Map Reduce on GPU

Amulya Reddy Cheula, Rakesh Tirupathi
Department of Computer Science, Arkansas State University, Jonesboro, Arkansas, United States

Abstract - *MapReduce is increasingly gaining popularity as a programming model for use in large-scale distributed processing. The model is most widely used when implemented using the Hadoop Distributed File System (HDFS). The use of the HDFS, however, precludes the direct applicability of the model to HPC environments, which use high performance distributed file systems. In such distributed environments, the MapReduce model can rarely make use of full resources, as local disks may not be available for data placement on all the nodes. This work proposes a MapReduce implementation and design choices directly suitable for such HPC environments. One of the challenges of HPCs is how to fully take the parallelism advantage presented by the multi-core CPUs and many-core GPUs; CUDA programming language is designed to surmount this challenge by taking gain of parallelization in both CPUs and GPUs.*

Keywords: MapReduce, CUDA, programming model, Big Data, HPC.

1 Introduction

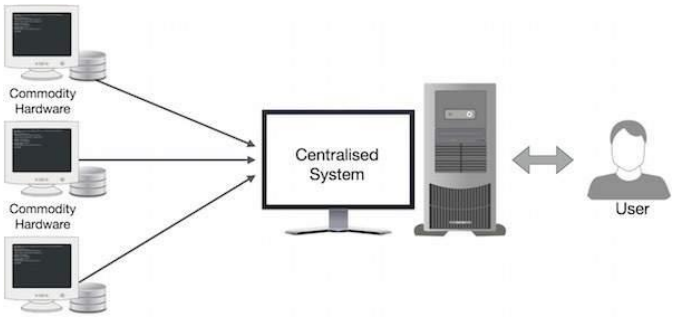
MapReduce is a programming model for writing applications that can process Big Data in parallel on multiple nodes. MapReduce provides analytical capabilities for analyzing huge volumes of complex data.

1.1 What is Big Data?

Big Data is a collection of large datasets that cannot be processed using traditional computing techniques. For example, the volume of data Facebook or Youtube need require it to collect and manage on a daily basis, can fall under the category of Big Data. However, Big Data is not only about scale and volume, it also involves one or more of the following aspects − Velocity, Variety, Volume, and Complexity.

1.2 Why MapReduce?

Google solved this bottleneck issue using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset.

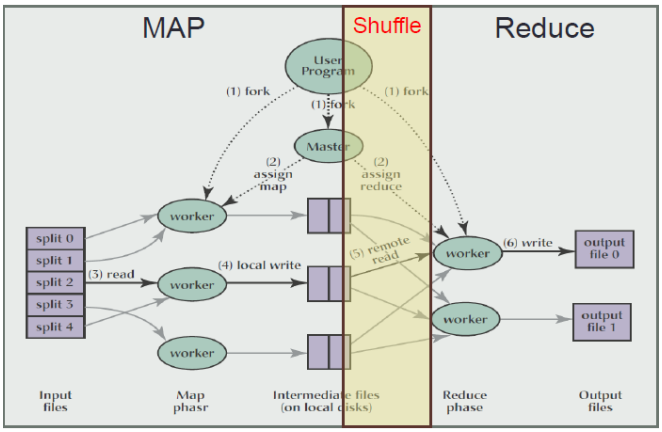


2 Working of Map Reduce

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).
- The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.

The reduce task is always performed after the map job. Let us now take a close look at each of the phases and try to understand their significance.



• **Input Phase** − Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.

• **Map** − Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.

• **Intermediate Keys** − The key-value pairs generated by the mapper are known as intermediate keys.

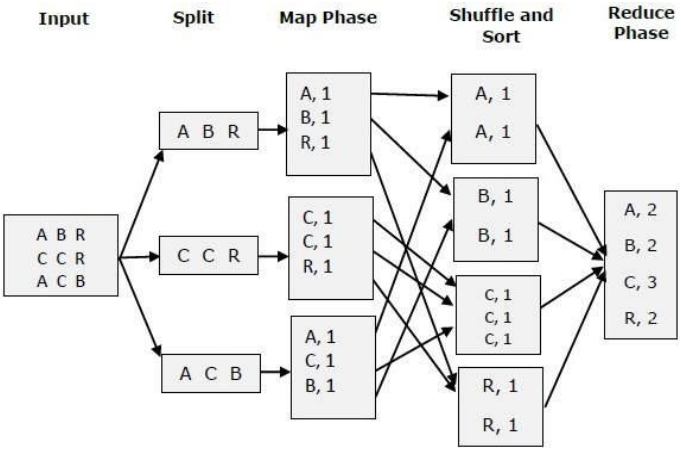
• **Combiner** − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReduce algorithm; it is optional.

• **Shuffle and Sort** − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.

• **Reducer** − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.

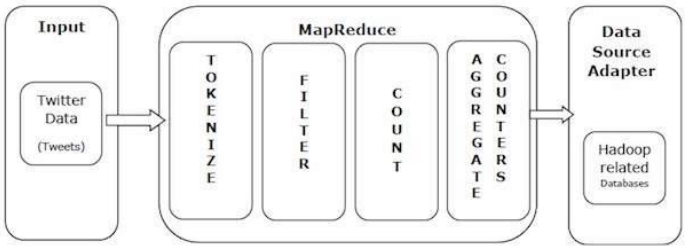
• **Output Phase** − In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

Let us try to understand the two tasks Map &f Reduce with the help of a small diagram



2.1 MapReduce-Example

Let us take a real-world example to comprehend the power of MapReduce. Twitter receives around 500 million tweets per day, which is nearly 3000 tweets per second. The following illustration shows how Tweeter manages its tweets with the help of MapReduce.



As shown in the illustration, the MapReduce algorithm performs the following actions

• **Tokenize** − Tokenizes the tweets into maps of tokens and writes them as key-value pairs.

• **Filter** − Filters unwanted words from the maps of tokens and writes the filtered maps as key-value pairs.

• **Count** − Generates a token counter per word.

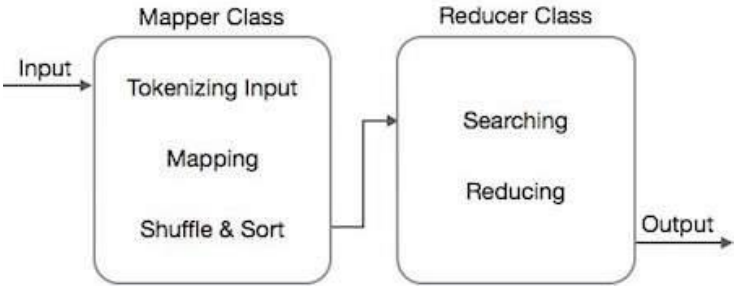
• **Aggregate Counters** − Prepares an aggregate of similar counter values into small manageable units.

3 Map Reduce Algorithm

MapReduce is generally used for processing large data sets. The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The map task is done by means of Mapper Class
- The reduce task is done by means of Reducer Class.

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.



MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

These mathematical algorithms may include the following

- Sorting
- Searching
- Indexing

- TF-IDF

3.1 Sorting

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

- Sorting methods are implemented in the mapper class itself.
- In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the Context class (user-defined class) collects the matching valued keys as a collection.
- To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of RawComparator class to sort the key-value pairs.
- The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

3.2 Searching

Searching plays an important role in MapReduce algorithm. It helps in the combiner phase (optional) and in the Reducer phase. Let us try to understand how Searching works with the help of an example.

3.2.1 Example

The following example shows how MapReduce employs Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

- Let us assume we have employee data in four different files − A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.

name, salary
satish, 26000
Krishna, 25000
Satishk, 15000
Raju, 10000

name, salary
gopal, 50000
Krishna, 25000
Satishk, 15000
Raju, 10000

name, salary
satish, 26000
kiran, 45000
Satishk, 15000
Raju, 10000

name, salary
satish, 26000
Krishna, 25000
manisha, 45000
Raju, 10000

- **The Map phase** processes each input file and provides the employee data in key-value pairs (<k, v> : <emp name, salary>). See the following illustration.

<satish, 26000>
<Krishna, 25000>
<Satishk, 15000>
<Raju, 10000>

<gopal, 50000>
<Krishna, 25000>
<Satishk, 15000>
<Raju, 10000>

<satish, 26000>
<kiran, 45000>
<Satishk, 15000>
<Raju, 10000>

<satish, 26000>
<Krishna, 25000>
<manisha, 45000>
<Raju, 10000>

- **The combiner phase** (searching technique) will accept the input from the Map phase as a key-value pair with employee name and salary. Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file. See the following snippet.

```
<k: employee name, v: salary>
Max= the salary of an first employee. Treated as max salary

if(v(second employee).salary > Max)
{
  Max = v(salary);
}
else
{
  Continue checking;
}
```

The expected result is as follows:

<satish, 26000>	<gopal, 50000>	<kiran, 45000>	<manisha, 45000>

- **Reducer phase** − Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows

<gopal, 50000>

3.3 Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as inverted index. Search engines like Google and Bing use inverted indexing technique. Let us try to understand how Indexing works with the help of a simple example.

3.3.1 Example

The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"

After applying the Indexing algorithm, we get the following output would be

"a": {2}
"banana": {2}
"is": {0, 1, 2}
"it": {0, 1, 2}
"what": {0, 1}

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

3.4 TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency. It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

3.4.1 Term Frequency (TF)

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

TF(the) = (Number of times term the 'the' appears in a document) / (Total number of terms in the document)

3.4.2 Inverse Document Frequency (IDF)

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like "is", "a", "what", etc. Thus we need to know the frequent terms while scaling up the rare ones, by computing the following

IDF(the) = log_e(Total number of documents / Number of documents with term 'the' in it).

The algorithm is explained below with the help of a small example.

3.4.2.1 Example

Consider a document containing 1000 words, wherein the word hive appears 50 times. The TF for hive is then (50 / 1000) = 0.05.

Now, assume we have 10 million documents and the word hive appears in 1000 of these. Then, the IDF is calculated as log(10,000,000 / 1,000) = 4.

The TF-IDF weight is the product of these quantities − 0.05 × 4 = 0.20.

4 CUDA Map Reduce Programming model

Here we are implementing this map reduce for the application word count. And also we are executing it in CUDA so that it can be executed on a GPU. The programming model of Map Reduce is as given below
Users implement interface of two functions:

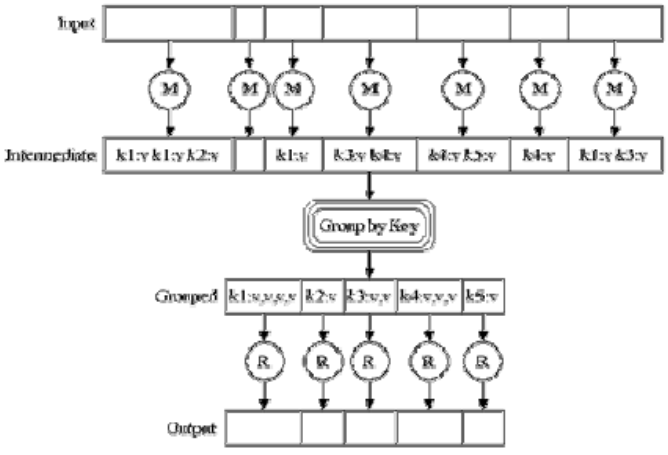
- map (in_key, in_value) -> list(out_key, intermediate_value)
- reduce (out_key, list(intermediate_value)) ->list(out_value)

Here we are using this map reduce programming model to run a simple WordCount application.

The logic for map reduce in map reduce is given as

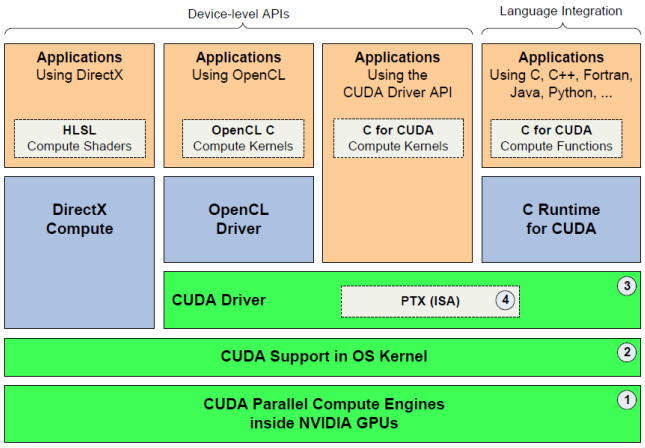
```
map(string key, string val):
// key: document name
// value: document content
for each word w in value:
emit_intermediate(w, "1");


reduce(string key, iterator values):
// key: a word
// values: a list of counts
int result=0;
for each v in values:
result+=ParseInt(v);
emit(AsString(result));
```

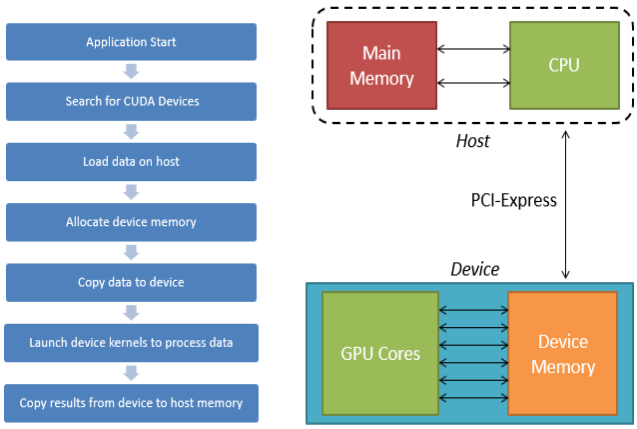


4.1.1 CUDA programming features

CUDA is also known as Compute Unified Device Architecture was developed in 2006 by NVIDIA as a general purpose parallel computing programming model, to run on NVIDIA GPUs to for parallel computations. With CUDA, Programmers are granted access to GPU memory and therefore, are able to utilize parallel computation not only for graphic application but general purpose processing (GPGPU). The memory stack model of CUDA is as given below



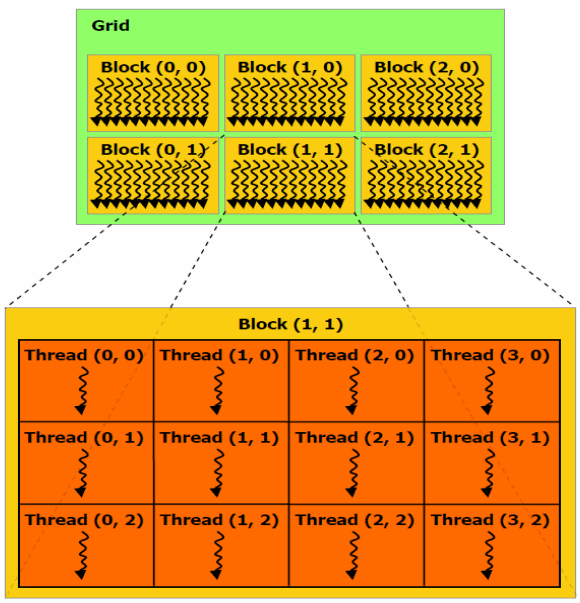
Process flow: A CUDA program execution, as shown below, is done in two parts, on the host – also known as the CPU, and on the device – also referred to as the GPU. CUDA programs interact with both the CPU and GPU during program execution. The process flow of a CUDA a program is then accomplished in the following three steps: (i) input data is copied from CPU memory to GPU memory; (ii) the GPU program is then loaded and executed; (iii) finally, results are copied from the GPU memory to the CPU memory.



***The kernel*:** The basic feature of a CUDA program is the kernel. This is the heart of the GPU program. Programmers using CUDA, a language based on the C programming language, can generate functions called kernels that when called, get executed in parallel by different CUDA threads**.** To define a kernel in CUDA, the __global__ declaring function is used; to specify the number of threads, the <<<…>>> syntax is utilized.

*A **thread:*** the smallest feature in the CUDA program model is a thread, as illustrated below. In CUDA, a kernel is a form of C program single distinct thread that depicts how that thread does computation.

*A **thread block*:** Another feature that CUDA provides to programmers is the ability to group a batch of threads into blocks. A thread block, as shown below, is a group of threads that get synchronized using barriers and communicate using shared memory.



***The grid*:** the grid, in CUDA, as demonstrated above, is a group of thread blocks that can coordinate using atomic operations in a global memory space shared by all threads. Threads in the grid get synchronized by means of global barriers and coordinate using global shared memory.

***Function types qualifiers**:* According to the CUDA manual, CUDA provides function type qualifiers that specify if a function gets executed on the device (GPU) or host (CPU) and likewise if the function can be called from a device or host. The function type qualifiers used in CUDA include : *(i)__global__* denotes a kernel function that gets called on host and executed on device. *(ii)__device__* denotes device function that gets called and executed on device. *(iii)__host__* denotes a host function that gets called and executed on host. *(iv)__constant__* denotes a constant device variable that is accessible by all threads. *(v)__shared__* denotes a shared device variable available to all threads in a block.

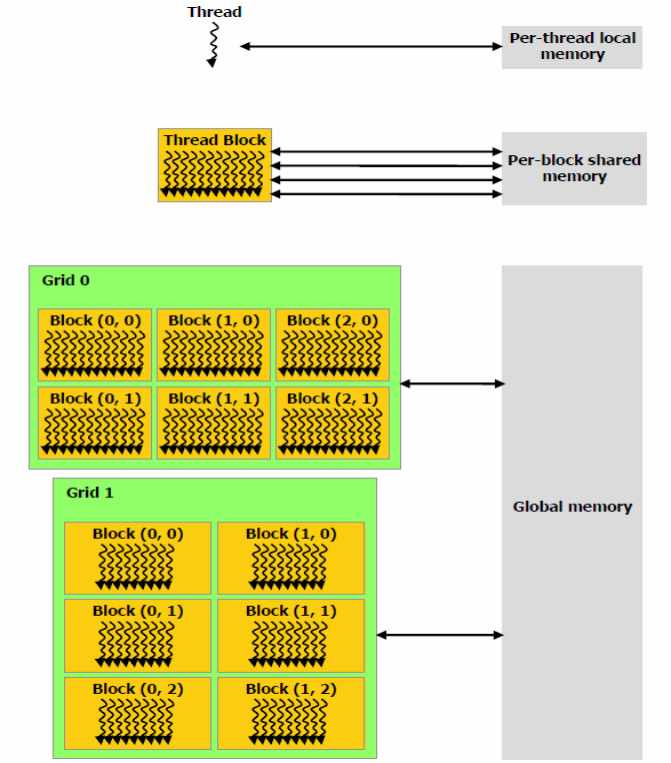
***Data types*:** CUDA provides built-in data types also referred to as vector types that are derived from the basic C language data types. These include char, short, int, long, long, float, and double. The vector types are structures that are accessible through the component values x, y, z, and w, using a constructor function structured as follows: make_<type name>; An example could include, float1 make_float1(float x, float y). A comprehensive list of data types as provided by the CUDA programming guide includes: *(i) Characters:* char1,

uchar1, char2, uchar2, char3, uchar3, char4, and uchar4. *(ii) Short:* short1, ushort1, short2, ushort2, short3, ushort3, short4, and ushort4. *(iii) Integers:* int1, uint1, int2, uint2, int3, uint3, int4, and the uint4. *(iv) Long:* long1, ulong1, long2, ulong2, long3, ulong3, long4, and ulong4. *(v) Long long*: longlong1, ulonglong1, longlong2, and ulonglong2. *(vi) Float:* float1, float2, float3, and float4. *(vii) Double*: double1 and double2.

Built-in variables: CUDA comes with built-in variables that indicate the grid and block sizes, and the block and thread indices, that are only applicable within a function and executed on the GPU; the variables include: *(i) gridDim* – denotes the dimensions of grid in blocks. *(ii) blockDim* – denotes the dimensions of block in threads. *(iii) blockIdx* – denotes a block index within grid. *(iv) threadIdx* – denotes a thread index within block.

***Thread management*:** While determinism is difficult to attain in multi-threading, CUDA provides a number of threads management functions that provide determinism supervision: *(i) __threadfence_block()* – enforces a wait until memory is available to the thread block. *(ii) __threadfence()* – implements a wait until memory is accessible to a thread block and device. *(iii) __threadfence_system()* – imposes a wait until memory is available a block, device and host. *(iv) __syncthreads()* – enforces a wait until all threads coordinate through synchronization.

Memory management: A CUDA program is always hosted on the CPU while the computation gets done on the GPU. The results are then sent back to the CPU, and as such, CUDA avails programmers with memory management tools that allocate and free memory on both host and device: *(i) cudaMalloc( )* – allocates memory on device. *(ii) cudaFree( )* – frees allocated memory on device. *(iii) cudaMemcpyHostToDevice, cudaMemcpy( )* – copies from host memory to device. *(iv) cudaMemcpyDeviceToHost, cudaMemcpy( )* – copies device results back to host memory. The CUDA memory model is as shown below :



5 CUDA Map Reduce Implementation

The implementation of map reduce is as given below : The core of the program lies in the map_reduce.cu file, which contains the generic code that performs the MapReduce job including the Cuda kernels and key/value pair partitioning.

5.1 MapReduce function

The mapreduce program is as given below :

```
#include <cstdio>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include <cuda_runtime.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

#include "config.cuh"

using namespace std;

extern __device__ void mapper(input_type *input, KeyValuePair *pairs);
extern __device__ void reducer(KeyValuePair *pairs, int len, output_type *output);

/*
 * Macro to check for GPU errors
```

```
 */
#define gpuErrChk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(
   cudaError_t code,
   const char *file,
   int line,
   bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n",
         cudaGetErrorString(code), file, line);
      exit(code);
   }
}

/*
 * An operator definition that allows comparisons between two KeyValuePairs
 * using the StrictWeakOrdering binary predicate. This does a byte by byte
 * comparison of the key, and returns True if the first pair has a key less than
 * second pair.
 */
struct keyValueCompare
{
   __host__ __device__ bool operator() (const KeyValuePair &lhs, const KeyValuePair &rhs)
   {
      void *char_lhs = (unsigned char *) &(lhs.key);
      void *char_rhs = (unsigned char *) &(rhs.key);
      for (int i = 0; i < sizeof(key_type); i++)
      {
         unsigned char *p1 = (unsigned char *) char_lhs + i;
         unsigned char *p2 = (unsigned char *) char_rhs + i;
         if (*p1 < *p2)
         {
            return true;
         }
         else if (*p1 > *p2)
         {
            return false;
         }
      }
      return false;
   }
};

// Declare mapper and reducer functions
void cudaMap(input_type *input, KeyValuePair *pairs);
void cudaReduce(KeyValuePair *pairs, output_type *output);

/*
 * Mapping Kernel: Since each mapper runs independently of each other, we can
 * give each thread its own input to process and a disjoint space where it can`
 * store the key/value pairs it produces.
 */
__global__ void mapKernel(input_type *input, KeyValuePair *pairs)
{
   for (size_t i = blockIdx.x * blockDim.x + threadIdx.x;
      i < NUM_INPUT;
      i += blockDim.x * gridDim.x)
      {
      mapper(&input[i], &pairs[i * NUM_KEYS]);
      }
}

/*
 * Reducing Kernel: Given a sorted array of keys, find the range corresponding
 * to each thread and run the reducer on that set of key/value pairs.
 */
__global__ void reduceKernel(KeyValuePair *pairs, output_type *output)
{
   for (size_t i = blockIdx.x * blockDim.x + threadIdx.x;
      i < NUM_OUTPUT;
      i += blockDim.x * gridDim.x)
   {
      int startIndex = 0;
      int count = 0;
      int valueSize = 0;
      int j;

      for (j = 1; j < NUM_INPUT * NUM_KEYS; j++)
      {
         if (keyValueCompare()(pairs[j - 1], pairs[j]))
         {
            if (count == i)
            {
               // This thread has found the correct number
               // There is a bit of warp divergence here as some threads
               // break before others, but we still make the most out of it
               // by calling the reducer at the very end, so there is not
               // any warp divergence where the bulk of the computation
               // should occur (the reducer).
               break;
            }
            else
            {
               count++;
               startIndex = j;
            }
         }
      }
```

```
        if (count < i)
        {
            // This thread doesn't need to process a key. We won't get here, but
            // this code is just there for assurance.
            return;
        }

        valueSize = j - startIndex;

        // Run the reducer
        reducer(pairs + startIndex, valueSize, &output[i]);
    }
}

/*
 * The main function that runs the bulk of the MapReduce job. Space is allocated
 * on the GPU, inputs are copied. The mapper is run. The key/value pairs are
 * sorted. The reducer is run. Output data is copied back from the GPU and
 * returned.
 */
void runMapReduce(input_type *input, output_type *output)
{
    // Create device pointers
    input_type   *dev_input;
    output_type  *dev_output;
    KeyValuePair *dev_pairs;

    // Determine sizes in bytes
    size_t input_size = NUM_INPUT * sizeof(input_type);
    size_t output_size = NUM_OUTPUT * sizeof(output_type);
    size_t pairs_size = NUM_INPUT * NUM_KEYS * sizeof(KeyValuePair);

    // Initialize device memory (we can utilize more space by waiting to
    // initialize the output array until we're done with the input array)
    cudaMalloc(&dev_input, input_size);
    cudaMalloc(&dev_pairs, pairs_size);

    // Copy input data over
    cudaMemcpy(dev_input, input, input_size, cudaMemcpyHostToDevice);
    //cudaMemset(dev_pairs, 0, pairs_size);

    // Run the mapper kernel
    cudaMap(dev_input, dev_pairs);

    // Convert the pointer to device memory for the key/value pairs that is
    // recognizable by the cuda thrust library
    thrust::device_ptr<KeyValuePair> dev_ptr(dev_pairs);

    // Sort the key/value pairs. By using the thrust library, we don't have to
    // write this code ourselves, and it's already optimized for parallel
    // computation
    thrust::sort(dev_ptr, dev_ptr + NUM_INPUT * NUM_KEYS, keyValueCompare());

    // Free GPU space for the input
    cudaFree(dev_input);
    // Allocate GPU space for the output
    cudaMalloc(&dev_output, output_size);

    // Run the reducer kernel
    cudaReduce(dev_pairs, dev_output);

    // Allocate space on the host for the output array and copy the data to it
    cudaMemcpy(output, dev_output, output_size, cudaMemcpyDeviceToHost);

    // Free GPU memory for the key/value pairs and output array
    cudaFree(dev_pairs);
    cudaFree(dev_output);
}

/*
 * Function to call the cuda map kernel and ensure no errors occur
 */
void cudaMap(input_type *input, KeyValuePair *pairs)
{
    mapKernel<<<GRID_SIZE, BLOCK_SIZE>>>(input, pairs);
    gpuErrChk( cudaPeekAtLastError() );
    gpuErrChk( cudaDeviceSynchronize() );
}

/*
 * Function to call the cuda reduce kernel and ensure no errors occur
 */
void cudaReduce(KeyValuePair *pairs, output_type *output)
{
    reduceKernel<<<GRID_SIZE, BLOCK_SIZE>>>(pairs, output);
    gpuErrChk( cudaPeekAtLastError() );
    gpuErrChk( cudaDeviceSynchronize() );
}
```

5.2 Configuration header file

The config.cuh file contains the type definitions and constants necessary for the job. These are specific for each MapReduce job. The config is as given below

```
#ifndef MAP_REDUCE_CUH
#define MAP_REDUCE_CUH

// Configure GPU parameters
#define GRID_SIZE 1024
#define BLOCK_SIZE 1024

// Set number of input elements, number of output elements, and number of keys
// per input element
#define NUM_INPUT 100000
#define NUM_OUTPUT 1
#define NUM_KEYS 1

// Example of custom input type
struct Word_Count
{
   char ch;
};

// Setting input, output, key, and value types
typedef Word_Count input_type;  //you can either give wordCount or char
typedef int output_type;
typedef char key_type;
typedef int value_type;

// Do not edit below this line

struct KeyValuePair
{
  key_type key;
  value_type value;
};

void runMapReduce(input_type *input, output_type *output);

#endif
```

5.3 Main Function

In this we define the mapper and the reducer function which will transform the input data into a set of key and value pairs.

The program is as given below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "config.cuh"

using namespace std;

/*
 * Mapping function to be run for each input. The input must be read from memory
 * and the the key/value output must be stored in memory at pairs. Multiple
 * pairs may be stored at the next postiion in pairs, but the maximum number of
 * key/value pairs stored must not exceed NUM_KEYS.
 */
__device__ void mapper(input_type *input, KeyValuePair *pairs)
{
   // We set the key of each input to 0.
   pairs->key = 0;
   char ch = input->ch;
   //pairs->key = ch;
   // We check if the input array has a space or a new line and set the value accordingly.
   //If so this will count the number of words in a file.
   if (ch == ' '||ch == '\n')
   {
      pairs->value = 1;
   }

   else
   {
       pairs->value = 0;
   }
}

/*
 * Reducing function to be run for each set of key/value pairs that share the
 * same key. len key/value pairs may be read from memory, and the output
 * generated from these pairs must be stored at output in memory.
 */
__device__ void reducer(KeyValuePair *pairs, int len, output_type *output)
{
   int wordCount = 0;
   for (KeyValuePair *pair = pairs; pair != pairs + len; pair++)
   {
      if(pair->value == 1)
      {
          wordCount++;   //increments the word count for pair values set to 1.
       }
   }
   // After calculating number of words in an input file, we will move the wordCount into the output variable.
   *output = wordCount;
}


/*
 * Main function that runs a map reduce job.
 */
int main(int argc, char const *argv[])
{
```

```
   // printf("\n My first Program. I am here\n");

    // Allocate host memory
   double start_s=clock();
    size_t input_size = NUM_INPUT * sizeof(input_type);
    size_t output_size = NUM_OUTPUT * sizeof(output_type);
    input_type *input = (input_type *) malloc(input_size);
    output_type *output = (output_type *) malloc(output_size);



   //Reading an input file and copying it into an input array
   FILE *f;
    char c;
   //char input[1000000];
    f=fopen("test.txt","rt");

   int i=0;
    while((c=fgetc(f))!=EOF)
   {
       //printf("%c",c);
      input[i].ch = c;
      i++;
    }

   //printf("\n The array is:  %s", input);
    fclose(f);

    // Run the Map Reduce Job
    runMapReduce(input, output);
   double stop_s=clock();

    // Iterate through the output array
    for (size_t i = 0; i < NUM_OUTPUT; i++)
    {
        printf("The total number of words in the file are: %d\n", output[i]);
    }

   printf("\n The value of stop is:   %f", stop_s);
   printf("\n The value of start is:  %f", start_s);
   printf("\n Time taken for word count execution in GPU is: %f", (stop_s-start_s)/double(CLOCKS_PER_SEC));


    // Free host memory
    free(input);
    free(output);

    return 0;
}
```

5.4 Results:

The commands for compilation and running the program is as given below:

Command to compile the files:
nvcc map_reduce.cu -dc
nvcc gpu_demo.cu -dc
nvcc map_reduce.o gpu_demo.o -o map_reduce

Command to Run the file:
./map_reduce

The output for the program is as shown below:

```
rakesh.tirupath@flyingdog:~/hpc_project$ nvcc map_reduce.cu -dc
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
rakesh.tirupath@flyingdog:~/hpc_project$ nvcc gpu_demo.cu -dc
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
rakesh.tirupath@flyingdog:~/hpc_project$ nvcc map_reduce.o gpu_demo.o -o map_reduce
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
rakesh.tirupath@flyingdog:~/hpc_project$ ./map_reduce
The total number of words in the file are: 9618

 The value of stop is:   2089952.000000
 The value of start is:  2715.000000
 Time taken for word count execution in GPU is: 2.087237rakesh.tirupath@flyingdo
```

The output when executed on a CPU is as shown below:

```
 The value of stop is:   5687.000000
 The value of start is:  15.000000
 Time taken for word count execution in CPU is: 5.672000
C:\Users\rakes_000\Desktop\hpc material_project\HPC tests>
```

6 Conclusion

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

7 References

IEEE papers on
[1] Kato Mivule, Benjamin Harvey, Crystal Cobb, and Hoda El Sayed. “A Review of CUDA, MapReduce, and Pthreads Parallel Computing Models”.
[2] Amit Sabne, Ahmad Mujahid Mohammed Razip, Kun Xu. “MapReduce on GPUs”.
[3] Grant Mackey, Saba Sehrish, John Bent, Julio Lopez, Salman Habib, Jun Wang. “Introducing Map-Reduce to High End Computing”.
[4] Wenguang CHEN. “Programming GPGPU with MapReduce”.
[5] Lanbo Zhang. “Large Scale Machine Learning based on MapReduce & GPU”.
[6] Jeffrey Dean, Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”.
[7] Jeff A. Stuart, John D. Owens. “Multi-GPU MapReduce on GPU Clusters”.
[8] Zacharia Fadika, Elif Dede, Madhusudhan Govindaraju, Lavanya Ramakrishnan. “Adapting MapReduce for HPC Environments”.
[9] Hui Li, Geoffrey Fox. “MapReduce Framework on GPU’s and CPU’s”.
[10] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo Tuyong Wang. “A MapReduce Framework on Graphics Processors”.