memCUDA: Map Device Memory to Host Memory on GPGPU Platform*

Hai Jin, Bo Li, Ran Zheng, Qin Zhang, and Wenbing Ao

Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@mail.hust.edu.cn

Abstract. The Compute Unified Device Architecture (CUDA) programming environment from NVIDIA is a milestone towards making programming many-core GPUs more flexible to programmers. However, there are still many challenges for programmers when using CUDA. One is how to deal with GPU device memory, and data transfer between host memory and GPU device memory explicitly. In this study, source-to-source compiling and runtime library technologies are used to implement an experimental programming system based on CUDA, called memCUDA, which can automatically map GPU device memory to host memory. With some pragma directive language, programmer can directly use host memory in CUDA kernel functions, during which the tedious and error-prone data transfer and device memory management are shielded from programmer. The performance is also improved with some near-optimal technologies. Experiment results show that memCUDA programs can get similar effect with well-optimized CUDA programs with more compact source code.

Keywords: GPU, CUDA, memory mapping, programming model.

1 Introduction

In the past decade, the era of multi-core computer processor is coming due to power and performance limitations of VLSI design. As a new computing platform, GPGPUs become more and more attractive because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and general-purpose instruction sets. It is also important to revisit parallel programming model and tools for general-purpose computing on GPU. Programming on GPU is so complex that it is a significant burden for developers. NVIDIA released their new product GTX architecture GPU with CUDA support in 2007 [1], which is a flagship tool chain and has become a *de facto* standard towards the utilization of massively parallel computing power of Nvidia GPUs.

* This work is supported by National 973 Basic Research Program of China under grant No.2007CB310900, the Ministry of Education-Intel Information Technology special research fund (No.MOE-INTEL-10-05) and National Natural Science Foundation of China (No.60803006).

C. Ding, Z. Shao, and R. Zheng (Eds.): NPC 2010, LNCS 6289, pp. 299–313, 2010.

CUDA allows the programmer writing device code in C functions called kernels. Compared to a regular function, a kernel is executed by many GPU threads in a *Single-Instruction-Multiple-Thread* (SIMT) fashion. Each thread executes the entire kernel once. Launching a kernel for GPU execution is similar to calling the kernel function. Local references in a kernel function are automatically allocated in registers (or local memory). References in device memories must be created and managed explicitly through CUDA runtime API. The data needed by a kernel must be transferred from host main memory into device memory before the kernel is launched, and result data also needs to be transferred back to host main memory after kernel execution. Note that these data transfer between host memory and device memories are both performed in an explicit manner.

For CPU-GPU heterogeneous computing architecture, the address spaces of host CPU and device GPU are separate with each other. The developer has to use CUDA APIs to manage device memory explicitly to realize the CPU accessing GPU data, including transfer data between a CPU and a GPU. Thus, it is very difficult to manage data transfer between host memory and various components of device memory, and manually optimize the utilization of device memory. The programmer has to make decisions on what data to move to/from device memory, when and how to move them. Practical experience shows that the programmer needs to make significant tedious and error-prone code changes. The explicit data transfer and communication between CPU and GPU has become one of the biggest obstacles when programming on GPU. This paper introduces a scheme that could improve the comprehensive performance of the system.

The present CUDA only supports data-parallel and not task-parallel, thus, the GPU is always exclusively used by one kernel. In fact, CUDA does not even allow independent kernels from the same process to run concurrently. When the kernel is flying on GPU, there are few high-efficiency mechanisms to support any kinds of communication between CPU and GPU. Due to the currently available NVIDIA GTX architecture GPU does not support task-level parallel (the next generation NVIDIA GPGPU Fermi [18] is claimed to support task-level parallel), the whole kernel invocation procedure must strictly pass through three following phrases: a) performing input data transfer from host memory to device memories (global, constant and texture memory); b) invocating the kernel to process the data; c) performing result data transfer from device memories to host memory. Generally, there are another two stages running on CPU, preprocessing or post-processing stages. As a trivial example, Fig. 1 illustrates the implementation of CUDA-based vector addition.

The above mentioned data transfers between host memory and device memory in CUDA program is performed explicitly. It not only increases development time, but also makes the program difficult to understand and maintain. Moreover, the management and optimization on data allocation in GPU device memories involves heavy manipulation of communication, which can easily go wrong, prolonging program development and debugging.

With these challenges in mind, we design *memCUDA*, a high-level directive-based language for CUDA programming, which can automatically map device memory to host main memory. Just using several pragma directives, the developer can focus on the

usage of host main memory instead of device memory. Therefore, it shields device memory from programmer and relieves the programmer's burden out of manually data transfer between host memory and device memory. Therefore, it supports the same programming paradigm familiar with CUDA. We have implemented a source-to-source compiler prototype that translates a *memCUDA* program to the corresponding CUDA program. Experiments with six standard CUDA benchmarks show that the simplicity and flexibility of *memCUDA* are provided and come at no expense to performance.

```
//Comput vector sum c = a + b
__global__ void vecAdd (float *a, float *b, float *c) {
    Int I = threadIdx.x + blockDim.x * blockIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    /* allocat h_a, h_b, and h_c with size N,
       And initialize host(CPU) memory */
     float *h_a = ... , *h_b = ... , *h_c = ...;

   //allocat device (GPU) memory.                                  Phrase 1
   float *d_a, *d_b, *d_c;
   cudaMalloc( (void **) &d_a, N*sizeof(float) );
   cudaMalloc( (void **) &d_b, N*sizeof(float) );
   cudaMalloc( (void **) &d_c, N*sizeof(float) );

   //copy host memory to device memory
   cudaMemcpy( d_a, h_a, N*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy( d_b, h_b, N*sizeof(float), cudaMemcpyHostToDevice);

   //execute the kernel on N/256 blocks of 256 threads each
   vecAdd <<< N/256, 256>>> (d_a, d_b, d_c);                     Phrase 2

   //copy the result data from device back to host memory
   cudaMemcpy( h_c, d_c, N*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);                                                Phrase 3
    ......
}
```

Fig. 1. CUDA vector addition example codelet

The remainder of this paper is organized as follows. In section 2, we review some related works. In section 3, we give a briefly introduction of the main framework of *memCUDA*. Section 4 introduces *memCUDA* language directives in details. Section 5 gives a description of *memCUDA* runtime supports. The sixth section, the *memCUDA* implementation details are introduced and we conduct some experiments to validate the performance of *memCUDA* and also give concluding remarks and research directions for future work in section 7.

2 Related Work

To make GPU programming more user-friendly, there has been lots of efforts to improve the development of new programming frameworks, among which RapidMind [2] is a well-known one and has been commercialized. Existing general purpose programming languages for the GPU are based on the stream processing mode. These languages include Brook [11][14], Sh [17], and etc. There are also some works using the source-to-source technology to improve the programmability or performance [15][16].

BSGP (*Bulk Synchronous GPU Programming*) [6], a new programming language for general-purpose computation on GPU, is easy to read, write, and maintain. BSGP is based on BSP (*Bulk Synchronous Parallel*) model. Multicore-CUDA (MCUDA) [12] is a system that efficiently maps the CUDA programming model to a multicore CPU architecture. They use the source-to-source translation process that converts CUDA code into standard C language that interfaces to a runtime library for parallel execution.

CUDA-lite [4] is a memory optimization tool for CUDA programs, which is also designed as a source-to-source translator. With some annotations, the compiler automatically transforms the flat program to CUDA code with the utilization of *shared memory*. This tool takes a CUDA program only with global memory, and optimizes its memory performance via access coalescing. This work puts the shield the *shared memory* from programmer. Our work could be considered as a supplement of CUDA-lite. CUDA-lite aims at automatic memory optimization, but it still requires the programmer to write a full-fledged data movement between host memory and device memory function in program. In contrast, *memCUDA* aims at shielding data movements and device management for programmer.

A directive-based language called hiCUDA [7] is defined for programming NVIDIA GPUs. The language provides programmers high-level abstractions to carry out the tasks mentioned above in a simple manner, and directly to the original sequential code. The use of hiCUDA directives makes it easier to experiment with different ways of identifying and extracting GPU computation, and managing device memory.

The syntax of *memCUDA* directives bears some similarity to hiCUDA directives. Our work partly is inspired by hiCUDA. However, there are significant differences between the two studies. hiCUDA adopts source-to-source compiling technology like macro replacement without any optimization. Although data movement APIs of CUDA are warped by hiCUDA directives, it also needs programmer consider data movement direction and explicit data movement. Our work focuses on memory mapping from device memory to main memory. The device is completely shielded from programmer. Moreover, the asynchronous execution optimization is also performed in *memCUDA*.

3 Main Framework

Fig. 2 shows the software architecture of *memCUDA*. At application level, *memCUDA* provides APIs and pragma directives to programmers. With the help of the API to explicitly express the usage of host main memory, programmers are alleviated from difficult and explicit data transfer and device memory usage, instead of focusing on performance tuning. Similar to OpenMP programming grammar, *memCUDA* API is built on top of C/C++ so that it can be easily adopted.

Below the application layer is *memCUDA* system layer, which consists of a source-to-source transformer, source code generator and a number of runtime libraries. The source-to-source transformer translates *memCUDA* pragma directives into native CUDA codes with runtime library APIs. At the same time, it also decides the

near-optimal technology, which includes utilizing the page-locked memory to achieve the overlap between kernel execution and data transfer through an adaptive algorithm. The lower NVIDIA NVCC is invoked to compile the transformed CUDA code to produce the binary code, which could run on the GPGPU platform. The runtime libraries in Fig. 2 are implemented as wrapper calls to the libraries, which is used to dynamic memory mapping.

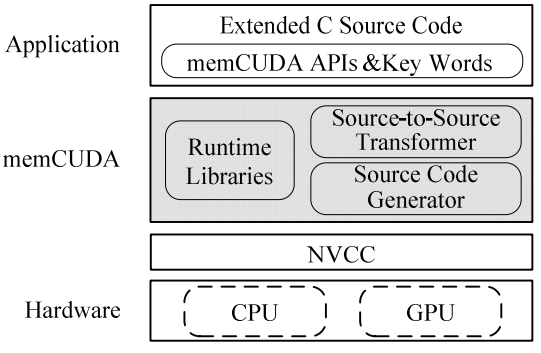


Fig. 2. Software framework of *memCUDA*

Current implementation of *memCUDA* is built on top of NVIDIA CUDA compiler NVCC for GPU. Instead of directly generating CPU and GPU native machine codes, *memCUDA* compiler generates CUDA source codes from *memCUDA* codes on the fly and then uses the system compilers to generate the final machine codes.

4 memCUDA Pragma Directives

The *memCUDA* directives are specified using the pragma mechanism provided by C and C++ standards. Each directive starts with *#pragma memcuda* and is case-sensitive. Preprocessing tokens following *#pragma memcuda* are subject to macro replacement with the runtime libraries and original CUDA APIs.

In naive CUDA programming framework, device memory is typically allocated with *cudaMalloc()* and freed with *cudaFree()*. Data movements between host memory and device memory are typically done with *cudaMemcpy()* and some other APIs. These APIs are all abandoned in *memCUDA* system, instead, they are all invoked implicitly by wrapped as *memCUDA* APIs. Through the simple pragma directives, *memCUDA* high-level compilers perform a source-to-source conversion of a *memCUDA*-based program to an original CUDA program. Currently, *memCUDA* does not support the mapping of texture and constant memory, and also does not support CUDA array structure direct mapping. *memCUDA* current provides four directives: **map**, **remap**, **unmap**, and **update** for global memory mapping, just as following:

- **map** directive performs the functions of establishing the mapping relationship. It indicates which references need to be mapped from device memory to host memory.

The size of data block also should be given. The shape of the references can eventually be directly derived from source code by compiler parser. Then device memory will be allocated. The new record of the mapping also will be inserted into *Memory Mapping Table* (*mmTable* for short, which will be introduced in the next section). In some cases the same data block in host memory needs to be mapped to several counterparts in device memory. Thereby the *map* directive does not perform the data transfer, which will be performed by the update directive.

- **remap** directive indicates refreshing the mapping relationship. In some cases, especially for the applications with multiple kernels, the old device memory block needs to be freed and substituted by a new block of device memory. So the remap operation refreshes the mapping relationship.
- **unmap** directive is for annotating to dissolute the mapping relationship. It will invoke the operation that free data block of device memory. The mapping record in *mmTable* will also be deleted.
- **update** directive triggers data transfer operation between host memory and device memory. The transfer direction is decided by STATE attribute of *mmTable*.

For rigorous programmer, the *update* directive could be combined with *map* or *remap* directive. After the mapping relationship installed, the operation which performed by *update* directive could be executed automatically. That means the host memory block must be initialized before the mapping relationship installed. For the flexibility of programming, we keep the update directive independency of data transfer from the *map* and *remap* directive.

Fig. 3 shows the current form of the pragma directives in *memCUDA*. Each directive manages the lifecycle of mapping relationship between data block in device memory and the correspondent data block in host memory. Since data management in global memory always happens before or after kernel execution, the pragma directives must be placed in host code regions. In contrast, the kernel region does not need to be modified. All the pragma directives are stand-alone, and the associated actions taken by runtime libraries happen at the place of the directive.

```
#pragma memcuda map( reference , size { , reference , size }* )
#pragma memcuda remap( reference { , reference }* )
#pragma memcuda unmap( reference { , reference }* )
#pragma memcuda update( reference { , reference }* )
```

Fig. 3. *memCUDA* pragma directives

For comparison purpose, the original hand-written CUDA version of TPACF in Parboil benchmark suite [5] and *memCUDA* rewritten version both are shown in Fig. 4. For naive CUDA code (see Fig. 4(a)), it needs programmer to allocate device memory and move data from host memory to device memory through invoking CUDA APIs. However, in *memCUDA* version (see Fig. 4(b)), there are no references in device address space. After using the map pragma, device memory allocation and the mapping relationship all are undertaken automatically. Thus, the kernel can use host memory directly. When data requires moving between host memory and device memory, the

update pragma needs to be inserted in the rational place in host code region. Then data movements are all performed by runtime libraries and the movement direction (e.g. from host memory to device memory, or vice versa) determined in the parsing phase of the compiler. Above all, *memCUDA* shields device memory and never exposes it to programmer.

```
……
// allocate cuda memory to hold all points
float * d_x_data;
cudaMalloc((void**) & d_x_data, 3*f_mem_size);

float * d_y_data = d_x_data + NUM_ELEMENTS*(NUM_SETS+1);
float * d_z_data = d_y_data + NUM_ELEMENTS*(NUM_SETS+1);

hist_t * d_hists;
cudaMalloc((void**) & d_hists,
           NUM_BINS*(NUM_SETS*2+1)*sizeof(hist_t));

// allocate system memory for final histograms
Hist_t *new_hists = (hist_t *)
        malloc(NUM_BINS*(NUM_SETS*2+1)*sizeof(hist_t));

// Initialize the boundary constants for bin search
initBinB( &timers );
printf("kick off tpacf on cuda!\n");

// **===-- Kick off TPACF on CUDA--===**
cudaMemcpy(d_x_data, h_x_data, 3*f_mem_size,
           cudaMemcpyHostToDevice);

TPACF(d_hists, d_x_data, d_y_data, d_z_data);

cudaMemcpy(new_hists,d_hists, NUM_BINS*(NUM_SETS*2+1)*
           sIzeof(hist_t), cudaMemcpyDeviceToHost);
……
```

(a) Original CUDA Code

```
……
Hist_t *new_hists = (hist_t *)
        malloc(NUM_BINS*(NUM_SETS*2+1)*sizeof(hist_t));

Int  hist_size = NUM_BINS*(NUM_SETS*2+1)*sizeof(hist_t);

#pragma ecuda map(h_x_data, 3*f_mem_size; new_hists,
                  hist_size)

// Initialize the boundary constants for bin search
initBinB( &timers );

// **====- Kick off TPACF on CUDA--===**
#pragma ecuda update(h_x_data)

TPACF(new_hists, h_x_data, h_y_data, h_z_data);

#pragma ecuda update(new_hists)
……
```

(b) *memCUDA* Code

Fig. 4. Original CUDA code vs *memCUDA* TPACF code

From Fig. 4, we can see that *memCUDA* code is much simpler to write, understand and maintain. The programmer does not need to manage GPU device memory nor use explicit data movement between device memory and host memory. Nonetheless, *memCUDA* supports the same programming paradigm familiar to CUDA programmers.

5 Runtime Support

Runtime support is in charge of memory mapping when the program is running. The lifecycle of the memory mapping could be described through a state transition diagram as shown in Fig. 5.

- **Mapped state:** In this state, the mapping relationship has been established. The data block in host memory and device memory both are allocated. The row which records this mapping relationship is also inserted into *mmTable*.
- **Transferred state:** In this state, the data has been transferred from host/device memory to device/host memory. The transfer direction is determined by the value of STATE attribution in *mmTable*.
- **Destroyed state:** This state means the mapping relationship is terminated.

Responding to these states, the mapping tasks include: *Map()*, *Update()*, *Remap()*, and *Unmap()*. Especially, the *Update()* could perform two directions' data transfer between host memory and device memory. The following of this section describes *memCUDA* runtime libraries.

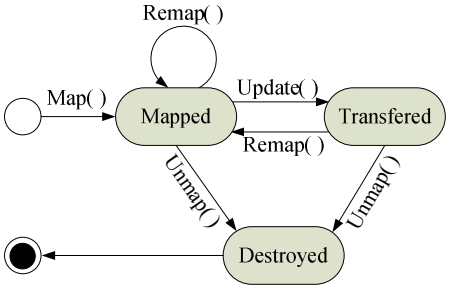


Fig. 5. State transition diagram for data mapping

5.1 Mapping Table Structure

Currently CUDA framework does not support concurrent executions of two kernels, and the size of host memory generally is much larger than device memory, memory mapping is designed as one-to-one mapping between host address space and device address space, shown in Fig. 6. The mapping information is maintained by a structure under a table shape, basically used for looking up and translating operations: *mmTable*, which holds information for optimizing the look up mechanism and implementing the mapping from GPU device memory address space to host memory address space. A row contains following four attributes for mapping mechanisms:

- **BASE_H:** the base address of data block in host memory address space, which is the key attribute of the table;
- **BASE_D:** the base address of the correspondent data block in device memory address space;
- **SIZE:** the size of data block;
- **STATE:** the state of current mapping operation, which is used to determine the next data movement direction from host memory to device memory, or vice versa.

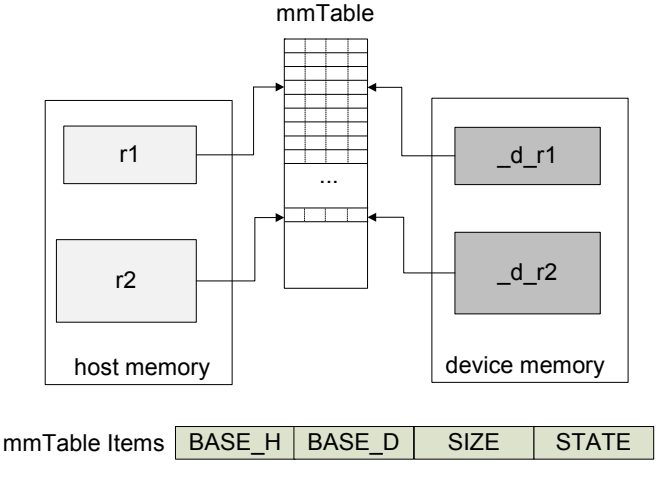


Fig. 6. Illustration of memory-mapping mechanism

5.2 Memory Mapping

The implementation of mapping mechanism is divided into two phases. The first phase takes place before launching the kernel to setup the mapping relationship; the second one occurs inside the mapping runtime system when the kernel flying.

For the first phase, it requires some coordination with the compiler supported by source-to-source code replacement. When the mapping operation pragma is invoked, a data block will be allocated in device memory and the size is the same with its counterpart in host memory. The data will be moved from host main memory to device memory. After that, the mapping relationship will be established. Then a new row will be inserted into *mmTable* to record this mapping relationship, and the attributes of the base address of data block in host device memory address space, the base address of the correspondent data block in device memory address space, and the size of data block and the state of current mapping operation will all be evaluated. The pseudo code is described in Algorithm 1.

Algorithm 1: Map() directive routine

Input: map() directive statement

```
Parse map() directives to get references addresses and their sizes
For each reference r in map() directive:
  //mapping references
  If (mmTbl.contain(r) == false )
    allocate device memory d_r with size r_size
    move data from host memory to device memory
    mmTbl.insert(r, d_r, d_r_size)
  else
  if (mmTbl.size(r) > r_size)
     mmTbl.release(r)        // free r mapped device memory
```

For the second phase, when a host memory reference is adopted as an input parameter of a GPU kernel, the implementation will keep track about the base address stored in *mmTable*. Each time a new instance of a memory reference occurs in kernel region, the implementation checks out the base address and size of the corresponding reference in device memory address space from *mmTable*. The host memory reference will be replaced by the device memory reference which is checked out from *mmTable*. It is under the compiler responsibility to assign an entry in *mmTable* for each memory reference in the code.

mmTable is updated at the end of mapping process. The row assigned to memory reference the mapping operation was treating is appropriately filled: base address of data block in host main memory, base address of the corresponding block data in GPU device memory, the size of data block and the state of data block.

5.3 Asynchronous Concurrent Execution

Kernel invocation in CUDA is asynchronous, so the driver will return control to the application as soon as it has launched the kernel. At the same time, asynchronous copies are allowed if the used host memory is allocated as page-locked memory, which the page will never be swapped out of the memory. In this case, the GPU computation and data transfer between *page-locked* memory and device memory can be overlapped to improve the overall performance of the program. However, page-locked host memory is never be swapped out, it is a scarce resource and allocations as page-locked memory may start failing long before allocations in page-able memory. In addition, by reducing the amount of physical memory available to the operating system for paging, allocating too much page-locked memory will reduce overall system performance. In *memCUDA*, we also consider to adopt this property of page-locked memory to improve the performance. *memCUDA* will adaptively allocate page-locked memory instead of page-able memory even if programmer does not use it explicitly. *memCUDA* will transfer the program into CUDA code with asynchronous concurrent execution support, when meeting the following conditions:

1. There are multiple kernels in the application. Only in this case, the former kernel's data transfer can be overlapped with the next kernel execution.
2. There is no data dependence among the kernels. Currently, data flow dependence analysis is an armature technique and there are lots of tools could achieve this goal, including GCC. In *memCUDA* system, the data dependence analysis is based on related function module including the Cetus [3].
3. The amount of device memory that kernels require is less half of host memory, the reason has been mentioned in last paragraph.

6 Implementation and Performance Evaluation

We implement a prototype system to translate input *memCUDA* programs to equivalent CUDA programs under the Cetus source-to-source compilation framework [3]. This allows the use of existing CUDA compiler tool chain from NVIDIA to generate binaries. Fig. 7 shows the compilation flow. Our work focuses on *memCUDA* Extensions Handler module and the Running Libs module.

First, we extend the naive CUDA grammar rules with slight modifications to the IR and preprocessor to accept ANSI C with language extensions of CUDA and *memCUDA* pragma directives. Then ANTLR[13], which is used as an internal C language parser by Cetus, scans and parses the *memCUDA* source code to establish a *Cetus Intermediate Representations* (IR for short) syntax tree.

Second, *memCUDA* Extensions Handler module is in charge of source-to-source transformation and data flow dependence analysis based on the prior Cetus IR tree. It transforms *memCUDA* IR syntax tree to naive CUDA IR tree. *memCUDA*-specific directives will be replaced by naive CUDA and runtime libraries APIs. At the same time, some optimization manners will also be imposed in the phase such as the asynchronous current execution through data flow analysis. So the output of this phase is a modified Cetus IR tree against the original one. Then the code generator module will print out the source code through traversing the final IR tree. At last, NVCC compiler will be called to compile the source code into binary code.

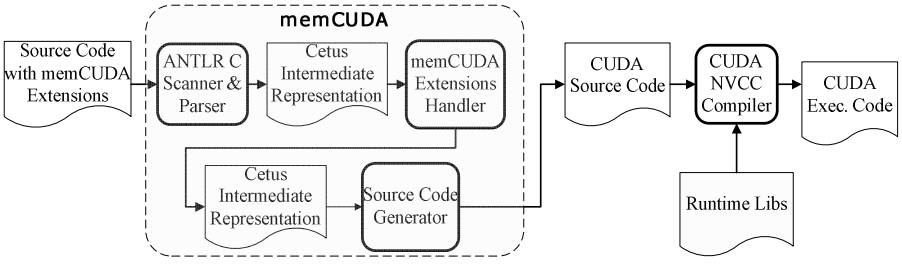


Fig. 7. The compilation workflow of *memCUDA* system

The experiments are conducted on a NVIDIA GeForce GTX 260+ GPU device. The device has 896MB of DRAM and has 27 multiprocessors (MIMD units) clocked at 576MHz. Each multiprocessor has 8 streaming processors running at twice the clock frequency of the multiprocessor and has 16KB of *shared memory* per multiprocessor. We use CUDA 2.1 for the experiments. The CUDA code is compiled with NVIDIA CUDA Compiler (NVCC) to generate device code launched from the CPU (host). The CPU is an Intel Core 2 Quad 9550 at 2.83GHz with 12MB L2 cache. The size of memory is 4GB. The GPU device is connected to CPU through a 16-x PCI Express bus. The host programs are compiled using Intel C Compiler 10.1 at -O3 optimization level.

We present six applications, listed in Table 1, as the benchmarks: MM, MRI-q, MRI-FHD, CP, PNS, and TPACF, the running example of this paper. Except MM is extracted from CUDA SDK 2.1, the other five applications[1] are selected from Parboil benchmark suite [5]. For the limitation of the space, we do not explain the detail configure parameters of each benchmark. The interested reader could refer to the web site [5]. The input dataset and parameters setting are all selected from the standard sample datasets of the Parboil benchmark suite. For the matrix multiply benchmark, we write our own *memCUDA* version and obtained CUDA version from NVIDIA CUDA SDK [7]. For the other five benchmarks, we obtain the sequential version and CUDA version from the Parboil benchmark suite. In that benchmark suite, CUDA version is heavily optimized [9][10]. We rewrite the *memCUDA* versions and use *memCUDA* to compile them. We compare the *memCUDA* version's performance against the original version from Parboil suite.

In Fig. 8, we can see that there are no noticeable performance differences between original CUDA version and *memCUDA* version. Generally, extra overhead is less than 5% comparing with the original version. This means *memCUDA* does not introduce much performance penalties. The performance loss arises from the extra operations on the mapping between host memory and devices memory. However, the tiny overhead does not beyond the boundary that we could afford.

[1] Currently, *memCUDA* only realizes the *global memory* mapping, and does not support the *texture*, *constant memory* and CUDA array structure mapping. So the SAD and RPES in the benchmark suite are not used in our experiments.

Table 1. CUDA benchmarks for evaluating the *memCUDA* compiler

Kernel	Configuration
Matrix Multiplication (MM)	Vary by dimensions of matrix
Magnetic Resonance Imaging Q (MRI-Q)	Large size; Small size
Magnetic Resonance Imaging FHD (MRI-FHD)	Large size; Small size
Coulombic Potential (CP)	40000atoms, Vary by grid size
Petri Net Simulation (PNS)	2000×2000 Scale of Petri net; Trajectory 4
Two Point Angular Correlation Function (TPACF)	Default configure

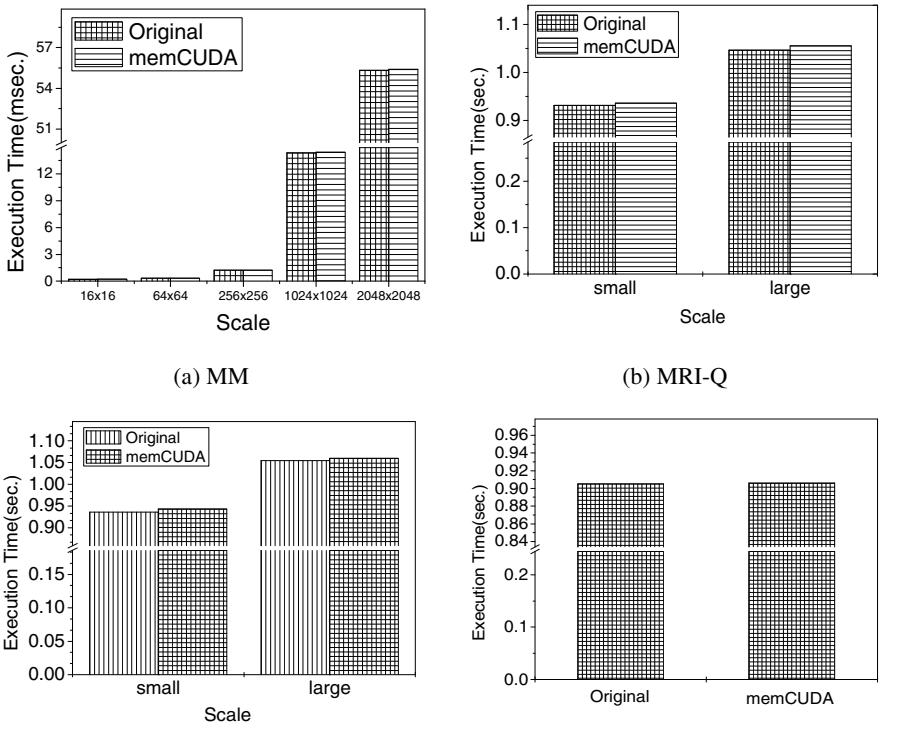
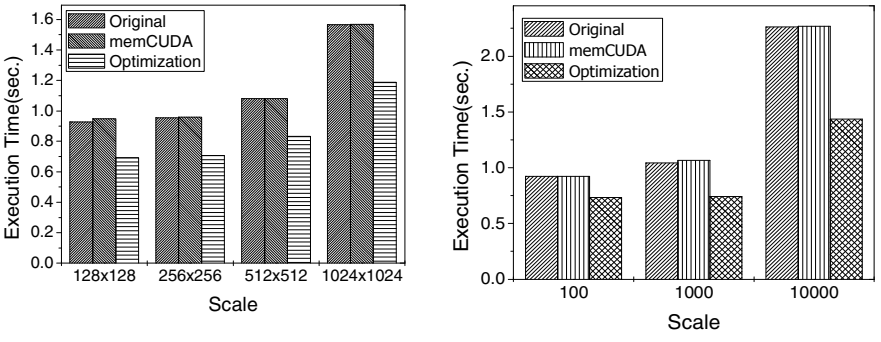


Fig. 8. Performance comparison of *memCUDA* version over CUDA version

Second, we also evaluate the efficiency of asynchronous execution optimization. Both CP and PNS repeatedly invoke kernel functions in a for-loop structure, so we conduct the experiment on CP and PNS. The others have only one kernel to execute, so the asynchronous execution optimization is not valid to them. The experimental results are shown in Fig. 9. The original version is naive CUDA version from Parboil benchmark suite. *memCUDA* label presents the *memCUDA* version without asynchronous execution

optimizations. Optimized version is the one that automatically performs the asynchronous execution optimizations by *memCUDA*. In Fig. 9, we can see the optimized version is superior the prior two versions obviously. In most cases, the performance boosts about 30% comparing to the other two versions.



(a) CP (b) PNS

Fig. 9. Performance comparison of original CUDA version, *memCUDA* version and optimized *memCUDA* version

7 Conclusions and Future Work

In this paper, we present a study to reduce programmers' burden of data movement between host memory and device memory under GPU CUDA programming environment. *memCUDA* high-level language is implemented with the composition of source-to-source compiling and runtime libraries technologies. *memCUDA* produces code with performance comparable to hand-optimized version programs. The coding of *memCUDA* is lower than the same transformations by hand and a layer of abstraction is provided from the definition of warps in CUDA. Since *memCUDA* does not handle parallelizing aspects of GPU programming, as the memory optimizing module of an eventual overall framework, *memCUDA* will facilitate GPGPU programming to encompass parallelization and resource usage decisions to maximize performance.

The latest CUDA 2.3 supports a *zero-copy* mechanism, which could avoid allocating a block in device memory and copy data between this block and the block in host memory; data transfers are implicitly performed as needed by the kernel. It seems that main idea of *zero-copy* is same as ours. However, there is a tough constrain for *zero-copy* mechanism. Host block used for mapping the device memory must be the *page-locked* memory (which will never be swapped out by OS). In fact, the GPU driver always uses DMA (*Direct Memory Access*) from its internally pinned memory buffer when copying data from the host memory to the GPU global memory. The up limited size of available *page-locked* memory is the half of main memory. So lots of applications could not use the *zero-copy* mechanisms to get a performance improve. Unlike the hardware mapping supported by current CUDA devices and software, *memCUDA* uses

implicit copy operations inserted by its source-to-source compiler to maintain consistency between the memory spaces. This avoids the problem of needing to pin system pages.

Our currently ongoing and future work are followings: *a*) extend *memCUDA* to leverage constant and texture memory mapping; *b*) use some *classic* compiler optimization to automatically optimize transformation performance, such as adaptive loop unrolling to achieve more efficiency asynchronous execution when a kernel invoked in for-loop structure; *c*) simplify the directives in *memCUDA*, some of which can be replaced by compiler analyses.

References

1. NVIDIA. NVIDIA CUDA, http://www.NVIDIA.com/cuda
2. McCool, M.D., Wadleigh, K., Henderson, B., Lin, H.-Y.: Performance Evaluation of GPUs Using the RapidMind Development Platform. In: Proceedings of the ACM/IEEE Conference on Supercomputing (2006)
3. Lee, S.-I., Johnson, T., Eigenmann, R.: Cetus - an extensible compiler infrastructure for source-to-source transformation. In: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (2003)
4. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-M.W.: CUDA-lite: Reducing GPU programming complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
5. IMPACT Research Group. The Parboil benchmark suite (2007), http://www.crhc.uiuc.edu/IMPACT/parboil.php
6. Hou, Q., Zhou, K., Guo, B.: BSGP: bulk-synchronous GPU programming. ACM Transaction on Graphics 27(3) (2008)
7. Han, T.D., Abdelrahman, T.S.: hiCUDA: a high-level directive-based language for GPU programming. In: Proceedings of the Second Workshop on General Purpose Processing on Graphics Processing Units (2009)
8. NVIDIA, http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf
9. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-M.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 15th ACM SIGPLAN Principles and Practice of Parallel Computing (2008)
10. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.-Z., Stratton, J.A., Hwu, W.-M.W.: Program optimization space pruning for a multithreaded GPU. In: Proceedings of the International Symposium on Code Generation and Optimization (2008)
11. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Transaction on Graphics 23(3), 777–786 (2004)
12. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
13. ANTLR, http://www.antlr.org/

14. Liao, S.-W., Du, Z., Wu, G., Lueh, G.-Y.: Data and computation transformations for Brook streaming applications on multiprocessors. In: Proceedings of the 4th International Symposium on Code Generation and Optimization (2006)
15. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A Compiler Framework for Optimization of Affine Loop Nests for GPGPU. In: Proceedings of the 22nd Annual International Conference on Supercomputing (2008)
16. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2009)
17. Sh, A.: High-Level Metaprogramming Language for Modern GPUs (2004), `http://libsh.sourceforge.net`
18. NVIDIA, `http://www.nvidia.com/object/fermi_architecture.html`