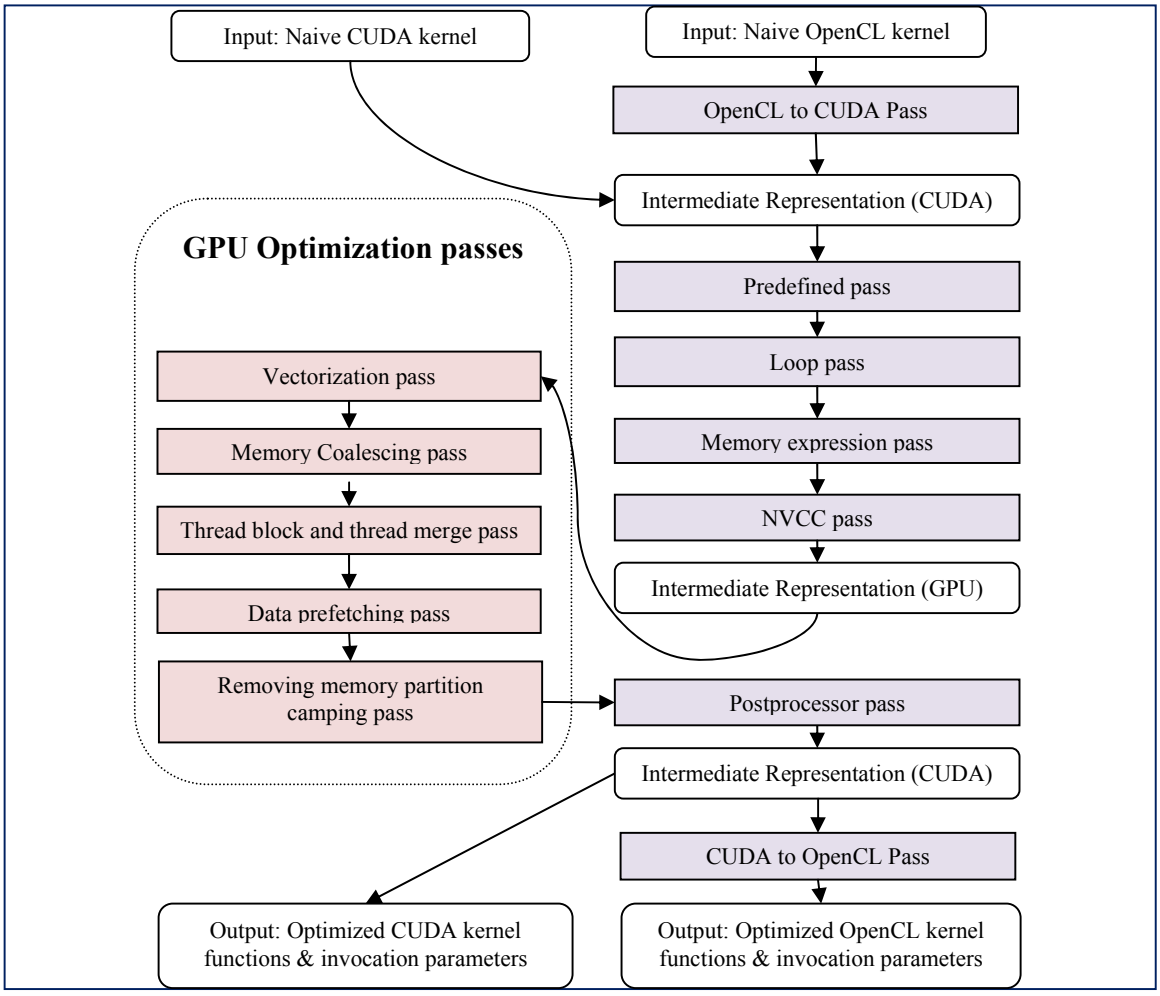

Figure 1. The framework of proposed compiler

kernels. Instead, they are needed for the preprocessor, GPU optimization passes and the postprocessor. As shown in the Figure 1, 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, 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 or OpenCL.

The functional passes are summarized as follows. We use matrix vector multiplication (MV) as a case study to illustrate the compilation steps.

(1) OpenCL to CUDA pass. Since our compiler uses the intermediate representation following the CUDA style, we convert the 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. To simplify the compiler, we use unified variables to express the internal variables of CUDA or OpenCL. For example, 'idx' in the 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 some attributes the procedure because of 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 "globalDimY" is set to 1 for the naive version as shown in Figure 2a. The compiler does not change the globalDimY when it performs

```
#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;
}
```

(a) Naive implementation of MV

```
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;
```

(b) Code after memory coalescing

```
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;
```

(c) Code after removing partition camping

Figure 2. Compiler optimization for MV

optimizations. When the CPU code invokes the GPU programs, it utilizes these parameters.

(3) Loop pass. Our compiler provides 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 makes the loop transformation easier when the compiler applies the optimization passes. In the MV kernel, the variable "*i*" is loop iterator.

(4) Memory expression pass. 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 a matrix array as a one-dimensional array. Such conversion is helpful to our optimization passes to determine data reuse. In the case of MV as shown in the Figure 2.a the access A[(idx)*width+(i)] is mapped to A(idx, i). There are several reasons for such mappings: 1) this macro definition has 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 better express the algorithm and it is convenient for our compiler to generate the texture memory version from the global memory version, because the compiler only needs to change the procedure declaration and the macro for memory access. Third, 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 indentified as a predefined index and i is identified as a loop index assuming that the memory access is in 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 A(idx, i) is a two dimension array. Itse Y dimension has a predefined index "idx" and its X dimension has a loop index "i".

(5) NVCC pass. Our compiler needs to know the accurate register usage and shared memory usage of the kernels. Therefore, our compiler invokes the vendor's compiler to compile the kernel to obtain the resource usage information. Such information is very useful to limit the search space of optimized kernels. The compiler adds these two attributes to the kernel procedure.

(6) Postprecessor pass. This pass translates 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 access for 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. If the optimized OpenCL kernel is preferred, we translate the CUDA intermediate representation into OpenCL.

The GPU optimization passes are as follows and the detailed implementation is presented in [3].

(1) Vectorization pass. Because the data type of memory accesses may have significant impact on bandwidth utilization, the compiler first checks data accesses inside a kernel procedure to see whether they can be grouped into a vector type data access. The Vectorization pass is ignored for MV on NVIDIA GPUs as vectorization does not improve memory access bandwidth on NVIDIA GPUs.

(2) Memory Coalescing pass. GPGPU requires the threads follow very strict patterns to achieve high global

memory bandwidth. The compiler detects the memory access pattern and converts non-coalesced memory accesses to coalesced ones. Figure 2b shows the MV code after the Memory Coalescing pass. Because the accesses for both A and B are based on loop iterator "i", which are not coalesced memory 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. There are two ways to reduce memory accesses: reuse data either in shared memory or in registers. When the workload of each thread block increases, the reused data in shared memory can be increased; when the workload of each thread increases, the reuse in registers is increased. In this pass, thread-block merge determines the workload for each thread block while thread merge decides the workload for each thread. The detailed discussion about thread merge and thread block merge can be found in [3].

(4) Data prefetching pass. Data prefetching is a well-known technique to overlap memory access latency with computation. It is implemented in our compiler. Because GPUs use multiple threads to overlap memory access latency, this step is skipped by default.

(5) Removing memory partition camping pass. 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 2c 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.

3. PERFORMANCE EVALUATION

In our experiments, we used both 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 [4].

From Figures 3 and 4, 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.

4. CONCLUDING REMARKS

In this paper, we present our experience in developing a compiler framework to optimize GPGPU programs using Cetus. As a source-to-source compiler framework, Cetus enables researchers like us to implement code optimizations on high level language without the knowledge of low level language like assembly. Optimizations at the high level language can be effective for different low level implementations. As shown in our work, the optimized OpenCL kernels can be effective for both NVIDIA and AMD platforms. To facilitate further development on our GPGPU compiler, we expect Cetus to add the OpenCL and CUDA support internally or some extension interfaces for parallel languages. Another important feature is static single assignment, which can simplify data dependency analysis.

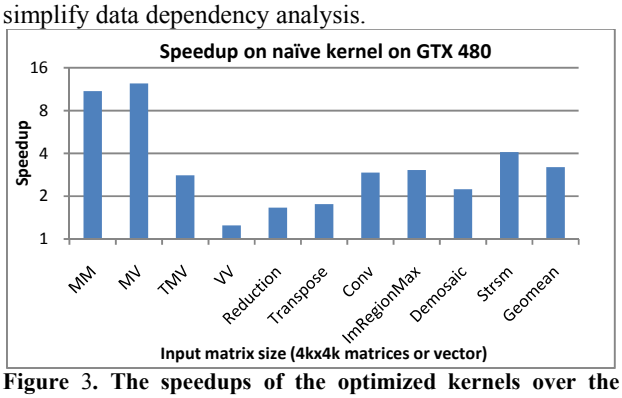

Figure 3. The speedups of the optimized kernels over the naive ones on GTX 480

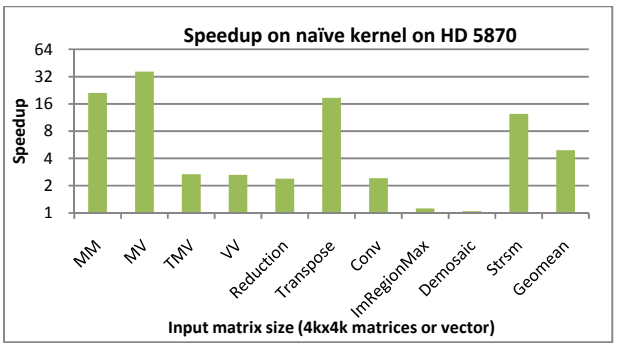

Figure 4. The speedups of the optimized kernels over the naive ones on HD 5870

5. References

[1] Lee, S.-I., Johnson, T. and Eigenmann, R. 2003. Cetus – an extensible compiler infrastructure for source-to-source transformation. In Proceedings of Workshops on Languages and Compilers for Parallel Computing (LCPC'03). 539–553.

[2] Stratton, J. A., Stone, S. S., and Hwu, W. W. 2008. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. The 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC'08). 16-30.

[3] Yang, Y., Xiang, P., Kong, J. and Zhou, H. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. The ACM SIGNPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10). ACM. 86-97.

[4] Yang, Y. and Zhou, H. 2010. GPGPU compiler. http://code.google.com/p/gpgpucompiler/