A Workload Balanced MapReduce Framework on GPU Platforms

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

by

Yue Zhang
B.S., Dalian Institute of Science and Technology, 2013

2015
Wright State University
December 15, 2015

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Yue Zhang ENTITLED A Workload Balanced MapReduce Framework on GPU Platforms BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Meilin Liu, Ph.D.
Thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination

Meilin Liu, Ph.D.

Jack Jean, Ph.D.

Travis Doom, Ph.D.

Robert E. W. Fyffe, Ph.D.
Vice President for Research and Dean of the Graduate School
ABSTRACT


The MapReduce framework is a programming model proposed by Google to process large datasets. It is an efficient framework that can be used in many areas, such as social network, scientific research, electronic business, etc. Hence, more and more MapReduce frameworks are implemented on different platforms, including Phoenix (based on multi-core CPU), MapCG (based on GPU), and StreamMR (based on GPU). However, these MapReduce frameworks have limitations, and they cannot handle the collision problem in the map phase, and the unbalanced workload problem in the reduce phase. To improve the performance of the MapReduce framework on GPGPUs, in this thesis, a workload balanced MapReduce framework (B_MapCG) on GPUs is proposed and developed based on the MapCG framework, to reduce the number of collisions while inserting key-value pairs in the map phase, and to handle the unbalanced workload problems in the reduce phase. The proposed B_MapCG framework is evaluated on the Tesla K40 GPU with four benchmarks and eight different datasets. The experimental results showed that the B_MapCG framework achieved big performance improvements for all the four test benchmarks both in the map phase and the reduce phase compared with MapCG.
## Contents

1  Chapter 1: Introduction  
   1.1  Motivation  
   1.2  Overview  

2  Chapter 2: Background  
   2.1  GPU and CUDA  
      2.1.1  CUDA C  
      2.1.2  CUDA Device Memory Model  
      2.1.3  Atomic Operations  
      2.1.4  Dynamic Parallelism  
   2.2  MapReduce  
   2.3  Hash Table  
   2.4  MapCG  

3  Chapter 3: A Workload Balance MapReduce Framework  
   3.1  Map phase  
   3.2  Reduce phase  

4  Chapter 4: Evaluation  
   4.0.1  Platforms  
   4.0.2  Benchmarks  
   4.1  Results  
      4.1.1  InvertedIndex  
      4.1.2  WordCount  
      4.1.3  PageRank  
      4.1.4  MonteCarlo  

5  Chapter 5: Conclusions and Future Work  

Bibliography  
Appendix
List of Figures

2.1 The CUDA program compiling process. ........................................... 5
2.2 A simple CUDA example. ................................................................. 7
2.3 CUDA Device Memory Model. ......................................................... 8
2.4 The mapping between CUDA and GPU Hardware. ......................... 8
2.5 Registers versus shared memory. ..................................................... 9
2.6 Kernel launch patterns. ................................................................. 12
2.7 An example of nested launch. ......................................................... 13
2.8 The execution of the MapReduce framework. ................................. 14
2.9 The shuffle and sort phase of MapReduce framework. ..................... 14
2.10 An example of handling the collision problem using linear probing. .... 16
2.11 An example of the chained hash table. ........................................... 17
2.12 An intermediate data structure of MapCG. ..................................... 18
2.13 Mapping many keys into the same bucket. ...................................... 18
2.14 Wrong key insertion versus Key insertion with atomic operations. .... 20
2.15 Inserting the keys into the key list. ............................................... 21
2.16 Inserting values associated with a particular key into the value list for this key. ................................................................. 21
2.17 The total execution time of different workload tasks. ..................... 22
2.18 The percentage of the apperances for words in 100M WordCount file. .. 22

3.1 The flowchart of the map phase of B_MapCG. ................................. 25
3.2 The definition of the key nodes and the value nodes. ....................... 26
3.3 The intermediate data structure of B_MapCG. .................................. 26
3.4 The number of collisions occurred in the WordCount implemented on the MapCG. ................................................................. 27
3.5 The collision problem caused by the atomic operations. ................ 28
3.6 The flowchart of the reduce phase on B_MapCG. ............................. 33
3.7 The checking process. ................................................................. 34
3.8 Illustration of partitioning. ........................................................... 35

4.1 An illustration example of PageRank. .............................................. 40
4.2 An illustration example of MonteCarlo. ......................................... 40
4.3 The average execution time of the four test benchmarks in the reduce phase of MapCG and B_MapCG. 

4.4 The average execution time of InvertedIndex in the map phase of B_MapCG and MapCG. 

4.5 The average execution time of InvertedIndex in the reduce phase of B_MapCG and MapCG. 

4.6 The average execution time of WordCount in the map phase of B_MapCG and MapCG. 

4.7 The average execution time of WordCount in the reduce phase of B_MapCG and MapCG. 

4.8 The average execution time of PageRank in the map phase on B_MapCG and MapCG. 

4.9 The average execution time of PageRank in the reduce phase on B_MapCG and MapCG. 

4.10 The average execution time of MonteCarlo in the map phase of B_MapCG and MapCG. 

4.11 The average execution time of MonteCarlo in the map phase of B_MapCG and MapCG.
List of Tables

2.1 The CUDA keywords for function declaration. .................................... 5
2.2 The CUDA keywords for variable declaration. .................................... 6

4.1 The average execution time of the map phase of B_MapCG and MapCG, and the speedup of B_MapCG over MapCG. ................................. 41
4.2 The average execution of the reduce phase of B_MapCG and MapCG, and the speedup of B_MapCG over MapCG. ................................. 41
4.3 The average execution time of the map phase of InvertedIndex on B_MapCG and MapCG, and the speedup of InvertedIndex on B_MapCG over MapCG. ................................. 43
4.4 The average execution time of the map phase of PageRank on B_MapCG and MapCG, and the speedup of PageRank on B_MapCG over MapCG. ................................. 46
Acknowledgment

I would like to take this opportunity to express my sincere appreciation to my master thesis advisor, Dr. Meilin Liu, for her continuous guidance, valuable advices and encouragement.

I would like to extend my sincere gratitude to my master thesis committee member, Dr. Jack Jean and Dr. Travis Doom, for their help and advices.

I would also like to thank my friend, Lifeng Liu, for his valuable suggestions and patience in helping me.

Finally, I would like to thank my family and friends for their love and support.
Chapter 1: Introduction

1.1 Motivation

Nowadays, the big data era is coming [15]. A huge amount of data is generated in different areas in one day even in one minute, such as Twitter, Facebook, Amazon, Google. To meet the powerful computing requirement in processing the big data, Google proposed a programming model called MapReduce [9], which became popular as an open-source framework called Hadoop is developed and published [1]. The MapReduce framework is used in many areas, such as social network, scientific research, electronic business, etc. Hence, more and more MapReduce frameworks are implemented on different platforms [6, 8, 10]. Phoenix [20, 23], MRJ [16], and Phoenix++ [22] have implemented a MapReduce framework on multicore CPUs.

General purpose GPU (GPGPU) is a massively parallel many-core architecture that could yield high throughput for many scientific applications with thread level parallelism [5]. MapReduce frameworks have been proposed for GPU platforms, including MapCG [14], StreamMR [11], Mars [12], and GPMR [21].

Elteir et al. [11] proposed StreamMR, an atomic operation free MapReduce framework on GPUs, which is targeting the old generations of AMD GPUs. Compared to Nvidia GPUs or the new generations of AMD GPUs, the performance of atomic operations on those old versions of GPUs is very modest. StreamMR avoids global atomic operations. However, StreamMR would allocate a sub hash table for each warp which has large space
overhead and limits the total number of warps. The records in each sub hash table are still stored in the linked lists, which means the intermediate data structure in the StreamMR can waste too much memory [11], so it is not suitable to process large datasets. In order to combine the intermediate results in those sub hash tables before the reduce phase, a complex and time consuming combining phase is introduced.

MapCG proposed by Hong et al. [14] is the state-of-the-art MapReduce framework on the GPU platforms to the best of our knowledge. MapCG could work on both CPUs and GPUs. Users of MapCG do not need to consider the hardware architectures while porting CPU MapReduce source code to GPU platforms. MapCG proposes a parallel hash table based on dynamic memory allocation to group intermediate outputs together for different keys. However, MapCG has a serious collision problem and a workload unbalance problem for some applications.

To improve the performance of the MapReduce framework on GPGPUs, in this thesis, we propose and develop a workload balanced MapReduce framework (B.MapCG) based on the MapCG framework to reduce number of collisions while inserting key-value pairs in the map phase, and to handle the unbalanced workload problems in the reduce phase. In the map phase, we introduce a segmentation table. In the reduce phase, we use dynamic parallelism to solve the workload unbalance problem.

We evaluate our B.MapCG framework on the Tesla K40 GPU with four benchmarks with eight different datasets. The experimental results showed that our B.MapCG framework achieved big performance improvements for all the four test benchmarks both in the map phase and the reduce phase compared with MapCG.

1.2 Overview

Chapter 1 describes the motivation to develop a workload balanced MapReduce framework on the GPU platforms.
Chapter 2 illustrates the basic concept used in this thesis, including GPU architecture, CUDA programming model, the MapReduce framework, the hash table, and the MapCG framework.

Chapter 3 presents our proposed B_MapCG framework. We illustrate in detail about how to deal with the serious collision problem in the map phase, and how to handle the workload unbalance problem in the reduce phase.

Chapter 4 reports the experimental results of the B_MapCG framework on the Tesla K40 GPU for four benchmarks. The timing performance of these four benchmarks on B_MapCG framework is compared with that of MapCG.

Chapter 5 concludes this thesis and also talks about the future work.
Chapter 2: Background

In this chapter, we will introduce some basic concepts used in this master thesis.

2.1 GPU and CUDA

GPU (Graphics Processing Unit) is designed by NVIDIA in 1999 to process graphics [5]. It has powerful parallel computing ability due to its massively parallel manycore architecture compared to CPU, so it has become a general purpose graphics processing unit (GPGPU) to process non-graphic applications. GPGPU cooperates with CPU to process more complicated problems including science problems, industry applications, financial applications, and some other areas.

2.1.1 CUDA C

To support parallel computing on GPGPU, NVIDIA develops CUDA (Compute Unified Device Architecture), a software and hardware architecture that allows programmers to code in high level languages without considering the graphics interfaces of GPUs [5]. CUDA extends from traditional C/C++ programming language, and it provides some new keywords and API functions to the programmers. The programmers can use NVCC (NVIDIA C Compiler) to compile the CUDA source programs. A CUDA source program has two parts of source codes. One part is the serial code that executes on the CPU, which is called
the host code. The other part is the code that runs in parallel on the GPU, which is called the device code. Thus, one CUDA source file is a mixture of the device code and the host code. While compiling the CUDA source file, the NVCC compiler will generate two different source code. They are then compiled by two different compilers (see Figure 2.1).

![CUDA Source File](image.png)

**Figure 2.1: The CUDA program compiling process.**

The NVCC compiler uses keywords added by programmers to identify the host code and the device code. Table 2.1 shows the keywords for function declaration. Table 2.2 shows the keywords for variable declaration.

**Table 2.1: The CUDA keywords for function declaration.**

<table>
<thead>
<tr>
<th>Function declaration</th>
<th>Executed environment</th>
<th>Called from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>host</strong> int hostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td><strong>global</strong> void kernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>device</strong> int deviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
</tbody>
</table>

Keyword “__host__” indicates that a function is a host function. Without this keyword, a function is also treated as a host function by default. The host functions must run on the
CPUs, and they can only be called from other host functions (see Table 2.1). The execution of a CUDA program begins with the execution of the main function on the host. In the host function, the programmers need to allocate the device memory, and copy data from the host to the device using API functions before the kernel function is launched. The kernel function is declared in the host function.

Kernel functions and device functions running on the device can be declared as follows (see Table 2.1). The device functions can only execute on GPUs.

- **global**: A function declared with `__global__` is a kernel function running on the device, and it can only be called from the host function. The return type of this function must be `void`.

- **device**: Adding `__device__` in front of a function indicates the function is a device function, and it can only run on GPUs. It cannot be called by the host functions directly. Instead it can only be called by the kernel functions and other device functions. Note that if programmers want to generate two versions of the same function (one is a host function that executes on CPUs, and the other is a device function that run on the GPUs), they can add both `__host__` and `__device__` in front of the function while declaring a function.

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__ __shared__ float sharedVar</code></td>
<td>shared memory</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><code>__device__ __constant__ float constVar</code></td>
<td>constant memory</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><code>__device__ float globalVar</code></td>
<td>global memory</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td>float localVar</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
</tbody>
</table>

The keyword `__shared__` is used to declare a shared variable that can be shared by all the threads in the same thread block. The keyword `__constant__` is used to declare a constant variable, which can only be read by the CUDA threads running on the GPUs. Constant variables cannot be modified by any CUDA threads running on the GPUs.
constant variable is stored at the constant memory, so they can be shared by all the threads in a grid (see Table 2.2).

Before calling the kernel function, programmers need to use an API function `cudaMalloc()` to allocate the device memory and copy the data from the host to the device (using `cudaMemcpy()`). After the execution of the CUDA threads is finished on the device, programmers need to copy the data from the device to the host (using `cudaMemcpy()`), and free the memory allocated on the device by using an API function `cudaFree()`.

The syntax to call the kernel function is:

```
kernelFunc <<< dimGrid, dimBlock >>> (args1, args2);
```

“dimGrid” and “dimBlock” are the execution configuration parameters. Both of them are three-dimension vectors. As the name shows, “dimGrid” and “dimBlock” specify the dimensions of a grid and the dimensions of a thread block, respectively. Figure 2.2 uses an example to illustrate the CUDA API functions.

```
1: void example(int *H_var, int *H_h, int n){
2:   int size = n * sizeof(int);
3:   int *D_var, *D_d;
4:   cudaMemcpy(D_var, H_var, size, cudaMemcpyHostToDevice);
5:   cudaMemcpy(D_d, H_var, size, cudaMemcpyHostToDevice);
6:   dim3 dimBlock(256, 0, 0);
7:   dim3 dimGrid(1, 0, 0);
8:   launch_Kernel<<<dimGrid, dimBlock>>>();
9:   cudaMemcpy(H_h, D_d, size, cudaMemcpyDeviceToHost);
10:  cudaFree(D_var);
11:  cudaFree(D_d);
12: }
```

Figure 2.2: A simple CUDA example.
2.1.2 CUDA Device Memory Model

Figure 2.3: CUDA Device Memory Model.

Figure 2.3 shows a simple CUDA device memory model [5]. Each grid has two thread blocks, and each thread block has two threads for simplicity of illustration. The threads in the same thread block communicate with each other through shared memory, and each thread has its own registers. Different thread blocks use global memory and constant memory to share the data.

Figure 2.4: The mapping between CUDA and GPU Hardware.
• Block: Each thread block can hold up to 1024 threads in CUDA 3.0. Threads blocks are scheduled to execute on streaming multiprocessors (see Figure 2.4). Every thread block has its own ID, and programmers can access them using predefined variables: blockIdx. Programmers can use blockIdx.x, blockIdx.y, and blockIdx.z to access the thread block id of a thread block in different dimensions.

• Warp: Each thread block is partitioned into many warps. Each warp consists of 32 threads, and the thread IDs within a warp are consecutive and increasing. Each warp is an execution and scheduling unit in SMs, and all the threads in one warp execute the same instruction in parallel, except the situation when warp divergence occurs. Thus, we better define the block size as a multiple of the warp size, 32.

• Grid: Each grid contains many thread blocks, and these blocks have the same size. A grid of threads are generated by the CUDA runtime system when the kernel function is launched.

![Registers versus shared memory.](image)

Figure 2.5: Registers versus shared memory.

• Register: It is an on-chip memory, so it has low latency when accessed by CUDA threads. Because of the high access speed, the register is always used to store the
data accessed frequently. Every thread has its own registers, which are used to store the data private to each thread.

- **Shared memory**: It is also an on-chip memory, so it also can be accessed with a higher speed compared to the global memory. Shared memory is slower than the registers (see Figure 2.5). The data stored in the shared memory is shared by all the threads in a thread block. It is efficient to use shared memory to share the intermediate results among the threads in a thread block. Programmers declare a shared variable by adding keyword “`shared`”.

- **Global memory**: The global memory is shared by all the threads in a grid. It is an off-chip memory, so the access speed is low. Because of the low access speed, programmers need to avoid global memory accesses, and also try to coalesce the global memory accesses as much as possible.

- **Constant memory**: It is also an off-chip memory. Programmers use keyword “`constant`” to declare a constant variable. Constant variables can be shared by all the threads in a grid, but no CUDA threads can modify the constant variable during kernel execution. Before using the constant memory, programmers need to allocate constant memory from the host function. Programmers also need to transfer the constant variables from the host to the device.

### 2.1.3 Atomic Operations

CUDA uses atomic operations to coordinate the parallel execution of multiple threads. An atomic function performs a read-modify-write operation on one 32-bit or 64-bit word residing in global or shared memory. For the read-modify-write operation, in order to guarantee the correctness of the result, the read-modify-write operation must be performed without interference from other threads. In other words, no other thread can access this
address until the operation is complete. Here are some atomic functions [7] used in this work:

- atomicCAS(): This atomic function has three parameters, an address `addr` in which the old value `old` is located, a compared value `cmp`, and a new value `new`. It first compares `old` with `cmp`. If they are equal, then `new` is stored to `addr`. If not, there is no change. No matter the new value `new` is stored in the address `addr` or not, as long as the memory location with the address `addr` still holds the old value `old`, the function returns the old value `old`. Notice that the return type of this function can only be three types: “int”, “unsigned int”, and “unsigned long long int”. Here is an illustration example:

  ```c
  int atomicCAS(int* address, int compare, int val);
  ```

- atomicAdd(): This atomic function has two parameters. One is the address `addr` of the memory location that holds the old value `old`, and the other is the value `val` to be added. After the complete of this atomic function, the result of `(old + val)` is stored to the memory location with the address `addr`. It can only return three types “int”, “unsigned int”, and “unsigned long long int”. Here is an illustration example:

  ```c
  unsigned int atomicAdd(unsigned int* address, unsigned int val);
  ```

### 2.1.4 Dynamic Parallelism

CUDA dynamic parallelism [5] is an extension of the CUDA programming model enabling a CUDA kernel to create new thread grids by launching new kernels. In other words, a kernel function can launch another kernel. Previously, only host functions could launch kernels in a CUDA program. However if there is a new task discovered while executing a kernel
and more threads are needed to finish this task, then dynamic parallelism enables a kernel to launch new kernels without burdening the host. In the situation illustrated in Figure 2.6(a), without using dynamic parallelism, the current kernel must be completed first. And then, after the current kernel finishes, the host function will launch a new kernel to finish the new task. This can increase the frequency of the communication between CPU and GPU, and the overhead of the whole program. The performance will be degraded correspondingly. However, using dynamic parallelism can reduce the communication overhead. As Figure 2.6(b) shows, when a new task is discovered, the current kernel does not need to be terminated and returns to the host. The current kernel will continue executing, and also create a new thread grid to work on the new task.

![Kernel launch patterns](image)

(a) Kernel launch patterns without dynamic parallelism.
(b) Kernel launch patterns with dynamic parallelism.

Figure 2.6: Kernel launch patterns.

For the programmers, launching a new kernel from a kernel has the same syntax as launching a kernel from the host code, which is shown as follows:

```
kernel <<< dimGrid, dimBlock, Ns, S >>> (args1, args2, args3);
```

Figure 2.7 shows an example of the nested kernel launch. A grid $B$ is called a child grid of
grid A when it is launched by grid A (which is called the parent grid). The parent grid A is not considered complete without its child grid B completing execution.

Figure 2.7: An example of nested launch.

2.2 MapReduce

The MapReduce framework was proposed by Google [1, 9, 17, 19]. This programming model is inspired by the functional languages and used to process large data sets (> $1TB$) in parallel. With the traditional serial programming method, it always takes a lot of time in processing the large data set, such as combining the web logs and the relational data [13]. The MapReduce framework can decrease the total execution time in processing the large datasets.

The MapReduce framework can be easily used by the programmers who do not know the distributed computing [2] to process the large datasets. The programmers only need to focus on satisfying business needs, and leave the parallel computing to the MapReduce runtime system.
Figure 2.8: The execution of the MapReduce framework.

Figure 2.8 shows the execution of MapReduce framework. As Figure 2.8 shows, the client submits a job to the MapReduce framework, then the MapReduce master divides the job into some small units and distributes them to the map and reduce workers. The map function takes a key-value pair as its input, and generates zero or more key-value pairs:

$$\text{Map}(\text{Key}_1, \text{Value}_1) \rightarrow \text{list}(\text{Key}_2, \text{Value}_2)$$

The next phase is the shuffle and sort phase. This phase is to group all the values belonging to the same key together and pass them to the reduce function (see Figure 2.9). The reduce
function takes the key and value list as an input, and produces zero or more key-value pairs as output:

\[ \text{Reduce}(\text{Key2, list(Value2)}) \rightarrow \text{list(\text{Key3, Value3})} \]

Algorithm 1 shows the pseudocode of a MapReduce example: counting the appearances of each word in a document to illustrate how the MapReduce framework works.

**Algorithm 1: WordCount application.**

**Map Function:**

1. Map(String key, String value){
   //key: the input file's name
   //value: the content of this file
   2 for each word in value do
   3     emit_intermediate_output(word, 1);
   end
}

**Reduce Function:**

4. Reduce(String key, List of Values){
   //key: a word
   //value: values associated with this word
   5     int sum = 0;
   6 for each val in Values do
   7     sum += val;
   8     emit_output(key, sum);
   end
}

The map function emits each word with an associated number of occurrences (‘1’ in this example). The reduce function adds together all the numbers emitted for a particular word. In addition, the user writes code to fill in a MapReduce specification object with the names of the input and output files, and optional parameters. The user then invokes the MapReduce function, passing it the specification object. Finally, the users code is linked together with the MapReduce API function library. And the map functions and the reduce functions are automatically parallelized by the MapReduce framework.
2.3 Hash Table

Hash Table is an efficient data structure to process key-value pairs [4, 18]. It has a special function called hash function. The hash function is used to calculate the index for a given key.

Hash function has two parts. The first part is the hash code function that converts the given key to an integer $I$. The second part is a compression function, which maps $I$ into the range of 0 to N-1. N is the number of buckets in the hash table.

However, two or more different keys might map to the same bucket of the hash table, which is called collision. Open-addressing and chaining are used to solve the collision problem.

Open-addressing has three methods that can solve the collision problem:

- **Linear probing**: If the bucket $b[i]$ that key $k_1$ maps to is occupied by another key $k_2$, then it probes the next bucket, i.e., $b[(i + 1) \mod N]$. If this bucket is also occupied by another key $k_3$, then it probes the next one, until it finds an empty bucket $b[m]$. Then $k_1$ is inserted into this bucket $b[m]$ (see Figure 2.10).

![Figure 2.10: An example of handling the collision problem using linear probing.](image)

- **Quadratic probing**: It is similar to linear probing. Only the probing method is different. If the bucket $b[i]$ that $k_1$ maps to is occupied, then it will go to bucket $b[(i + g(j)) \mod N]$ iteratively, where $g(j) = j^2, j = 1, 2, 3, \ldots$, until it finds one empty bucket $b[m]$.

"
• Double hashing: It uses another hash function $h(k)$ to find the empty bucket $b[m]$. If the bucket $b[i]$ that $k_1$ maps to is occupied, then it goes to bucket $b[(i + g(j)) \mod N]$ iteratively, where $g(j) = j \times h(k), j = 1, 2, 3, \ldots$, until it finds one empty bucket $b[m]$

In a chained hash table, if there are two or more keys mapped to the same bucket $b[i]$, then in this bucket $b[i]$, a linked list is used to chain all the keys mapped to this bucket together to handle the collision problem as illustrated in Figure 2.11.

![Figure 2.11: An example of the chained hash table.](image)

### 2.4 MapCG

The MapCG framework [14] implements MapReduce framework to achieve portability between CPU and GPU. The MapCG framework can be divided into two major parts: the first part is the MapCG API functions, and the other is the MapCG runtime system. The users only need to focus on the MapCG API functions (Map function and Reduce function) for a specific application, and the application with the provided map function and reduce function will be automatically parallelized by the MapCG runtime system.
In the Map phase, the MapCG framework breaks the large input data into equal-sized pieces using the $\textit{Splitter()}$ function, and sends these pieces to the Map function. The Map function processes them and emits intermediate key-value pairs into a chained hash table (see Figure 2.12). The Reduce function gets the intermediate key-value pairs from the hash table and produces the final result.

![Figure 2.12: An intermediate data structure of MapCG.](image)

As Figure 2.12 shows, the MapCG framework uses a chained hash table to store the key-value pairs. It chains the keys in the same bucket together, and links the values belonging to the same key into a value-list. There still exist problems in the intermediate data structure of the MapCG framework.

![Figure 2.13: Mapping many keys into the same bucket.](image)
In the map phase, a collision would occur when many keys are mapped into the same bucket and multiple insertions are done simultaneously (see Figure 2.13). Since the insert operation is a read-modify-write operation, all of these insertion operations must be executed in serial. Otherwise, some keys will be failed to be inserted into the key-list. For example, if there are three threads, and each of them is processing one key. All of these three keys are mapped to the same bucket (Assume there is a three-key key list in the bucket already, key1, key2, key3). Each thread gets the key-list pointer before any of them insert the key into the key-list. Then, each thread inserts its own key into the key-list. Thus, the final key-list after the three keys are inserted by the three threads might only contain four keys: key1, key2, key3, and the key that each thread inserted if the insertion is not done by the atomic operations (see Figure 2.14(a)). Apparently, the final result of the key-list is wrong. The correct key list should contain six keys, as shown in Figure 2.14(b). Thus, in MapCG, atomic operations are used to solve this problem. Atomic operation is a serial operation. When a thread is executing an atomic operation, the execution cannot be interrupted or switched to another thread, so the performance will be influenced by the atomic operations.

The same collision problem also occurs on inserting the values associated with a particular key into the value list of this key (see Figure 2.15). Each thread holds a value associated with key2, and each of them gets the value-list pointer for key2 before any of them insert the value into the value-list for key2. Then each thread inserts its own value associated with key2 into the value-list for key2. Apparently, the similar result occurred. There might be only one value inserted into the value-list of key2 successfully. Again, atomic operations can be used to solve this problem. However, the performance will be degraded by the atomic operations. Figure 2.16 shows the final result after the values associated with key2 are inserted into the value list of key2 without using atomic operations, and using atomic operations. The MapCG framework does not handle the unbalanced
(a) The wrong result after the keys are inserted in parallel without using atomic operations.

(b) The final result after inserting keys with atomic operations.

Figure 2.14: Wrong key insertion versus Key insertion with atomic operations.

problem in the reduce phase, which can result in performance loss. In the reduce phase, each thread deals with a task, but the workload of each task might be different (see Figure
2.17). Obviously, the thread that works on the small workload task finishes first, and returns to the idle status. If all the threads finish their tasks, except the thread with the largest workload task, then all the threads that already finished have to wait for the last thread to finish its task. Only after all the threads finish their tasks, they can be scheduled to work on another new job.

Taking a 100M “WordCount” file as an example: Some words, like “a” and “the”,
have very high frequent appearance. But a person’s name may just appear once in a text file. In Figure 2.18, the x-axis represents the number of appearances of a word, and the y-axis represents the percentage of the words whose appearances are within a certain range. There are approximately 47% of the words in this 100M text file that only appear once. The
words which appeared twice in this 100M text file account for about 21% of the words. The very frequently used words (>128 times) only has a very small percentage (no more than 10%) in this 100M text file. So, the threads that handle the less frequently appeared word will finish first, and wait for the threads that process the very frequently appeared words to finish their tasks. Because of this, most of the execution time of some threads is wasted in waiting for others. This influences the overall performance of the system.
Chapter 3: A Workload Balance

MapReduce Framework

The MapReduce framework is a programming model proposed by Google to process large datasets. Many MapReduce frameworks are implemented on different platforms, including Phoenix (based on multicore CPUs), MapCG (based on GPUs), and StreamMR (based on GPUs). To the best of our knowledge, at this time, MapCG is the state-of-the-art MapReduce framework on GPGPUs, which uses parallel chained hash table as the core data structure to group intermediate results of the map phase by keys. However, MapCG does not handle the unbalanced workload problem in the reduce phase. To improve the performance of the MapReduce framework on GPGPUs, we propose and develop a workload balanced MapReduce framework (B_MapCG) based on the MAPCG framework to reduce the collision problem while inserting key-value pairs in the map phase, and to handle the unbalanced workload problem in the reduce phase. In this chapter, we present the details of our proposed B_MapCG framework. The B_MapCG framework has a map phase and a reduce phase, which will be introduced as follows in detail.
3.1 Map phase

Figure 3.1: The flowchart of the map phase of B_MapCG.

Figure 3.1 shows the flowchart of the map phase of B_MapCG. This flowchart shows the high level working procedure, and working mechanism of the map phase of B_MapCG, which will be introduced in detail as follows.

In the map phase of B_MapCG, to sort the key-value pairs, there are two main data structures proposed: one data structure is the key structure, which contains four variables: a key name $k$, a key size $key_size$, a pointer to the value-list $valuelist$ of this key, and a
pointer to the segmentation table \texttt{seg	able}. The other data structure is the value structure, which contains three variables: a value name \texttt{val}, a value size \texttt{val.size}, and a pointer to the next value (see Figure 3.2).

\begin{verbatim}
1: struct Value{
2:     char* val;
3:     unsigned int val_size;
4:     Value* next;
5: }
6: struct Key{
7:     char* k;
8:     unsigned int key_size;
9:     Value* valuelist;
10:     Value** seg_table;
11: }
\end{verbatim}

Figure 3.2: The definition of the key nodes and the value nodes.

In the map phase of B\_MapCG, we choose a mixture of the open addressing hash table based on linear probing and the chained hash table as the intermediate data structure, as shown in Figure 3.3. In the chained hash table, multiple keys could be chained together in one bucket. Therefore, in the MapCG framework, it needs to use the atomic operations to ensure the correct insertion of multiple keys mapped to the same bucket, which means only one key can be inserted into this bucket at a time, and all the other keys mapped to this
bucket need to wait in the inserting queue until this key’s insertion is finished. For the open addressing hash table, one bucket can only hold one key. Thus, if the bucket that one key mapping to is occupied by another key, then this key needs to find another available bucket by probing instead of waiting in the same inserting queue, which can reduce the number of collisions when multiple keys are mapped to the same bucket.

![Figure 3.4: The number of collisions occurred in the WordCount implemented on the MapCG.](image)

The other collision problem can also occur when multiple values are to be chained in a key node. Figure 3.4 shows the number of collisions of the WordCount application implemented on the MapCG framework with a 100M text file as the input. It is acceptable when the number of collisions is small (less than 512) for a large input file. However, if the number of collisions is bigger than 512, a lot of time will be wasted in waiting. In the worst case, there are 131072 collisions as shown in Figure 3.4, which is caused by the atomic operations, as illustrated in Figure 3.5.

In Figure 3.5, each column represents one thread, and each row represents one clock
cycle. The circle with straight lines means this thread holds a value waiting to be inserted. The square with points means the value was inserted successfully. If the thread does not have the priority to insert the value into the hash table within this clock cycle, then it will repeatedly try to insert the value in the following clock cycles, until it inserts the value successfully. The 4th column in Figure 3.5 shows this worst situation. This is why there are 131072 collisions while inserting values in the MapCG framework. To reduce the number of collisions, we add a segmentation table for some key nodes in our proposed B_MapCG.

In B_MapCG, we use a threshold $T$ ($T = 2$) to decide whether a key node needs a segmentation table or not. If the number of collisions is less than $T$, then there is no need to add the segmentation table. The values for a particular key are chained in the value list for this key node. Otherwise, a segmentation table (the default size is 8 buckets) is added for this key node and the value list with values already inserted for this particular key is moved to the first bucket of the table. The values belonging to this key processed by other threads will be put into this segmentation table according to their warp ID. It is simple to decide which bucket the value processed by a particular thread should map to. It just uses
Algorithm 2: The pseudocode of adding a segmentation table atomically.

1 if CASPTR(address of current_key->next, seg_table, 0xffffffffffffffff) then
2     creat a segmentation table on this key node.
end

the modulo result of the warp ID and the segmentation table size as the index:

\[ index = \text{warpID} \mod \text{table.size}; \]

If the number of collisions is still bigger than \( T \) after adding a segmentation table with the default size, then the size of the segmentation table will be doubled repeatedly, until the number of collisions is smaller than \( T \). The biggest size of the segmentation table is 2048 buckets.

There is only one segmentation table in a key node, so atomic operations are used to maintain the correctness when multiple threads try to add a segmentation table in a key node. Algorithm 2 shows the pseudocode of this atomic operation. For the key node that does not have a segmentation table, it sets \( \text{seg_table} = \text{NULL} \). Line 1 is an “if” condition that contains the “atomic compare and swap” operation. The \text{CASPTR} is a modified version of \text{atomicCAS}. It also does the compare-and-swap operation like \text{atomicCAS} does. The only difference between them is that \text{CASPTR} returns a Boolean type. There is an \text{atomicCAS} function inside the \text{CASPTR} function. If the return value of the \text{atomicCAS} function is equal to the old value (the second parameter in the \text{CASPTR} function), then the \text{CASPTR} function returns “true”. Otherwise, it returns “false”. Obviously, in this algorithm, the values of \text{current_key->next} and \text{seg_table} are “NULL” initially when a key node does not have a segmentation table. After this “atomic compare and swap” operation, the value of \text{seg_table} is no longer “NULL”, so the “if” condition cannot be true anymore when other threads try to add a segmentation table in this key node. Thus, it ensures only
Algorithm 3: The pseudocode of \texttt{insert\_value()} function.

\textbf{Input:} k, val \\
\textbf{Output:} the number of collisions \\
1. \texttt{collision} $=$ 0; \\
2. \textbf{while do} \\
3. \hspace{10pt} \texttt{curHead} $=$ k->valuelist; \\
4. \hspace{10pt} \texttt{val->next} $=$ curHead; \\
5. \hspace{10pt} \textbf{if} \ CASPTR(address of k->valuelist, curHead, val) \textbf{then} \\
6. \hspace{20pt} \texttt{return collision;} \\
7. \hspace{10pt} \texttt{collision}++; \\
end

one segmentation table is created for a key node.

Algorithm 3 inserts a new key value pair into the value list of a key, and also returns the number of collisions. There is an atomic operation at line 5 to ensure the correctness of parallely inserting the values into the key node. Line 5 of Algorithm 3 compares the pointer \texttt{k->valuelist} with the head of the current value list \texttt{curHead}. If they are equal, it means no value nodes \texttt{val2} have been inserted into this key node before \texttt{val}. Thus, \texttt{val} would be inserted successfully and no collision occurred. If they are not equal, it means that other values have been inserted \texttt{val2} before \texttt{val} and a collision occurred. So, the variable collision is increased by 1. Then the execution goes back to the beginning of the while loop and the insertion is tried again until it is inserted successfully.

Algorithm 4: The pseudocode of \texttt{insert\_into\_segTable()} function.

\textbf{Input:} seg\_table, val, index \\
\textbf{Output:} collision \\
1. \texttt{collision} $=$ 0; \\
2. \textbf{while do} \\
3. \hspace{10pt} \texttt{curHead} $=$ seg\_table[index]; \\
4. \hspace{10pt} \texttt{val->next} $=$ curHead; \\
5. \hspace{10pt} \textbf{if} \ CASPTR(address of seg\_table[index], curHead, val) \textbf{then} \\
6. \hspace{20pt} \texttt{return collision;} \\
end \\
7. \hspace{10pt} \texttt{collision}++; \\
end
Algorithm 4 inserts a key value pair into the segmentation table instead of the value list of a key node, and returns the number of collisions.

Algorithm 5 shows the pseudocode of the map phase in B_MapCG. The first step is to check whether there is a segmentation table in this key node. If there is no segmentation table, then insert_value() function is invoked to insert the value into the value list of this key node and the value of collision is returned too. If collision is bigger than T, then a segmentation table is added to this key node and the existing value-list for this key node is moved into the first bucket of this table. For those keys with the segmentation table added, the index of the bucket in the segmentation table for which the value of this thread should be mapped to is calculated according to the thread warp ID (line 6). Then the value is inserted into the segmentation table and the variable collision is obtained by calling the function insert_into_segTable(). If collision in this key node is still bigger than T, then the size of the segmentation table is doubled.

Algorithm 5: The pseudocode of the Map phase in the B_MapCG framework.

1. if no segmentation table on key k then
2.     collision = insert_value(k, val);
3.     if collision > T then
4.         add segmentation table and set table_size to 8 buckets;
5.         move the existing value-list into the first bucket of the segmentation table;
6.     else
7.         index = warpID % table_size;
8.         collision = insert_into_segTable(seg_table, val, index);
9.         if collision > T then
10.            table_size = table_size * 2;
11.       end
12.     end
end

In the map phase of B_MapCG, there is a counter created for the value list of the keys (to count the number of nodes in the value list) that do not have a segmentation table, but not for the keys that have a segmentation table. This is because the counting operation also needs to use atomic operations to ensure the correctness. This will hurt the performance.
addition, in the reduce phase, different methods are used to process the keys with a segmentation table and the keys without a segmentation table. For the keys with a segmentation table, dynamic parallelism are used to do the reduction. Thus, there is no reason to use a counter for the keys with a segmentation table.

3.2 Reduce phase

The main purpose of the B_MaoCG in the reduce phase is to solve the unbalanced workload problem compared to the MapCG. Figure 3.1 shows the flowchart of the reduce phase of B_MapCG. This flowchart shows the high level working procedure, and working mechanism of the reduce phase of B_MapCG, which will be introduced in detail as follows.

Algorithm 6: The pseudocode of the Reduce phase in the B_MapCG framework.

```
if counter[tid] > T || bucket[tid].seg_table != NULL then
    counter[tid] = T;
end

totalNum = prefix_scan(counter);
num_reduceTh = totalNum / T + 1;
if tid < num_reduceTh then
    start = the first value larger than or equal to tid * 4;
    end = the first value larger than or equal to (tid+1)* 4;
    for [start, end) do
        if bucket[id].seg_table != NULL then
            Apply dynamic parallelism (generate child-threads);
        else
            Apply normal reduction;
        end
    end
end
```

In the map phase, two types of keys are generated. One type of keys have a value-list associated with them, and the other type of keys have a segmentation table. In the
reduce phase, dynamic parallelism is used to handle the keys with a segmentation table. For the keys without a segmentation table, the normal reduction operations would be applied. Algorithm 6 shows the pseudocode of the reduce phase in the B_MapCG framework.

The first step in the reduce phase is to check whether there is a segmentation table in a key node or if the value in the counter for the key without a segmentation table is...
bigger than the threshold $T$. For checking purpose, an intermediate table is created, with each bucket corresponding to one key node and storing a counter value (The counter value for the bucket corresponding to a key node with a segmentation table is set to $\$$. Every thread checks one bucket as illustrated in Figure 3.7. If a bucket corresponds to a key node with a segmentation table, then the counter value is set to $T$ (see Figure 3.7). If a bucket corresponds to a key without a segmentation table whose counter value is larger than the threshold $T$, then the counter value is set to $T$. In the example shown in Figure 3.7, the threshold $T$ is 4.

"$\$" means there is a segmentation table in this bucket.

Line 1 and line 2 in Algorithm 6 show this checking process. After the checking process is finished, the prefix sum operations are applied and the total number of values ($totalNum$ in line 3) that need to be summed are obtained. Prefix sum [5] is applied to get the sum of the prefixes for one array:

$$Input\ array: [a_0, a_1, a_2, a_3] \implies Output\ array: [a_0, (a_0 + a_1), (a_0 + a_1 + a_2), (a_0 + a_1 + a_2 + a_3)]$$

For example, assume the input array is [2, 3, 5, 6, 10]. After a prefix sum operation, the prefix sum output result is [2, 5, 10, 16, 26]. The prefix sum is computed, so it is easy for the threads to partition the counter array $counter$ into blocks with a similar size, so the workload is close to balance for each thread in the reduce phase.
After the data is partitioned into blocks, one thread processes one block, as shown in Figure 3.8. Thus, the number of the threads needed for partitioning is calculated first. Line 4 is used to calculate the total number of threads needed, and line 5 is an “if” condition to check whether the thread is valid to do the partitioning. The threads that passed the “if” condition will search the whole array and find the beginning index and the ending index of a block. Each thread starts at an index that the first value in this array is bigger than or equal to \( tid \times 4 \) and ends at an index that the first value in this array is bigger than or equal to \((tid + 1) \times 4\) (see Figure 3.8).

We use a parallel binary search to search for the starting index and ending index with the input array sorted in ascending order. Algorithm 7 shows the pseudocode of the parallel binary search. \( base \) is the value to find (line 2). \( a \) is the beginning index and \( b \) is the ending index (line 3). There is no data dependency, so each thread can do this binary search in parallel. If \( b \) is bigger than \( a + 1 \), then it will go into the “while” loop, and search the array \( \text{counter} \). There is a middle index \( c \) which is an average of \((a + b)\) (line 5), which is used to check the location of \( base \). The input array is sorted in ascending order, so if \( base \) is bigger
than $counter[c]$ (line 6), it means the index of $base$ should be located at the right of $c$. Thus, it changes $a$ to $c+1$ (line 7). Otherwise, it sets $b$ to $c$ (line 8). And then, the execution goes back to the beginning of the “while” loop and the search is applied in a smaller range again until it finds the expected index. Line 9 to line 12 is used to ensure the correct index is found.

**Algorithm 7:** The pseudocode for parallel binary search.

```
Input: counter
Output: index
for tid = 0 to n in parallel do
    base = tid*4;
    a=0, b=counter.length-1;
    while b>a+1 do
        c=(a+b)/2;
        if base > counter[c] then
            a=c + 1;
        else
            b=c;
        end
    end
    if counter[b] >= base then
        index = b;
    end
    if counter[a] >= base then
        index = a;
    end
end
```

After the beginning index and the ending index are found, the reduction operation would be applied for the keys from $start$ to $end$. Again each thread will check whether this key has a segmentation table or not. If it does, then this thread will use dynamic parallelism to do parallel reduction. It means this thread (parent thread) will generate more threads called child threads according to the number of buckets in the segmentation table ($table_size$). Each of the child threads does a reduction operation for the values in the value list associated with a bucket in the segmentation table and returns the reduction result to the
array allocated in the global memory to store the result of each child thread. After all the child threads finished their reduction tasks, the parent thread will also apply the reduction operation on the array that stores the results that all the child-thread generated. Finally, the final reduction result is generated for this key node. If there is no segmentation table in this key node, then the normal reduction operation is applied and a final result for this key node is also generated.
Chapter 4: Evaluation

We evaluate the performance of our proposed B.MapCG on Nvidia’s Tesla K40 GPU system. Four benchmarks (InvertedIndex, WordCount, PageRank, and MentoCarlo) with 8 different datasets are selected as test workloads to evaluate the performance of our proposed B.MapCG.

4.0.1 Platforms

We introduce the test platforms and the test benchmarks of B.MapCG in this section.

The B.MapCG framework is tested on a PC with the Inter Core i7 processor and an NVIDIA Tesla K40 GPU. Core i7 has 4 cores with a base frequency of 4GHz. The maximum memory bandwidth is 25.6GB/s. Tesla K40 features 1 Kepler GK110B GPU with 12GB GDDR5 memory and 2880 CUDA parallel processing cores. The GPU frequency is 745MHz. The Peak single-precision floating point and peak double-precision floating point are 4.29Tflops and 1.43Tflops, respectively.

4.0.2 Benchmarks

We choose 4 benchmarks with 8 different datasets to evaluate the performance of our proposed B.MapCG. The four test benchmarks are illustrated as follows.

- **InvertedIndex**: It traverses the HTML file, and extracts all the web links in this HTML file. In the map phase, it traverses the web links in the HTML file. When
it extracts one link, then it will emit the link as a key with a value “1”. The reduce function calculates the summation for each web link.

• **WordCount**: It counts the number of appearances of a word in each text file. The map function processes a small unit of the input file, and emits a word (the key) with a value “1” (the value) as the intermediate key-value pair output. The reduce phase does the summation for each word, and returns the total number of appearances of each word as a result.

• **PageRank**: PageRank is an algorithm that calculates the weights to estimate the importance of a website. Figure 4.1 shows a simple illustration example. Three pages (B, C, and D) are linked from page A. Thus, page A has a chance of 1/3 to jump to one of these three pages. Assume the PageRank of A is “1”, then each of the three pages, B, C, and D can get “1/3” PageRank from A. And also, two pages link into A, so A can get PageRank from these two pages. The PageRank calculation function [3] is:

\[
\text{PageRank}(p_i) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{\text{PageRank}(p_j)}{L(p_j)}
\]

Here, \(N\) is the total number of pages. \(d\) is the damping factor, which is set to 0.85. \(M(p_i)\) is a set of pages that are linked into \(p_i\). \(L(p_j)\) is the total number of pages that are linked from \(p_j\). Taking Figure 4.1 as an example, the input of the map phase is:\((A, PR_A), (B, PR_B), (C, PR_C), \) and \((D, PR_D)\). The intermediate output is:\((A, list < p_B, p_D >), (B, list < p_A >), (C, list < p_A, p_B >), \) and \((D, list < p_A, p_C >)\). The reduce phase takes the intermediate output as the input and generates the new PageRank of each page: \((A, PR'_A), (B, PR'_B), (C, PR'_C), \) and \((D, PR'_D)\).

• **MonteCarlo**: MonteCarlo is used to calculate the area of the dark part in Figure 4.2.
In the map phase, MonteCarlo generates points \((x, y)\) randomly and use the function:
\[x^2 + y^2 < r^2\] to calculate how many points are located in the dark area. If a point is located in the dark area, then \((0, 1)\) is emitted as an intermediate output. Here, 0 is the key, and 1 is the associated value. Otherwise, \((0, 0)\) is emitted as the intermediate output. In the reduce phase, the summation operation of these points is applied.

4.1 Results

In this section, the experimental results are presented. In the experiments, we compare the performance of our proposed B_MapCG with that of MapCG. Four benchmarks with
8 different data sizes are used to evaluate the performance. The program is executed 10 times and the average execution time is used as the execution time. The execution time of the map phase and the reduce phase is measured separately.

Table 4.1: The average execution time of the map phase of B_MapCG and MapCG, and the speedup of B_MapCG over MapCG.

<table>
<thead>
<tr>
<th>Application</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvertedIndex</td>
<td>0.630612625</td>
<td>0.68258625</td>
<td>1.0824</td>
</tr>
<tr>
<td>WordCount</td>
<td>0.715364875</td>
<td>2.676660875</td>
<td>3.7417</td>
</tr>
<tr>
<td>PageRank</td>
<td>0.45083975</td>
<td>1.111639125</td>
<td>2.4657</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>0.120385375</td>
<td>7.29363975</td>
<td>60.5858</td>
</tr>
</tbody>
</table>

Table 4.1 reports the average execution time (the time unit is second) for the map phase of B_MapCG and MapCG, and the speedup of B_MapCG over MapCG. As we can see from the table, the timing performance of B_MapCG is improved over that of MapCG for all of these 4 benchmarks. The speedup of the test benchmark InvertedIndex is 1.0824, which is not as good as other benchmarks, and that is because InvertedIndex spends too much time in finding a correct web link while inserting the key-value pairs. The MonteCarlo has the highest speedup (the speedup is 60.5858 on average) among these four benchmarks because the number of intermediate outputs is small.

Table 4.2: The average execution of the reduce phase of B_MapCG and MapCG, and the speedup of B_MapCG over MapCG.

<table>
<thead>
<tr>
<th>Application</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvertedIndex</td>
<td>0.03424</td>
<td>0.12305075</td>
<td>3.5938</td>
</tr>
<tr>
<td>WordCount</td>
<td>0.23774975</td>
<td>1.6285475</td>
<td>6.8498</td>
</tr>
<tr>
<td>PageRank</td>
<td>0.2401632</td>
<td>0.39501575</td>
<td>1.6448</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>0.014221125</td>
<td>0.318415</td>
<td>22.3903</td>
</tr>
</tbody>
</table>

Table 4.2 reports the average execution time for the reduce phase of B_MapCG and MapCG. The speedup of B_MapCG over MapCG for all the four test benchmarks is high because B_MapCG solves the unbalanced workload problem in the reduce phase. The MonteCarlo application again achieves the highest speedup (the speedup is 22.3903 on
average). For the best case of benchmark MonteCarlo, the speedup of B_mapCG over MapCG is about 37.5. (see Figure 4.3). The WordCount application achieves the second highest speedup, which is 6.84 on average. PageRanks achieves the lowest speedup, which is about 1.65, and it is still pretty good. Figure 4.3 reports the average execution time of the four test benchmarks in the reduce phase of B_mapCG and MapCG.

![Figure 4.3: The average execution time of the four test benchmarks in the reduce phase of MapCG and B_mapCG.](image)

### 4.1.1 InvertedIndex

In the map phase of the InvertedIndex application, InvertedIndex searches the web link in the HTML file as the intermediate key. Hence, this application spends a lot of time in finding the correct web link. The same web link usually does not appear multiple times in an HTML file, so the collision problem is not very serious in the map phase of the InvertedIndex. And also there is an overhead to create a segmentation table, which can be ignored when the collision problem is serious. However, if the collision problem is not very serious, this overhead could only influence the performance a little bit. Thus, the
performance of InvertedIndex in the map phase on B_MapCG is not improved substantially over MapCG (see Table 4.3 and Figure 4.4).

Table 4.3: The average execution time of the map phase of InvertedIndex on B_MapCG and MapCG, and the speedup of InvertedIndex on B_MapCG over MapCG.

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.158092</td>
<td>0.158797</td>
<td>1.0045</td>
</tr>
<tr>
<td>128</td>
<td>0.292054</td>
<td>0.300063</td>
<td>1.0274</td>
</tr>
<tr>
<td>192</td>
<td>0.423013</td>
<td>0.436838</td>
<td>1.0327</td>
</tr>
<tr>
<td>256</td>
<td>0.566263</td>
<td>0.581155</td>
<td>1.0263</td>
</tr>
<tr>
<td>320</td>
<td>0.689311</td>
<td>0.744295</td>
<td>1.0798</td>
</tr>
<tr>
<td>384</td>
<td>0.828134</td>
<td>0.909137</td>
<td>1.0978</td>
</tr>
<tr>
<td>448</td>
<td>0.9727</td>
<td>1.091514</td>
<td>1.1222</td>
</tr>
<tr>
<td>512</td>
<td>1.115334</td>
<td>1.238891</td>
<td>1.1108</td>
</tr>
</tbody>
</table>

Figure 4.4: The average execution time of InvertedIndex in the map phase of B_MapCG and MapCG.

Figure 4.5 reports the average execution time of InvertedIndex in the reduce phase implemented on B_MapCG and MapCG respectively. In the reduce phase, the workload unbalance problem influences the performance. Our proposed B_MapCG has already solved
the workload unbalance problem using dynamic parallelism. Hence, the speedup of InvertedIndex in the reduce phase is 3.6 on average on B_MapCG over MapCG.

### 4.1.2 WordCount

The performance of WordCount in both the map phase and the reduce phase on B_MapCG is much better than that on MapCG (see Figure 4.6 and Figure 4.7). This is because the serious collision problem and workload unbalance problem are solved when the WordCount application is implemented on B_MapCG. In the map phase, B_MapCG uses a segmentation table to solve the collision problem. Thus, the speedup of WordCount is 3.75 on average in the map phase on B_MapCG over MapCG.

As we can see from Figure 4.7, the timing performance of WordCount in the reduce phase on B_MapCG is much better than on MapCG. The execution time of WordCount in the reduce phase on B_MapCG is less than 0.05 seconds when the input size is 512M. The speedup of WordCount is 6.85 on average in the reduce phase on B_MapCG over MapCG.
Figure 4.6: The average execution time of WordCount in the map phase of B_MapCG and MapCG.

Figure 4.7: The average execution time of WordCount in the reduce phase of B_MapCG and MapCG.
4.1.3 PageRank

PageRank emits a web page link as the intermediate key, and the page rank of this web page as its associated value. For a small size HTML file, a web page link would not appear multiple times. Thus, the number of collisions would be small. Unfortunately, if collisions still exist in a small size file, the overhead of allocating the memory for the segmentation table can influence the timing performance. However, the overhead of memory allocation for the segmentation table could be ignored when the collision problem is serious. But when the number of collisions is too small to influence the performance, this overhead will degrade the performance in the B_MapCG framework. As shown in Table 4.4 and Figure 4.8, the execution time of PageRank on B_MapCG with an 128M HTML file as input is slower than on MapCG.

Table 4.4: The average execution time of the map phase of PageRank on B_MapCG and MapCG, and the speedup of PageRank on B_MapCG over MapCG.

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.041761</td>
<td>0.054241</td>
<td>1.2988</td>
</tr>
<tr>
<td>128</td>
<td>0.306952</td>
<td>0.149284</td>
<td>0.4863</td>
</tr>
<tr>
<td>192</td>
<td>0.375991</td>
<td>0.439398</td>
<td>1.1686</td>
</tr>
<tr>
<td>256</td>
<td>0.456001</td>
<td>0.816644</td>
<td>1.7909</td>
</tr>
<tr>
<td>320</td>
<td>0.493082</td>
<td>1.232931</td>
<td>2.5005</td>
</tr>
<tr>
<td>384</td>
<td>0.583536</td>
<td>1.57739</td>
<td>2.7032</td>
</tr>
<tr>
<td>448</td>
<td>0.624959</td>
<td>2.079653</td>
<td>3.3277</td>
</tr>
<tr>
<td>512</td>
<td>0.724436</td>
<td>2.543572</td>
<td>3.5111</td>
</tr>
</tbody>
</table>

4.1.4 MonteCarlo

The MonteCarlo is a very special test case in these four benchmarks. There is only one intermediate key, 0, in MonteCarlo, so it has a very serious collision problem. So the inserting operations are almost serialized in the map phase. Thus, the timing performance of MonteCarlo on MapCG is bad. But on B_MapCG, the execution time of MonteCarlo is
less than 0.2 seconds (see Figure 4.10) because the segmentation table reduced the number of collisions. Thus, the speedup of MonteCarlo is 61 on average in the map phase on B_MapCG over MapCG.

There is only one intermediate key being generated after the map phase for MonteCarlo, so the reduction operation only needs to be applied on one value-list in the reduce
Figure 4.10: The average execution time of MonteCarlo in the map phase of B_mapCG and MapCG.

phase. On MapCG, only one thread can be used to process this value-list and no parallelism is extracted, so the timing performance is bad. On B_mapCG, the workload unbalance problem can be easily handled using the dynamic parallelism. Thus, MonteCarlo achieves a high speedup (the speedup is about 23 on average) in the reduce phase on B_mapCG over MapCG.

Figure 4.11: The average execution time of MonteCarlo in the map phase of B_mapCG and MapCG.
Chapter 5: Conclusions and Future

Work

The MapReduce framework is proposed by Google to process large datasets, which is an efficient framework that can be used in many areas, such as social network, scientific research, electronic business, etc. Hence, more and more MapReduce frameworks are implemented on different platforms, including Phoenix (based on multicore CPU), MapCG (based on GPU), and StreamMR (based on GPU). However, these MapReduce frameworks have limitations, and they cannot handle the collision problem in the map phase, and the unbalanced workload problems in the reduce phase.

To improve the performance of the MapReduce framework on GPGPUs, in this thesis, we propose and develop a workload balanced MapReduce framework (B_MapCG) based on the MapCG framework to reduce the collision problem while inserting key-value pairs in the map phase, and to handle the unbalanced workload problems in the reduce phase.

In the map phase, the serious collision problem degrades the performance of the test benchmarks on MapCG framework. Thus, we propose to add a segmentation table in some key nodes to reduce the number of collisions to improve the map phase performance. A threshold is used to determine whether a key node needs a segmentation table or not. If the number of collisions is bigger than the threshold, then a segmentation table is added to this key node, and the value list associated with this key is put into the first bucket of the
segmentation table. When there is no value-list anymore, the next coming threads insert the values into this segmentation table directly. Otherwise, it just inserts the value into the value-list that pointed by the key.

In the reduce phase, B_MapCG solves the workload unbalance problem using dynamic parallelism. Before the reduction operation, a thread will check whether there is a segmentation table associated with a key node. If a thread finds a key that does not have a segmentation table, then normal reduction operations will be applied. If there is a segmentation table associated with the key, then the thread will use dynamic parallelism to generate more child-threads to do the reduction in each bucket of the segmentation table. After they are done, the parent thread calculates the final result.

We evaluate our B_MapCG framework on the Tesla K40 GPU with 4 benchmarks and 8 different datasets. The experimental results showed that our B_MapCG framework achieved high speed up both in the map phase and the reduce phase compared with MapCG. From the evaluation result, we can see that B_MapCG is more efficient than MapCG, especially for the application that has a serious collision problem, including WordCount, PageRank and MonteCarlo, and the applications with a workload unbalance problem, including InvertedIndex, WordCount, and MonteCarlo. So, B_MapCG framework achieves high speedup both in the map phase and the reduce phase over the MapCG framework.

However, our proposed B_MapCG framework still has limitations: The user has to know the approximate size to ensure that the open addressing hash table is big enough to hold all the intermediate keys. When the hash table is not large enough, we change the open addressing hash table back to the chained hash table to solve the problem in this thesis. However, using a chained hash table can generate more collisions than using open addressing hash table. To develop an efficient MapReduce framework, an efficient data structure [4, 18] is needed.

- In our future work, we will test our proposed B_MapCG framework with more test benchmarks, specifically, we will test our proposed B_MapCG framework with ma-
chine learning related benchmarks, including deep learning.

- We will design more efficient intermediate data structures in our proposed B.MapCG framework to further reduce the number of collisions in the map phase, and try to completely solve the workload unbalance problem in the reduce phase.

- In our future work, we will try to allocate intermediate buffers in the shared memory to further improve the performance of our proposed B.MapCG framework on GPUs.
Bibliography


Appendixes

In this section, some additional experimental results of running the four test benchmarks on B_MapCG and MapCG are reported.

A. The execution time of the four test benchmarks in the map phase on B_MapCG and MapCG:

InvertedIndex

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.158092</td>
<td>0.158797</td>
<td>1.0045</td>
</tr>
<tr>
<td>128</td>
<td>0.292054</td>
<td>0.300063</td>
<td>1.0274</td>
</tr>
<tr>
<td>192</td>
<td>0.423013</td>
<td>0.436838</td>
<td>1.0327</td>
</tr>
<tr>
<td>256</td>
<td>0.566263</td>
<td>0.581155</td>
<td>1.0263</td>
</tr>
<tr>
<td>320</td>
<td>0.689311</td>
<td>0.744295</td>
<td>1.0798</td>
</tr>
<tr>
<td>384</td>
<td>0.828134</td>
<td>0.909137</td>
<td>1.0978</td>
</tr>
<tr>
<td>448</td>
<td>0.9727</td>
<td>1.091514</td>
<td>1.1222</td>
</tr>
<tr>
<td>512</td>
<td>1.115334</td>
<td>1.238891</td>
<td>1.1108</td>
</tr>
</tbody>
</table>
### WordCount

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.151152</td>
<td>0.700204</td>
<td>4.6324</td>
</tr>
<tr>
<td>128</td>
<td>0.311622</td>
<td>0.989739</td>
<td>3.1761</td>
</tr>
<tr>
<td>192</td>
<td>0.458276</td>
<td>1.816355</td>
<td>3.9635</td>
</tr>
<tr>
<td>256</td>
<td>0.628266</td>
<td>2.435508</td>
<td>3.8766</td>
</tr>
<tr>
<td>320</td>
<td>0.782059</td>
<td>3.27359</td>
<td>4.1859</td>
</tr>
<tr>
<td>384</td>
<td>0.958888</td>
<td>3.54532</td>
<td>3.6974</td>
</tr>
<tr>
<td>448</td>
<td>1.097458</td>
<td>4.257559</td>
<td>3.8797</td>
</tr>
<tr>
<td>512</td>
<td>1.335198</td>
<td>4.3946</td>
<td>3.2913</td>
</tr>
</tbody>
</table>

### PageRank

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.041761</td>
<td>0.054241</td>
<td>1.2988</td>
</tr>
<tr>
<td>128</td>
<td>0.306952</td>
<td>0.149284</td>
<td>0.4863</td>
</tr>
<tr>
<td>192</td>
<td>0.375991</td>
<td>0.439398</td>
<td>1.1686</td>
</tr>
<tr>
<td>256</td>
<td>0.456001</td>
<td>0.816644</td>
<td>1.7909</td>
</tr>
<tr>
<td>320</td>
<td>0.493082</td>
<td>1.232931</td>
<td>2.5005</td>
</tr>
<tr>
<td>384</td>
<td>0.583536</td>
<td>1.57739</td>
<td>2.7032</td>
</tr>
<tr>
<td>448</td>
<td>0.624959</td>
<td>2.079653</td>
<td>3.3277</td>
</tr>
<tr>
<td>512</td>
<td>0.724436</td>
<td>2.543572</td>
<td>3.5111</td>
</tr>
</tbody>
</table>

### MonteCarlo

<table>
<thead>
<tr>
<th>Size(Points)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.112072</td>
<td>1.476041</td>
<td>13.1705</td>
</tr>
<tr>
<td>200000</td>
<td>0.116191</td>
<td>3.114841</td>
<td>26.8079</td>
</tr>
<tr>
<td>300000</td>
<td>0.121228</td>
<td>4.817464</td>
<td>39.7389</td>
</tr>
<tr>
<td>400000</td>
<td>0.12022</td>
<td>6.341184</td>
<td>52.7465</td>
</tr>
<tr>
<td>500000</td>
<td>0.127045</td>
<td>8.20709</td>
<td>64.5999</td>
</tr>
<tr>
<td>600000</td>
<td>0.122826</td>
<td>9.686219</td>
<td>78.8613</td>
</tr>
<tr>
<td>700000</td>
<td>0.121367</td>
<td>11.594517</td>
<td>95.5327</td>
</tr>
<tr>
<td>800000</td>
<td>0.122134</td>
<td>13.111762</td>
<td>107.3555</td>
</tr>
</tbody>
</table>
B. The execution time of the four test benchmarks in the reduce phase on B_MapCG and MapCG:

**InvertedIndex**

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.022325</td>
<td>0.028438</td>
<td>1.2738</td>
</tr>
<tr>
<td>128</td>
<td>0.028582</td>
<td>0.055614</td>
<td>1.9458</td>
</tr>
<tr>
<td>192</td>
<td>0.029996</td>
<td>0.082065</td>
<td>2.7359</td>
</tr>
<tr>
<td>256</td>
<td>0.036001</td>
<td>0.109525</td>
<td>3.0423</td>
</tr>
<tr>
<td>320</td>
<td>0.035901</td>
<td>0.136025</td>
<td>3.7889</td>
</tr>
<tr>
<td>384</td>
<td>0.038814</td>
<td>0.162791</td>
<td>4.1941</td>
</tr>
<tr>
<td>448</td>
<td>0.039366</td>
<td>0.190927</td>
<td>4.8501</td>
</tr>
<tr>
<td>512</td>
<td>0.042935</td>
<td>0.219021</td>
<td>5.1012</td>
</tr>
</tbody>
</table>

**WordCount**

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.053304</td>
<td>0.368139</td>
<td>6.9064</td>
</tr>
<tr>
<td>128</td>
<td>0.104476</td>
<td>0.723794</td>
<td>6.9279</td>
</tr>
<tr>
<td>192</td>
<td>0.153397</td>
<td>1.087338</td>
<td>7.0884</td>
</tr>
<tr>
<td>256</td>
<td>0.209562</td>
<td>1.45719</td>
<td>6.9535</td>
</tr>
<tr>
<td>320</td>
<td>0.271503</td>
<td>1.811836</td>
<td>6.6734</td>
</tr>
<tr>
<td>384</td>
<td>0.279559</td>
<td>2.168464</td>
<td>7.7567</td>
</tr>
<tr>
<td>448</td>
<td>0.367909</td>
<td>2.531143</td>
<td>6.8798</td>
</tr>
<tr>
<td>512</td>
<td>0.462288</td>
<td>2.880476</td>
<td>6.2309</td>
</tr>
</tbody>
</table>
### PageRank

<table>
<thead>
<tr>
<th>Size(M)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.023716</td>
<td>0.026417</td>
<td>1.1139</td>
</tr>
<tr>
<td>128</td>
<td>0.045181</td>
<td>0.053404</td>
<td>1.182</td>
</tr>
<tr>
<td>192</td>
<td>0.076646</td>
<td>0.182339</td>
<td>2.379</td>
</tr>
<tr>
<td>256</td>
<td>0.174552</td>
<td>0.319594</td>
<td>1.8309</td>
</tr>
<tr>
<td>320</td>
<td>0.251538</td>
<td>0.450394</td>
<td>1.7906</td>
</tr>
<tr>
<td>384</td>
<td>0.349495</td>
<td>0.579735</td>
<td>1.6588</td>
</tr>
<tr>
<td>448</td>
<td>0.454274</td>
<td>0.710398</td>
<td>1.5638</td>
</tr>
<tr>
<td>512</td>
<td>0.545904</td>
<td>0.837845</td>
<td>1.5348</td>
</tr>
</tbody>
</table>

### MonteCarlo

<table>
<thead>
<tr>
<th>Size(Points)</th>
<th>B_MapCG (s)</th>
<th>MapCG (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.012921</td>
<td>0.067736</td>
<td>5.2423</td>
</tr>
<tr>
<td>200000</td>
<td>0.01356</td>
<td>0.139395</td>
<td>10.2799</td>
</tr>
<tr>
<td>300000</td>
<td>0.013923</td>
<td>0.209523</td>
<td>15.0487</td>
</tr>
<tr>
<td>400000</td>
<td>0.014118</td>
<td>0.282793</td>
<td>20.0307</td>
</tr>
<tr>
<td>500000</td>
<td>0.014451</td>
<td>0.352133</td>
<td>24.3674</td>
</tr>
<tr>
<td>600000</td>
<td>0.01471</td>
<td>0.426169</td>
<td>28.9714</td>
</tr>
<tr>
<td>700000</td>
<td>0.014882</td>
<td>0.499629</td>
<td>33.5727</td>
</tr>
<tr>
<td>800000</td>
<td>0.015204</td>
<td>0.569942</td>
<td>37.4863</td>
</tr>
</tbody>
</table>