MapCG: Writing Parallel Program Portable between CPU and GPU

Chuntao Hong
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University
Beijing China
hct05@mails.thu.edu.cn

Dehao Chen
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University
Beijing China
chendh05@mails.thu.edu.cn

Wenguang Chen
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University
Beijing China
cwg@thu.edu.cn

Weimin Zheng
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University
Beijing China
zwm-dcs@thu.edu.cn

Haibo Lin
China Research Lab of IBM
Beijing, China
linhb@cn.ibm.com

ABSTRACT

Graphics Processing Units (GPU) have been playing an important role in the general purpose computing market recently. The common approach to program GPU today is to write GPU specific code with low level GPU APIs such as CUDA. Although this approach can achieve very good performance, it raises serious portability issues: programmers are required to write a specific version of code for each potential target architecture. It results in high development and maintenance cost.

We believe it is desired to have a programming model which provides source code portability between CPUs and GPUs, and different GPUs: Programmers only need to write one version of code and can be compiled and executed on either CPUs or GPUs efficiently without modification.

In this paper, we propose MapCG, a MapReduce framework to provide source code level portability between CPU and GPU. Different from OpenCL, our framework is based on MapReduce, which provides a high level programming model, making programming much easier.

We describe the design of the MapReduce-based high-level programming language and the underlying runtime system to enable portability between CPU and GPU. A prototype of MapCG runtime was implemented, supporting multi-core CPU and NVIDIA GPUs. Experiments show that our implementation can execute the same source code efficiently on multi-core CPU platforms and GPUs, achieving an average of 1.6-2.5x speedup over previous implementations of MapReduce on eight commonly used applications.

PACT'10, September 11–15, 2010, Vienna, Austria.

Categories and Subject Descriptors

D.1.3 [**SOFTWARE**]: PROGRAMMING TECHNIQUES—*Concurrent Programming*

General Terms

Languages

Keywords

portability, parallel, GPU programming

1. INTRODUCTION

In the past few years, a wide variety of accelerators have emerged in the traditional general purpose computing market. Among them, graphic processing unit (GPU) has gained the most popularity because of the evolution which makes GPU capable of actual general purpose computations. Especially since the emergence of CUDA[16], a lot of traditional scientific applications have been ported to GPU, bringing huge amount of speedups.

However, there are still plenty of applications that are reluctant to be ported to these emerging systems, even though they could get performance improvements. There are two major obstacles that lead to this situation.

First, the technology is developing in extremely fast pace. As a result, a popular architecture might become obsolete within couple of years. The new emerging architecture, which could be significantly faster, may also employ very different programming interface and instruction set architecture (ISA). For example, dynamic memory allocator, which is widely used in CPU programs, is not support by current GPUs, which makes GPU programs very much different from its CPU counterparts. As a result, programmers need to write target specific code for each version of the architecture, which incurs a great amount of expenses on both development and maintenance. And as soon as more than one versions of architecture co-exist, all the versions of code need to be maintained.

Second, dealing with communication and load balance in traditional parallel computing has been difficult. The complex memory hierarchy of the accelerator based computing systems makes it even more difficult to tune the performance of applications. Besides, programmers usually need to manage another level of communication between CPU and GPU, adding more burdens to the already complex process.

Let's take GPU as an example to show why the porting could be difficult. The current mainstream approach to make use of GPU in a commodity environment is to write programs in GPU languages, such as CUDA[16] and Brooks[4]. During the run time, CPU starts a thread to port certain amount of workload to GPU and wait until it finish. During GPU execution, CPU is idle. However, to utilize the CPU computation power, programmers still need to write the CPU version of code. Once the code is updated, both versions need to be adjusted, incurring 2X of effort than normal CPU-only environment. Even worse, because GPU and CPU share the different memory address space, programmers need to maintain two copies of variables and arrays on both sides with extreme caution because this is the place where bugs are commonly discovered.

A commonly adopted solution for better programming such emerging parallel systems, as adopted by EXOCHI[22] and Merge[14], is to write architecture-specific code for each architecture, and integrate them in a unified runtime framework. This approach is applicable because the runtime framework will take care of the communication, which releases some pain from programmers. Additionally, it enables architectural specific features on each architecture which can be employed to optimize performance. However, in this approach, writing multiple versions of code for each architecture introduces much more complexity, thus this approach is still time-consuming, expensive and error-prone.

To cure the pain of developing and maintaining multiple copies of code, OpenCL[11] is proposed to ensure source level portability among different architectures. It offers a low level abstraction of several different architectures such as CPU, CELL and GPU. In OpenCL, programmers follow the low level abstraction to express the program semantic as well as locality. Using OpenCL to write portable code is, however, not a panacea. First, the level of abstraction provided by OpenCL is very low, making it difficult to program and port applications to OpenCL. Second, although programmers only need to write one copy of code, it's still mandatory to manage the communications among different memory hierarchies, and between CPU and accelerators, such as thread scheduling and synchronization between CPU and accelerators, which is usually error-prone.

In this paper, we propose MapCG, a framework which offers source code level portability between CPU and accelerators, and a runtime system that allows programmers to focus on the implementation of parallel algorithms instead of side-burdens (communication, load balance, etc.) incurred by accelerator based environment. Without losing generality, we choose nVidia GPU as the accelerator. But the methodology described in this paper can also be applied to systems consisting of other accelerators. In MapCG, programmers only need to write a program once, and it will run on both CPU and GPU cores efficiently. The key challenges we address in MapCG include:

1. Parallel programming framework: We need to express parallelism and synchronization for both CPU and GPU cores uniformly. We believe a high level abstraction is essential for good portability and ease of use, which are our top two design goals of MapCG. Thus, we build our framework based on Map-Reduce, a popular domain specific parallel programming model motivated by functional languages.
2. Portability to GPU: Because current GPUs are lack of some key features, such as dynamic memory allocator, it cannot support dynamic data structures which are critical to the performance of MapCG framework. *Mars*, the state-of-the-art MapReduce framework for GPU, requires programmers to write separate phases to count memory usage in the map phase and use sorting to group intermediate results of the map phase, which harms both development and execution efficiency. In MapCG, we show that we can use atomic instructions in GPU to implement a lightweight memory allocator which is specially optimized for MapReduce on GPU and works well for thousands of threads.

In this paper, we make the following contributions:

1. **Design and implementation of the MapCG framework**. Programmers express parallelism of the workload with the MapReduce model. *MapCG* framework automates the scheduling of MapReduce tasks on both CPUs and GPUs, and generates highly efficient code which out-performs other state-of-the-art MapReduce frameworks on either platforms.
2. **A lightweight memory allocator for MapReduce**. We reveal that the MapReduce framework does not require a full featured memory allocator. A light weight memory allocator is sufficient and efficient for MapReduce framework, especially when it is accessed by massive threads. We implement the lightweight memory allocator on both CPUs and GPUs. In addition, the memory allocator on GPUs also helps eliminate additional counting phases used in previous GPU MapReduce framework.
3. **Design and implement a Hash Table on GPUs**. The hash table is used to implement a hash based grouping mechanism to replace the original sorting based counterpart to group the intermediate data on GPUs.

Experiments are conducted on a 24-core AMD machine and a GTX280 GPU connected to a quad-core Intel CPU. Results show that our implementation can execute the same source code efficiently on multi-core CPU platforms and GPUs, achieving an average of 1.6-2.5x speedup over previous implementations of MapReduce on eight commonly used applications.

2. RELATED WORK

2.1 MapReduce Model and Implementations

The MapReduce programming model originated from functional languages, and was proposed as a parallel programming model by Google[6]. In this model, programmers express the parallelism of the program by writing Map() and Reduce() function. The runtime environment schedules the tasks to parallel threads. Because of its simplicity, the MapReduce programming model has received great popularity since

its birth. It has been widely used in data mining[8], machine learning[5] and many other fields[15][17].

There are many different implementations of MapReduce on various platforms. It is implemented on clusters inside Google. There are also open-source implementations on clusters, such as Hadoop[1].

In the meanwhile, efforts are devoted to extending the original MapReduce model to achieve better performance. Ostrich[24] proposes Tiled MapReduce, an extension to MapReduce that applies tiling to MapReduce programs, in order to optimize the use of memory, cache and CPU resources.

As multi-core CPU has become mainstream, researcher also seek to use MapReduce as programming models on multi-core CPU platforms. Phoenix[19] is an implementation of the MapReduce framework for multi-core CPU platforms. Phoenix-2[23] is the latest version of Phoenix, which provides improved scalability over Phoenix. For simplicity, we use Phoenix in place of Phoenix-2 in later text.

Although Phoenix provides an efficient MapReduce implementation on multi-core CPUs, it has several drawbacks.

One major issue is that programmers are responsible for managing the memory in Phoenix. In Phoenix, the keys and values are passed by reference. Although this might improve the performance by eliminating unnecessary memory copies, it significantly harms the programmability because programmers need to manage the memory manually. Moreover, it's impossible to support platforms that have different address spaces, such as systems comprised of CPU and GPU.

There are also attempts to implement MapReduce on accelerators, such as GPU[12], FPGA[20] and CELL[18]. Mars[12] is an efficient MapReduce framework based on NVIDIA CUDA[16].

Mars has several differences from the original MapReduce model, due to the limitations of GPU.

First, Mars has two counting phases, namely MapCount and ReduceCount. During the Map and Reduce phases, the emitted data must be stored in GPU memory. However, GPU code is currently unable to allocate memory dynamically. The video memory space is allocatable only from CPU. The MapCount and ReduceCount phases are designed to work around this problem. MapCount is a stage performed before the Map phase. It counts the size of intermediate data each thread will generate, so that after the MapCount stage, GPU memory can be allocated for each thread. The MapCount phase basically executes the same code as the Map phase. But when the intermediate data is emitted, the framework keeps only the size of the intermediate data. The actual data is dropped. In this way, the amount of intermediate data each thread will generate in the Map phase can be known. After the MapCount phase, the framework allocates video memory for each GPU thread, where the emitted data is stored in the Map phase. The same process applies to the ReduceCount phase. This design is tricky, but inefficient. Mars proposes using different code in the MapCount and ReduceCount phases, so as to reduce the overhead caused by MapCount and ReduceCount functions. Thus it is necessary to write two additional functions, which is error-prone.

Another problem with Mars is that, instead of hashing, it uses sorting to group the intermediate pairs. This is generally because it is hard to implement hash table on GPU. In the hash table, intermediate pairs that are hashed to the same bucket should be organized using data structures such as a list. But since memory allocation is not available in GPU, it is impossible to allocate the list nodes dynamically. Moreover, even if the list nodes can be dynamically allocated, ensuring the correctness of concurrent insertion is not straightforward. As a result, Mars uses bitonic sort[10] to sort the intermediate pairs after the Map phase, which is less efficient than hashing.

2.2 Programming Models for GPU and Other Accelerators

Graphics Processing Unit (GPU) is playing more and more important role in computing. They have received such popularity mainly because they have much higher peak performance than CPU, usually by an order of magnitude. However, the differences in threading model and instruction set architecture (ISA) between CPU and GPU make it hard to execute current multi-thread CPU code directly on GPU.

As a result, new programming models have been proposed to program GPU, such as CUDA[16], CTM[2] and Brook[4]. These models are GPU-specific. Applications written in these models cannot be executed on CPU or other platforms except GPU.

To minimize the programming effort, other programming models are proposed to bridge the gap between CPU and GPU.

OpenCL[11] tries to provide unified view of ISA between CPU, GPU and other accelerators. It enables the programmer to write the same kernel code to execute on different processors including CPU and GPU. However, programmers are still responsible for handling the communication between the processors.

EXOCHI[22] and Merge[14], on the other hand, try to provide a uniform framework to take advantage of different types of processors by hiding the communication. They require the programmer to write different versions of the same function for different architectures.

There are also other efforts that enable the automatic translation between multi-threaded CPU code and CUDA code. MCUDA[21] translates CUDA code into multi-thread CPU code automatically, and [13] proposes a framework to translate OpenMP code into CUDA code. These two works saves the porting effort between CPU and GPU. But multi-thread programs are still needed to take advantage of these frameworks.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of the MapCG framework. We first describe the overall design of the framework, specification of the high-level programming language, and then present the implementation of MapCG runtime on both CPU and GPU.

3.1 Overall Design

Figure 1 shows an overview of the MapCG framework. The MapReduce framework is composed of two major parts: the MapReduce-based high-level programming language, and the MapCG runtime for specific architectures. The high-level programming language provides the programmers with a unified, high-level parallel programming environment. The MapCG runtime bridges the gaps of different hardware, and executes MapReduce programs efficiently on different platforms.

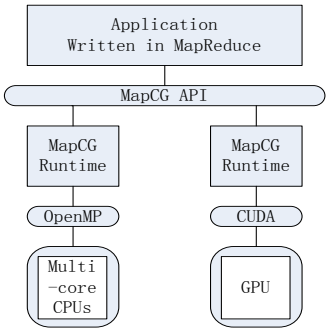

Figure 1: Overview of the MapCG framework.

Programmers developing applications deal with the MapCG programming API only. The programming API provides a MapReduce parallel model, and a unified view of instruction set architecture (ISA). Programmers write the Map and Reduce functions in C-like language, and express the parallelism of the application in MapReduce model. In our current implementation, we use the language specifications in CUDA[16], as the language for writing Map and Reduce functions. Code in Map and Reduce functions are restricted as in CUDA. However, to keep the code portable, we don't support the extensions in CUDA, such as __global__ and __device__.

The framework generates the CPU and GPU versions of the Map and Reduce functions by source code translation, and then uses the MapCG runtime library to execute them on CPU and GPU respectively.

The MapCG runtime is responsible for executing MapReduce code efficiently on different architectures. It also fills the gaps between architecture capabilities. For example, it provides string library on GPU, which enables the programmer to call string library in the Map and Reduce functions.

3.2 MapReduce Based API

The API we provide for the MapReduce programming model is shown in Table 1.

In the execution of a MapReduce application, input data is split by the Splitter() function into pieces, and the pieces are passed to the Map function. The Map function process the data and emits intermediate pairs using MapCG_emit_intermediate(). The intermediate pairs are then grouped and passed to the Reduce function, which emits data using the MapCG_emit() function. The data emitted by Reduce can then be obtained by invoking the MapCG_get_output() function.

Unlike Phoenix, we follow strict pass-by-value semantics for the MapCG_emit_intermdiate() and MapCG_emit() functions. The key/value pairs are always copied by definition, although the implementation may optimize away some copying operations as long as it does not break this semantic. This pass-by-value strategy has several benefits.

First, it relieves the burden of memory management from programmers. If the pairs are passed by reference, the programmer should make sure that the memory content does not get destroyed before they are used. They should also remember to manually free the memory when the pairs are no longer needed.

Second, passing the pairs by value makes sure that the pairs emitted are accessible from both CPU and GPU, which

Functions defined by programmer:
*void Splitter(void * in, unsigned in_size, unsigned in_idx, void *& out, unsigned & out_size)* splits a piece of the input data
*void Map(void * in, unsigned in_size)* map function, process one piece of data
void Reduce(Key_t key, Val_list_t vlist) reduce function, reduction on values associated with the same key
*unsigned Hash(void * key, unsigned keysize)* hash function
*bool Key_Equal(void * key1, unsigned size1, void * key2, unsigned size2)* compares two keys, return true if they are equal
Functions defined by the framework:
void MapCG_init(const MapCG_Spec_t & spec) initialize the framework, specify input, number threads, etc.
void MapCG_map_reduce() start the MapReduce process
MapCG_output_t MapCG_get_output() get the output data
*void MapCG_emit_intermediate(void * key, unsigned keysize, void * val, unsigned valsize)* emit an intermediate pair
*void MapCG_emit(void * val, unsigned valsize)* emit a final result

Table 1: MapReduce-based API provided by MapCG.

makes it possible to distribute tasks simultaneous on both CPU and GPU. Currently, we have implemented the MapCG framework so that we can use multiple CPUs and GPUs to execute the MapReduce tasks at the same time. However, due to separate memory address and the low bandwidth between CPU and GPU, using CPU and GPU at the same time does not offer much performance improvement, sometimes it even degrades performance. We will discuss this problem in more detail in section 5.

Passing pairs by value do have a drawback, though. It requires the copying of intermediate data, which may incur some overhead. But we consider this overhead justifiable. In most cases, the keys and values are just simple types such as integer. Copying the value of these types around is no more expensive than copying their pointers. Even if the keys and values are large in size, we copy them at most twice – first for inserting into the hash table, and then dump them to the output. The keys and values are stored in linked list when they are inserted into hash table, thus no data movement occurs when new keys and values are inserted.

Another difference between Phoenix and MapCG is that, MapCG never sorts the key/value pairs. For one thing, the ordering of the pairs may not be desired. For example, in the WordCount program, as described in section 4.1, we may want to sort the result by the frequencies each word appears, instead of by the word. For another, we can design the hash table so that the key/value pairs are ordered as we want. This technique can be used in applications such as MatrixMultiplication. Nevertheless, for other applications that do require the ordering of the results, we provide an optional post-processing sorting phase.

This design has two benefits. First, for complex keys, it is usually much easier to determine if they are equal than to determine their order. For example, we can assert that two strings are not equal if they don't have the same size, but we can only determine they order by comparing each character

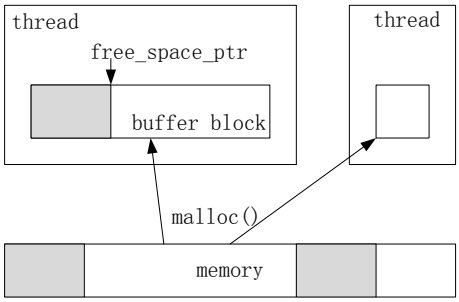

Figure 2: Demonstration of the specialized memory allocator on CPU.

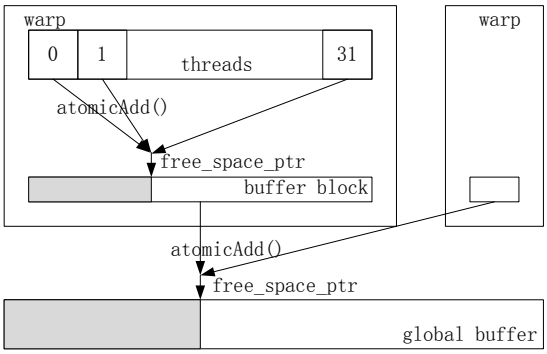

Figure 3: Demonstration of the specialized memory allocator on GPU.

in the strings. Moreover, if the results do need to be sorted, sorting them after the Reduce phase is preferable to sorting after the Map phase, because the amount of intermediate pairs are usually much larger than the pairs after reduction.

3.3 Lightweight Memory Allocator for MapReduce Framework

One of the major bottlenecks of the Phoenix MapReduce framework is the allocation of memory, as observed by [23]. This bottleneck is caused by the massive requests for malloc() when the keys and values are generated. Our framework also has such problem when the keys and values are copied into the hash table.

One way to improve the performance of memory allocation is to replace the default memory allocator by some high-performance parallel allocators such as Hoard[3] and LFMalloc[7]. However, in the context of MapReduce framework, we can achieve even better performance by using lightweight specialized memory allocator.

In the MapReduce framework, memory blocks are allocated to store the data emitted in the Map and Reduce functions. These memory blocks are massive, small blocks. Their life cycle lasts from the creation to the end of the MapReduce execution. They can be safely freed as the end of the MapReduce execution. Hence we can design the memory allocator so that it is optimized for large amount of small block allocations, and allows free() only at some given points. Such design greatly simplifies the memory allocator.

3.3.1 Memory Allocator on CPU

Figure 2 demonstrates the structure of our memory allocator on CPU.

As designed in many multi-thread memory allocators, in our memory allocator, each thread holds a local "buffer block", a large block of memory allocated directly using the malloc() call. When a memory allocation request is issued, the requesting thread first tries to allocate the block from its own buffer block. If the current buffer block does not have enough free space to satisfy the request, then the thread malloc() a new buffer block directly from the system, and add it to the buffer block list. We double the size of the buffer block each time a new buffer block is allocated, so as to further reduce the number of request to malloc(). Because the memory allocated is freed together, we don't keep track of the individual allocated blocks. Instead, we record only the buffer blocks. At the end of the MapReduce execution, we can simply free these buffer blocks, which is much faster than freeing the small blocks one by one.

In most of the cases, an allocation with our memory allocator involves only an addition to the "free space pointer" of the local buffer block. This provides extremely fast memory allocation, especially for the MapReduce applications, in which large amounts of small blocks are allocated.

3.3.2 Memory Allocator on GPU

Currently, dynamic memory allocation in GPU code is not supported by CUDA, which prevents the MapReduce framework from allocating the memory space for intermediate data dynamically. Mars uses two counting phases to work around this problem, as stated in section 2.1. We solve this problem by implementing a specialized memory allocator on GPU.

Figure 3 shows the design of the memory allocator for GPU. Similar to the memory allocator for CPU, threads in the GPU also hold local buffer blocks. However, there are also differences.

First, GPU threads cannot use malloc() to allocate buffer blocks, because there is no such functionality on GPU. In our design, we allocate a large block of video memory, called "global buffer", before the GPU threads are created. The global buffer serves as a global memory pool for all the GPU threads. The maximum size of the global buffer can be determined because memory usage pattern are fixed in MapReduce applications. When GPU threads need a new buffer block, it simply increases the "free space pointer" of the global buffer. The increase of the offset is implemented using the atomicAdd() operation provided by GPU, to ensure the correctness of concurrent operations.

Aside from the difference in fetching buffer blocks, the memory allocator on GPU has another major difference from that on CPU. In order to fully leverage the power of GPUs, GPU programs typically spawns tens of thousands of threads simultaneously. If each one of the threads allocates a buffer block of several kilo-bytes, the total amount would be hundreds of mega-bytes, which is a significant amount, considering that mainstream GPUs have only 1 4GB video memory. Also, if the GPU threads use only a fraction of the buffer block, there will be a lot of inner fragments, wasting a lot of space. In our design, we keep a buffer block for each warp. A warp in CUDA stands for a bunch of threads, usually 32, that are always scheduled together and execute the same in-

struction at the same time. Because the 32 threads use the same buffer block, we use atomic operations to fetch a new block from the buffer block.

Keeping a buffer block for each warp has another potential benefit: it improves memory access bandwidth by causing more coalesced memory access. Memory accesses from the threads in the same half-warp (the first or second part of a warp) are coalesced into 64-byte, 128-byte and 256-byte memory transactions that cover the memory addresses. When the memory addresses accessed by the threads are close together, the access will result in fewer transactions, thus improving memory bandwidth efficiency. Our memory allocator tend to provide neighboring memory blocks to the threads in the same warp, giving more opportunity for memory coalescing.

To further reduce the cost of memory allocation, we make use of the shared memory provided in GPU. Shared memory is a fast on-chip storage shared in a thread block. Operations to the shared memory have much lower latency than those to the global memory. By keeping the information of buffer blocks in shared memory, we can greatly reduce the cost of memory allocation.

3.4 Hash Table on GPU

To group the key/value pairs on GPU, we need to implement a hash table. However, implementing a hash table on GPU is not trivial. There are two major challenges. First, the data nodes must be dynamically allocated, which is impossible without a memory allocator. Second, the hash table should provide efficient concurrent insertion.

Fortunately, we already have a memory allocator, which solve the first problem. We implement a closed addressing hash table, in which data in the same hash bucket are kept in a list. The dynamic allocation of the list nodes are carried out using the memory allocator presented in section 3.3.2.

The efficiency and correctness of concurrent insertion is ensured by using a lock-free algorithm. The lock-free algorithm ensures that the insertion never gets blocked by any specific thread. Especially, it precludes the possibility of deadlocks among threads in the same warp.

Lock-free linked-list implementations are hard to get right. There are many articles discussing this[9]. However, we can simplify the list to allow only concurrent insertion. The deletion can be done automatically when the memory allocator is destroyed.

Figure 4 demonstrates the insertion operation.

The CASPTR (Compare And Swap PoinTeR) function takes three arguments, the address of the original pointer, the expected value of the pointer, and the new value of the pointer. If the value stored in the address equals to the expected value, then it replaces the content by the new value. The compare-and-swap operation is executed atomically. The CASPTR is implemented in atomicCAS() function provided by GPU.

In the above code, we should have freed the memory space allocated for SMA_Free in line 14. But since the memory allocator does not support dynamic freeing of memory blocks, the SMA_Free function is actually empty. This could result in memory leaks, but since memory is managed by our memory allocator, we can be sure that this piece of memory will be released at the end of the MapReduce execution. Similarly, all the allocated list nodes can be freed when the memory allocator is destroyed.

```
1 void InsertPair(list, key, value){
2    key_list_node_t * curr=list->head;
3    key_list_node_t * new_node=NULL;
4    if(curr==NULL){
5       new_node=NewKeyListNode(key);
6       if(CASPTR(&head,NULL,new_node)){
7          InsertValue(new_node->value_list, value);
8          return;
9       }
10   }
11   curr=head;
12   while(1){
13      if( KeyEqual(curr->key, key) ){
14         SMA_Free(new_node);
15         InsertValue(curr->value_list, value);
16         return;
17      }
17      if( curr->next==NULL ){
19         if(new_node==NULL)
20            new_node=newKeyListNode(key,keysize);
21         if(CASPTR(&(curr->next),NULL,new_node)){
22            InsertValue(new_node->value_list, value);
23            return;
24         }
25      }
26      curr=curr->next;
27   }
28}
```

Figure 4: Code snippet of the lock-free algorithm in hash table implementation.

4. EXPERIMENT RESULTS

We evaluated MapCG framework on both CPU and GPU platforms. In this section, we first briefly introduce the experiment platform and the benchmarks we use. Then we show how effective MapCG performs by comparing it with previous implementations and hand-tuned code.

4.1 Experiment Setup

Our experiments are conducted on two different platforms:

1. A 24-core AMD machine, which is composed of 4 six-core Opteron CPUs, equipped with 32GB main memory. Each CPU has 6 cores working at 2.4GHz, and 6MB shared L3 cache. Each core has 128KB L1 cache and 512KB L2 cache. The 64 bit Linux is used as the operating system.

2. A GTX280 GPU, which is equipped with 30 multiprocessors and 1GB of video memory. Each multiprocessor has 8 processors working at 1.3GHz. In other words, GTX280 has 240 streaming processors onboard. It is connected to a 2.4GHz quad-core Intel CPU through PCI-Express bus.

We choose eight applications from different fields to test the performance of our implementation. These applications cover the fields of enterprize computing (WordCount and StringMatch), web log analysis (PageViewCount) and web document searching and clustering (InvertedIndex and K-means), web document processing (SimilarityScore) and scientific computing (MatrixMultiplication and N-Body simulation). Table 2 shows the detailed information on these applications, including the dataset size used and number of lines of code used to implement them on MapCG, Phoenix and Mars.

Application	Data Size	Code Size		
		MapCG	Phoenix	Mars
StringMatch (SM) Search file for given string	S: 32MB, M: 128MB, L: 256MB	194	189	285
WordCount (WC) Determine frequency of words in a file	S: 10MB, M: 50MB, L: 100MB	204	208	345
PageViewCount (PVC) Determine frequency of pages viewed in a web log	S: 32MB, M: 64MB, L: 128MB	239	258	340
InvertedIndex (II) Build reversed index from HTML files	S: 32MB, M: 64MB, L: 128MB	259	271	527
SimilarityScore (SS) Compute the cosine similarity between given dense vectors	S: 1024, M: 2048, L: 4096	188	212	265
MatrixMultiplication (MM) Dense integer matrix multiplication	S: 512, M: 1024, L: 2048	198	192	254
Kmeans (Kmeans) Iterative clustering algorithm to classify data points into groups	S: 10K points, M: 50K, L: 100K	299	274	421
N-body (Nbody) All-pair N-body simulation	S: 4096, M: 16384, L: 65536	283	265	324

Table 2: Applications used in our experiments.

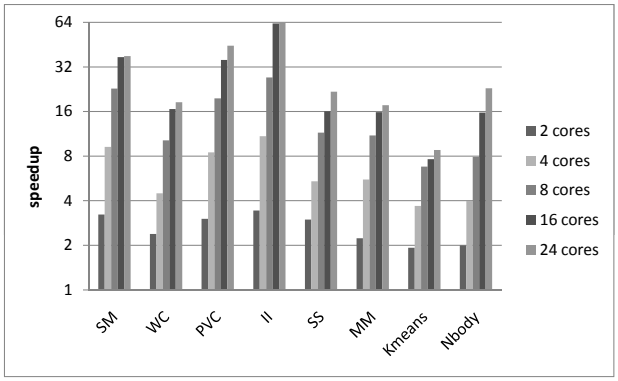

Figure 5: Speedup with MapCG for the large datasets as we scale the number of processors cores on the multi-core AMD machine.

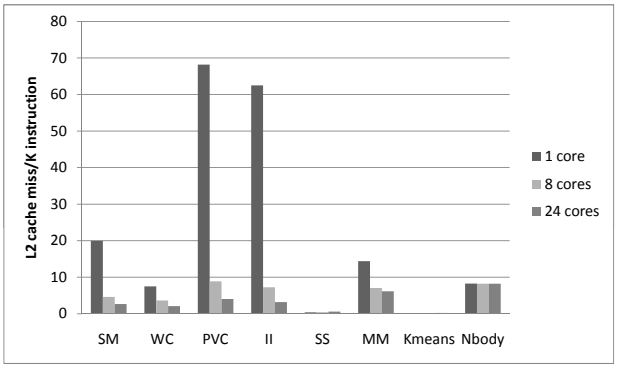

Figure 6: Number of L2 cache misses per 1000 instructions with different number of processor cores.

4.2 Performance

4.2.1 *Performance on Multi-core CPU Platform*

Figure 5 shows the speedup achieved as we scale the number of processor cores on the AMD machine. We can see that MapCG has achieved very good scalability. 7 out of 8 applications can achieve more than 16X speedup in a 24-core environment. Among them, super-linear speedup is achieved in 4 applications. A further study of L2 cache miss rate, as shown in Figure 6, indicates that the super-linear speedup is mainly due to the increased cache size as the number of processor core increases. Also note that in in Nbody, the L2 cache misses rate remains the same as the number of cores increase. This is because the working-set of the Nbody is approximately 4MB, which can never fit into L2 cache regardless of the number of cores used. However, the working-set can always fit in L3 cache. As a result, this application has a perfect linear scale curve.

Figure 7 shows the performance comparison between MapCG and Phoenix-2. In this test, all 24 cores are enabled to deliver full throughput. And the speedup against Phoenix is shown for MapCG across different datasets. We can see that MapCG is constantly faster than Phoenix-2. On average, MapCG achieves 2-3x speedup over Phoenix-2. The performance gain of MapCG is mainly caused by different strategies in managing the key/value pairs. Phoenix-2 al-

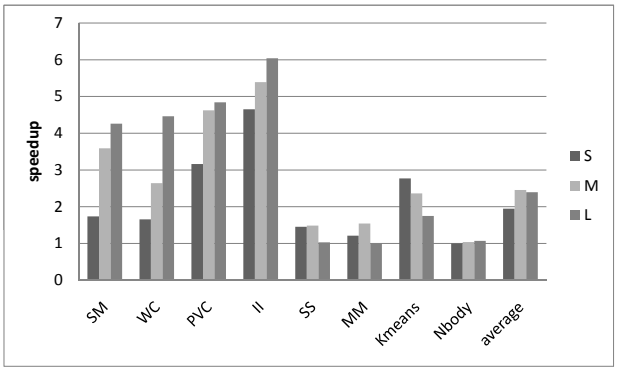

Figure 7: Speedup of MapCG over Phoenix-2 on the AMD machine, using Small, Medium and Large datasets.

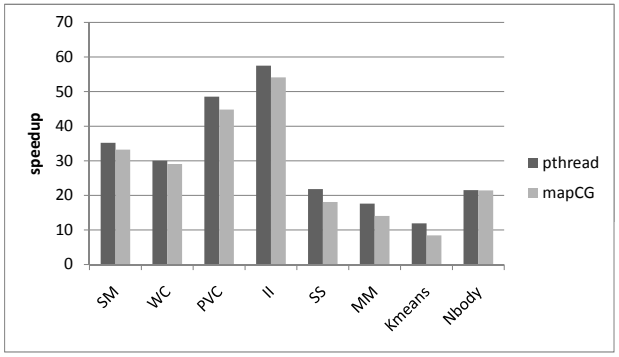

Figure 8: Speedup of pthread version and MapCG version of the benchmarks on 24-core AMD machine.

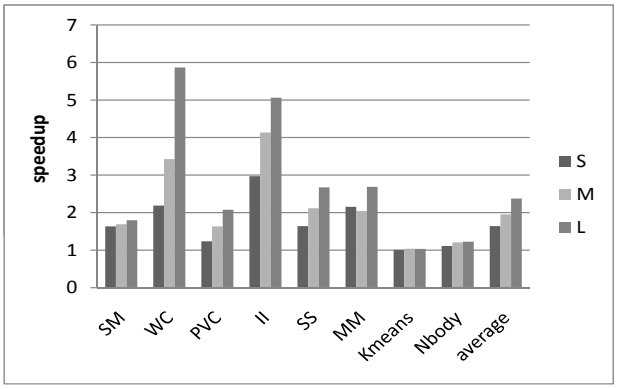

Figure 9: Speedup of MapCG over Mars on GTX280 GPU, using Small, Medium, and Large datasets.

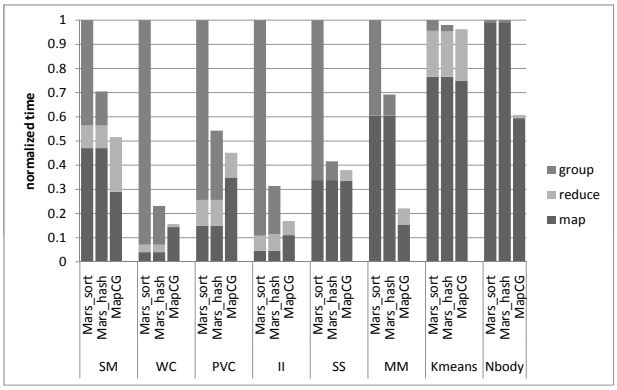

Figure 10: Time breakup of the three phases in Mars_sort, Mars_hash and MapCG. Results collected using the Large datasets.

ways sorts the key/value pairs, while MapCG never sorts them. Another contributor is the use of customized memory allocator, which greatly reduces the number of system calls when many key/value pairs are emitted. Hence the speedup varies with different application and input size.

For StringMatch, WordCount, PageViewCount, and InvertedIndex, MapCG can achieve high speedup over Phoenix-2. This is because these five applications emit large amounts of key/value pairs. Phoenix spends much time on sorting them.

The other four applications are compute-intensive. They spend most of the execution time in arithmetic operations. The amount of key/value pairs is small, considering the computation complexity of the programs. As a result, MapCG show almost identical performance on these applications, compared with Phoenix.

For the first four applications, we observe higher speedup with larger input datasets. As the characteristics of the applications are similar with different input sizes, we will use only the largest dataset to analysis the performance of MapCG.

To analyze the efficiency of the MapCG model against hand-coded multi-thread programs, we also implement the p-thread version of the eight applications. The p-thread code of WordCount, StringMatch, InvertedIndex, Matrix-Multiplication and Kmeans come from the Phoenix. We wrote the other three applications. For all of the p-thread code, we tune them to be as fast as possible. Figure 8 shows the speedup of the p-thread version and the MapCG version over sequential code. We can see that the MapCG code shows some slowdown compared with p-thread. But the difference is tolerable. This difference is mainly caused by the additional copying of the key/value pairs. While MapCG always copies the key-value pairs into hash table and then extracts them out, the p-thread version can directly assign them to pre-allocated memory space.

4.2.2 Performance on GPU

In this section, we present the performance of MapCG on GPU. We compare the performance with Mars, a previous implementation of MapReduce on GPU.

Figure 9 demonstrates the speedup of MapCG against Mars. As shown in the figure, MapCG is constantly better than Mars. On average, it achieves 2.4x speedup over Mars on large datasets, and 1.6x speedup on small datasets.

The speedup mainly comes from two optimizations: replacing sorting by using hash functions and removing counting phases.

Mars uses sorting to group the intermediate key/value pairs after Map phase. On the contrary, we use hashing to group the pairs, which is more efficient. To demonstrate the benefit of replacing sorting with hashing, we modify the Mars framework, so that the Group phase of Mars uses hashing. We name this version of Mars as Mars_hash, and the original Mars as Mars_sort. Figure 10 presents the time break down of different phases in Mars_sort, Mars_hash and MapCG when running with large datasets.

As we can see, the group phase is very important for performance. For WordCount, PageViewCount and InvertedIndex, the most important part of the program is to identify the right entry corresponding to a key. In Mars, the Group phase is responsible for searching for the right entry for the keys. Thus, in these three applications, the Group phase takes up most of the time. For SimilarityScore and Matrix-Multiplication, most of the time should have been spent on arithmetic operations, (i.e. the Map phase), but we can see that the Group phase also takes significant time. This is one of the drawbacks of the MapReduce programming model. Though using hashing to group the intermediate data re-

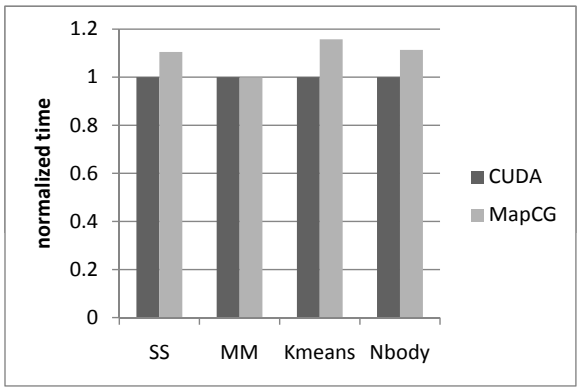

Figure 11: Normalized execution time of MapCG over CUDA on four applications.

duces most of the grouping time, it still takes considerable amount of time.

We can see that Mars_hash spends much less time in the Group phase than Mars_sort. MapCG don't need the Group phase at all. The pairs are grouped when they are inserted into the hash table in the MapCG_emit_intermediate() function called by Map function.

In addition to using hashing instead of sorting, the MapCG framework gains advantage over Mars by using the memory allocator. The memory allocator makes the counting phases unnecessary. Moreover, the memory allocator also improves memory access pattern by increasing coalesced access. The memory allocator tends to allocate memory blocks close to each other for threads in the same warp. So there is more opportunity for coalescing the memory access when these memory blocks are accessed, improving memory bandwidth efficiency. These two optimizations are supposed to reduce the time in Map phase. However, the key/value pairs are grouped together when they are emitted in MapCG. This increases the time in the Map phase. Overall, MapCG has longer Map phase in WordCount, PageViewCount and InvertedIndex, compared with Mars.

MapCG also has longer Reduce in StringMatch, SimilarityScore and MatrixMultiplication, because it has to copy the emitted results from the hash table into an array.

Figure 11 shows the normalized execution time of MapCG over CUDA on four of the applications. The other four applications, StringMatch, WordCount, PageViewCount and InvertedIndex are extremely awkward to write in CUDA. Programmer would be required to write the hash table from scratch. As a result, we show only the performance on SimilarityScore, MatrixMultiplication, Nbody and Kmeans. The CUDA code has been tuned to be as fast as possible. However, we do not make use of shared memory in the CUDA code. Our implementation of MapCG is not capable of using shared memory now. Future work may improve this by employing compiler optimization.

We can see in the figure that the MapCG version of these four applications achieves performance close to hand-coded CUDA code, though there are still inherent overheads introduced by the MapReduce framework.

5. LIMITATIONS, DISCUSSIONS AND FUTURE WORK

5.1 Employing the memory hierarchy in GPUs with MapCG

MapCG uses a subset of C language to describe the single thread behavior of *Map* or *Reduce*. While this approach favors portability between CPUs and GPUs, it loses a few low level hardware features which are useful for performance optimization. Especially,our current MapCG implementation can not take advantage of the *shared memory* and *constant memory* in GPUs[16]. This problem can be addressed partially with the compiler support which recognize shared global variables or read only variables and promote them to corresponding memory hierarchies automatically which is our future work.

5.2 Running MapCG applications on both CPUs and GPUs at the same time

We provide three runtime modes in MapCG implementation: CPU+GPU co-processing, CPU-only and GPU-only computation. The CPU+GPU co-processing mode enables user to run one MapCG application on both CPU cores and GPUs at the same time. However, we observe that it does not yield significant better performance in this mode. For the 8 benchmarks we tested, the speedup of using CPU+GPU over the faster of CPU-only and GPU-only is always below 1.1. 2 of them even slows down when executed in CPU+GPU mode.

The results can be explained in two aspects: Firstly, CPUs and GPUs are good at different applications with big margin. For example, on the GPU machine, GPU is 86 times faster than CPU in MatrixMultiplication. In such cases, tt does not make much benefit to let the slower architecture join the computation. Secondly, using CPU-GPU co-processing introduces overhead. One of the major overhead comes from serialization and de-serialization, which are required to transfer non-array data between CPUs and GPUs.

5.3 CUDA 3.0 and MapCG

Future improvement in GPU architecture will have various impact on the MapCG framework. For example, the CUDA 3.0 is claimed to provide dynamic memory allocators, C++ support, cache and unified memory space between CPUs and GPUs.

The general trend of CUDA 3.0 is to make GPUs more similar to CPUs on the programming interface, which is aligned with philosophy of MapCG. The full-featured memory allocator and C++ support in CUDA 3.0 will enable MapCG to use larger subset of C and C++ to specify single thread behavior. C++ support also allows MapCG to use function overloading to provide more elegant programming interface. Unified memory address of CPUs and GPUs can be used to reduce the overhead of transferring data between them. Introduction of the L1 and L2 cache can also improve the performance of MapCG programs without the need to employ memory hierarchy specific optimizations.

6. CONCLUSION

In this paper, we present the MapCG, a high level framework that offers source code portability between CPU and GPU. We propose a few techniques to fill the gap between

GPU hardware features and requirement of MapCG , such as memory allocator and dynamic data structures.

Experiments show that our implementation offers higher performance than Phoenix and Mars, the state-of-the-art MapReduce implementations on CPU and GPU, respectively. The comparison with hand-coded multi-thread programs shows that we can achieve comparable performance with hand-tuned code.

We view this piece of work as a instantiation of the concept "write once, run anywhere" in the accelerators context, which is the very basic rule of programming language design and implementation. We hope the work will motivate more interactions between the programming language community and hardware community.

7. REFERENCES

[1] Apache hadoop, http://hadoop.apache.org/.

[2] ATI. Ati ctm guide.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

[4] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 163–174, New York, NY, USA, 2002. ACM.

[8] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. *eScience, IEEE International Conference on*, 0:277–284, 2008.

[9] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM.

[10] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[11] Khronos Group. Opencl specification.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT*, pages 260–269, 2008.

[13] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[14] Michael D. Linderman, Jamison D. Collins, Hong Wang 0003, and Teresa H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS*, pages 287–296, 2008.

[15] Suzanne Matthews and Tiffani Williams. Mrsrf: an efficient mapreduce algorithm for analyzing large collections of evolutionary trees. *BMC Bioinformatics*, 11(Suppl 1):S15+, 2010.

[16] NVIDIA. Cuda programming guide.

[17] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow.*, 2(2):1426–1437, 2009.

[18] M. Mustafa Rafique, Benjamin Rose, Ali Raza Butt, and Dimitrios S. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *IPDPS*, pages 1–12, 2009.

[19] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

[20] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102, New York, NY, USA, 2010. ACM.

[21] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. pages 16–30, 2008.

[22] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang 0003. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI*, pages 156–166, 2007.

[23] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[24] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-MapReduce: Optimizing Resource Usage of Data-parallel Applications on Multicore with Tiling. In *Parallel Architectures and Compilation Techniques (PACT)*, 2010.