

**Abstract:**

*Map-Reduce is a programming model to parallelize computations of large tasks by parallelly solving sub-tasks. The sub-tasks are mapped to multiple 'workers' that concurrently solve their parts and the outputs of the sub-tasks are reduced back to form a solution to the primary task.*

*MapReduce is inspired by the map and reduce primitives present in LISP and many other functional languages. Map-reduce is generally implemented on colossal load of data termed as ‘Big Data’. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real-world tasks are expressible in this model.*

*Programs written in this functional style are automatically parallelized and executed. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.*

*A wide range of computing problems could be presented in MapReduce model. The scalability of MapReduce is up to thousands of machines which is suitable for the real workload. So, This proposes a MapReduce implementation on HPC environments because of taking the advantage of maximum available CPU cores and many GPU cores for achieving high speed executions and CUDA programming language created by NVIDIA can make this possible for programmers to launch kernels and hence MapReduce on a GPU with C programming language.*

*Keywords: MapReduce, CUDA, Big Data, GPU, CPU.*

**1 Introduction**

Web indexes and web worker Applications every day do the information handling errands and there will be a tremendous measure of information that is should be prepared and because of this elite is fundamental. These makes it attractive to give a conventional system to engineers to execute these assignments effectively, productively, and without any problem.

The MapReduce system is a fruitful worldview to help such information preparing applications. It was initially proposed by Google for the simplicity of improvement of web search applications on countless machines. This system gives two crude activities

(1) a guide capacity to handle input key/esteem sets and to create middle key/esteem sets, and (2) a diminish capacity to blend all moderate sets related with a similar key. With a MapReduce system, engineers can execute their application rationale utilizing the two natives. The MapReduce runtime will consequently convey and execute the assignment on various machines or different processors in a solitary machine. Accordingly, this structure decreases the unpredictability of equal programming with the goal that the designer can without much of a stretch endeavor the parallelism in the basic processing assets for complex undertakings. Supported by the achievement of the CPU-based MapReduce structures, a MapReduce system can be prepared on illustrations processors, or GPUs.

**2 Map Reduce**

**2.1 What is Map Reduce?**

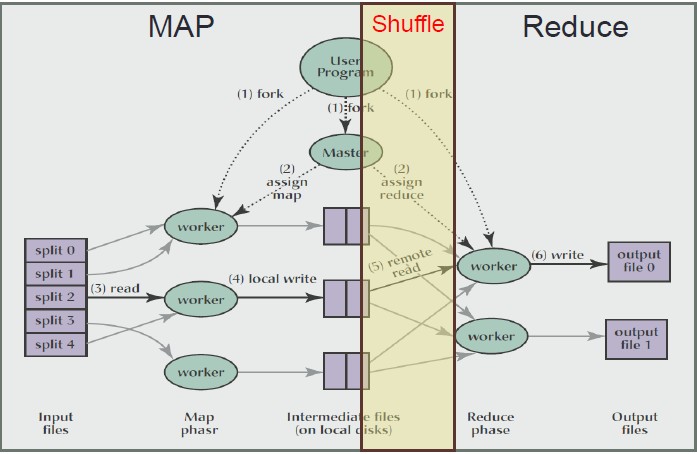
MapReduce is a processing technique and a program model for distributed computing. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. Be that as it may, when we compose an application in the MapReduce structure, scaling the application to run more than hundreds, thousands, or even huge number of machines in a group is just a setup change. This straightforward adaptability is the thing that has pulled in numerous software engineers to utilize the MapReduce model.

**2.2 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.



* **Input Phase:**

Record reader will be translating each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.

* **Map:**

It is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate 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 thegrouped 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.

**2.3 MapReduce Algorithm**

MapReduce is a Distributed Data Processing Algorithm, introduced by Google. MapReduce Algorithm is mainly inspired by Functional Programming model. MapReduce algorithm is mainly useful to process huge amount of data in parallel, reliable, and efficient way in cluster environments.

**MapReduce Algorithm Steps**

MapReduce Algorithm uses the following three main steps:

1. Map Function
2. Shuffle Function
3. Reduce Function

**Map Function:**

Map Function is the first step in MapReduce Algorithm. It takes input tasks and divides them into smaller sub-tasks. Then perform required computation on each sub-task in parallel.

This step performs the following two sub-steps:

1. Splitting
2. Mapping

* Splitting step takes input Dataset from Source and divide into smaller Sub-Datasets.
* Mapping step takes those smaller Sub-Datasets and perform required action or computation on each Sub-Dataset. The output of this Map Function is a set of key and value pairs as <Key, Value>.

**Shuffle Function:**

It is the second step in MapReduce Algorithm. Shuffle Function is also known as “Combine Function”.

It performs the following two sub-steps:

1. Merging
2. Sorting

It takes a list of outputs coming from “Map Function” and perform these two sub-steps on each and every key-value pair.

* Merging step combines all key-value pairs which have same keys (that is grouping key-value pairs by comparing “Key”). This step returns <Key, List<Value>>.
* Sorting step takes input from Merging step and sort all key-value pairs by using Keys. This step also returns <Key, List<Value>> output but with sorted key-value pairs.

Finally, Shuffle Function returns a list of <Key, List<Value>> sorted pairs to next step.

**Reduce Function:**

It is the final step in MapReduce Algorithm. It performs only one step: Reduce step.

It takes list of <Key, List<Value>> sorted pairs from Shuffle Function and perform reduce operation.

From final reduce function we will get the output as key vale pairs.

**Let’s see the Wordcount example using MapReduce:**

MapReduce is a computation that decomposes large manipulation jobs into individual tasks that can be executed in parallel across a cluster of servers. The results of tasks can be joined together to compute final results.

* Map Function – It takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (Key-Value pair).

(Map function in Word Count)

|  |  |  |
| --- | --- | --- |
| **Input** | Set of data | Pencil, Pen, pencil, pen, marker, pen, pencil, pen, marker, pencil, MARKER, PENCIL, penciL, peN, PEN, pen, PENCIL, MARKER |
| **Output** | Convert into another set of data  (Key, Value) | (Pencil,1), (Pen,1), (pencil,1), (pen,1), (marker,1),  (pen,1), (pencil,1), (pen,1), (marker,1), (pencil,1),  (MARKER,1),(PENCIL,1),(penciL,1), (peN,1), (PEN,1),  (pen,1), (PENCIL,1), (MARKER,1) |

* Reduce Function

Takes the output from Map as an input and combines those data tuples into a smaller set of tuples.

(Reduce function in Word Count)

|  |  |  |
| --- | --- | --- |
| **Input**  **(output of Map function)** | Set of Tuples | (Pencil,1), (Pen,1), (pencil,1), (pen,1), (marker,1),  (pen,1), (pencil,1), (pen,1), (marker,1), (pencil,1),  (MARKER,1), (PENCIL,1), (penciL,1), (peN,1), (PEN,1),  (pen,1), (PENCIL,1), (MARKER,1) |
| **Output** | Converts into smaller set of tuples | (PENCIL,7),  (PEN,7),  (MARKER,4) |

Workflow of MapReduce consists of 5 steps:

1. Splitting – The splitting parameter can be anything, e.g. splitting by space, comma, semicolon, or even by a new line (‘\n’).
2. Mapping – as explained above.
3. Intermediate splitting – the entire process in parallel on different clusters. In order to group them in “Reduce Phase” the similar KEY data should be on the same cluster.
4. Reduce – it is nothing but mostly group by phase.
5. Combining – The last phase where all the data (individual result set from each cluster) is combined together to form a result.

**3 Map Reduce implementation using CUDA on GPU**

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).

**3.1 CUDA Programming:**

**CUDA** is a parallel computing platform and **programming** model developed by Nvidia for general computing on its own GPUs (graphics processing units). **CUDA** enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation

A screenshot of a cell phone

Description automatically generated

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches*kernels* which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

**Kernels:**

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

A kernel is defined using the \_\_global\_\_ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> *execution configuration* syntax. Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in threadIdx variable**.**

**Thread Hierarchy:**

threadIdx is a 3-component vector. so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*.

A close up of a piece of paper

Description automatically generated

**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.

**Thread Management:**

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.

**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.

**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:**

Cuda provides function type qualifiers that tells a function where to execute whether on GPU (device) or CPU (host).

***(i) \_\_global\_\_:***

It denotes a kernel function that gets called on host and executed on device.

***(ii) \_\_device\_\_:***

It denotes device function that gets called and executed on device.

***(iii) \_\_host\_\_:***

It denotes a host function that gets called and executed on host.

**(*iv) \_\_constant\_\_:***

It denotes a constant device variable that is accessible by all threads.

***(v) \_\_shared\_\_:***

It denotes a shared device variable available to all threads in a block.

**Data Types:**

Cuda also provides built in data types that are derived from C language. It includes Char, short, int, long, float and double.

**Variables:** Cuda has some built in variable types that indicates grid, block sizes and block, thread indices.

***(i) gridDim:***

It denotes the dimensions of grid in blocks.

***(ii) blockDim***

It denotes the dimensions of block in threads.

***(iii) blockIdx***

denotes a block index within grid.

***(iv) threadIdx:***

denotes a thread index within block.

**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.

A screenshot of a cell phone

Description automatically generated

**4 MapReduce Implementation in CUDA:**

**Code:**

#include <cstdio>

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <cuda\_runtime.h>

#define GRID\_SIZE 5

#define BLOCK\_SIZE 5

#define MAX\_INPUT\_COUNT 100

#define MAX\_OUTPUT\_COUNT 100

#define MAX\_WORD\_COUNT 200

#define NUM\_KEYS 1000

typedef struct tagWord {

char szWord[300];

}Word;

typedef struct tagKeyPair {

char szWord[300];

int nCount;

}KeyPair;

typedef struct tagInputData {

Word pWord[200];

int nWordCount;

}InputData;

typedef struct tagOutputData {

KeyPair pKeyPair[NUM\_KEYS];

int nCount;

}OutputData;

typedef struct tagKeyValueData {

KeyPair pKeyPair[NUM\_KEYS];

int nCount;

}KeyValueData;

int g\_nKeyData = 0;

\_\_device\_\_ void mapper(InputData \*input, KeyValueData \*keyData)

{

int nWordCount = input->nWordCount;

keyData->nCount = nWordCount;

for (int i = 0; i<nWordCount; i++) {

KeyPair\* keyPair = &keyData->pKeyPair[i];

int j = 0;

char\* p = input->pWord[i].szWord;

while(\*p != 0)

{

keyPair->szWord[j] = \*p;

p++;

j++;

}

keyPair->nCount = 1;

}

}

\_\_device\_\_ void reducer(KeyValueData \*keyData, OutputData \*output)

{

int nCount = keyData->nCount;

output->nCount = nCount;

for (int i = 0; i<nCount; i++) {

int j = 0;

KeyPair keyPair = keyData->pKeyPair[i];

//printf("==== %s\t%d\n", keyPair.szWord, keyPair.nCount);

char\* p = keyPair.szWord;

j = 0;

while (\*p != 0)

{

output->pKeyPair[i].szWord[j] = \*p;

p++;

j++;

}

output->pKeyPair[i].nCount = 1;

}

}

\_\_global\_\_ void mapKernel(InputData \*input, int nInputCount, KeyValueData \*pairs)

{

int indexWithinTheGrid = threadIdx.x + blockIdx.x \* blockDim.x;

int gridStride = gridDim.x \* blockDim.x;

for (int i = indexWithinTheGrid; i < nInputCount; i += gridStride)

{

mapper(&input[i], &pairs[i]);

}

}

\_\_global\_\_ void reduceKernel(KeyValueData \*pairs, int nInputCount, OutputData \*output) {

for (size\_t i = blockIdx.x \* blockDim.x + threadIdx.x; i < nInputCount; i += blockDim.x \* gridDim.x) {

reducer(&pairs[i], &output[i]);

}

}

void cudaMap(InputData \*input, int nInputCount, KeyValueData \*pairs) {

mapKernel << <GRID\_SIZE, BLOCK\_SIZE >> >(input, nInputCount, pairs);

}

void cudaReduce(KeyValueData \*pairs, int nInputCount, OutputData \*output) {

reduceKernel << <GRID\_SIZE, BLOCK\_SIZE >> >(pairs, nInputCount, output);

}

void runMapReduce(InputData \*input, int nInputCount, OutputData \*output) {

InputData \*dev\_input;

OutputData \*dev\_output;

KeyValueData \*dev\_pairs;

size\_t input\_size = nInputCount \* sizeof(InputData);

size\_t output\_size = MAX\_OUTPUT\_COUNT \* sizeof(OutputData);

size\_t pairs\_size = nInputCount \* sizeof(KeyValueData);

cudaMalloc(&dev\_input, input\_size);

cudaMalloc(&dev\_pairs, pairs\_size);

cudaMemcpy(dev\_input, input, input\_size, cudaMemcpyHostToDevice);

cudaMap(dev\_input, nInputCount, dev\_pairs);

cudaFree(dev\_input);

cudaMalloc(&dev\_output, output\_size);

cudaReduce(dev\_pairs, nInputCount, dev\_output);

cudaMemcpy(output, dev\_output, output\_size, cudaMemcpyDeviceToHost);

cudaFree(dev\_pairs);

cudaFree(dev\_output);

}

void outputText(KeyPair\* keyPair, int nCnt) {

KeyPair newPair[100];

int index = 0;

for (int i = 0; i < nCnt; i++) {

bool bExist = false;

for (int j = 0; j <= index; j++)

{

if (!strcmp(newPair[j].szWord, keyPair[i].szWord))

{

newPair[j].nCount++;

bExist = true;

break;

}

}

if (bExist)

continue;

strcpy(newPair[index].szWord, keyPair[i].szWord);

newPair[index].nCount = 1;

index++;

}

for (int i = 0; i < index; i++) {

printf("%s\t\t%d\n", newPair[i].szWord, newPair[i].nCount);

}

}

int main(int argc, char const \*argv[])

{

int GPUcardcount;

cudaGetDeviceCount(&GPUcardcount);

printf("\n-----------------------------------------------------------------------------------\n");

printf("Total number of devices : %d\n",GPUcardcount);

printf("\n-------------------------------------------------------------------------------------\n");

for(int i=0; i<GPUcardcount;i++) {

/\*cudaproperties variable is declared below\*/

cudaDeviceProp prpt;

/\* Using cudaGetDeviceproperties function \*/

cudaGetDeviceProperties(&prpt, i);

/\* print all the device prpterties \*/

printf("Device number : %d\n", i);

printf("Device name : %s\n", prpt.name);

printf("warp size : %d\n", prpt.warpSize);

printf("Tcc driver : %d\n",prpt.tccDriver);

printf("PCI busID : %d\n", prpt.pciBusID);

printf("Max pitch (in bytes) : %lu\n", prpt.memPitch);

printf("clock rate in kilohertz : %d\n", prpt.clockRate);

printf("PCI device ID : %d\n", prpt.pciDeviceID);

printf("integrated : %s\n", (prpt.integrated? "Yes":"No"));

printf("computeMode : %s\n", (prpt.computeMode? "Yes": "No"));

printf("ECC enabled : %s\n", (prpt.ECCEnabled? "Yes" : "No"));

printf("concurrentKernels : %s\n", (prpt.concurrentKernels ? "Yes": "No"));

printf("can overlap Device : %s\n", (prpt.deviceOverlap? "Yes" : "No"));

printf("can map Host memory : %s\n", (prpt.canMapHostMemory ? "Yes" : "No"));

for(int n =0; n<3;n++) {

printf("Max dim %d of grid block : %d\n",n, prpt.maxGridSize[n]);

}

}

printf("Input Data:\n");

clock\_t t1, t2;

double total\_t;

t1 = clock();

FILE\* pFile = fopen("test.txt", "rt");

InputData\* pInputData = (InputData\*)malloc(MAX\_INPUT\_COUNT\*sizeof(InputData));

OutputData\* pOutputData = (OutputData\*)malloc(MAX\_OUTPUT\_COUNT \* sizeof(OutputData));

int nInputCount = 0;

if (pFile) {

char szLine[100];

while(!feof(pFile))

{

memset(szLine, 0, 100);

fgets(szLine, 100, pFile);

printf("%s", szLine);

pInputData[nInputCount].nWordCount = 0;

int nWordIndex = 0;

int nIndex = 0;

char szWord[200];

if(strlen(szLine)==1)

nInputCount++;

else{

for (int i = 0; i < strlen(szLine); i++) {

if (szLine[i] == ' ' || szLine[i] == 0x0d || szLine[i] == 0x0A)

{

szWord[nIndex] = 0;

nIndex = 0;

strcpy(pInputData[nInputCount].pWord[nWordIndex].szWord, szWord);

pInputData[nInputCount].nWordCount++;

nWordIndex++;

if (szLine[i] == 0x0d)

break;

}

else {

szWord[nIndex] = szLine[i];

nIndex++;

}

}

}

nInputCount++;

}

fclose(pFile);

}

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

runMapReduce(pInputData, nInputCount, pOutputData);

int nTotalCount = 0;

KeyPair keyPair[NUM\_KEYS];

for (int i = 0; i < nInputCount; i++) {

int nCnt = pOutputData[i].nCount;

for (int j = 0; j < nCnt; j++) {

strcpy(keyPair[nTotalCount].szWord, pOutputData[i].pKeyPair[j].szWord);

keyPair[nTotalCount].nCount = pOutputData[i].pKeyPair[j].nCount;

nTotalCount++;

}

}

cudaEventRecord(stop, 0);

t2=clock();

outputText(keyPair, nTotalCount);

printf("Total Count:%d\n", nTotalCount);

float Time\_elapsedgpu;

cudaEventElapsedTime(&Time\_elapsedgpu, start, stop);

total\_t = (double)(t2 - t1) / 1000.0;

printf("Time taken to perform word count on CPU: %f ms.\n", total\_t );

printf("Time taken to perform word count using Map reduce on GPU: %f ms.\n", Time\_elapsedgpu);

free(pOutputData);

free(pInputData);

return 0;

}

**5 References**

**[1]**[**https://devblogs.nvidia.com/introduction-cuda-aware-mpi/**](https://devblogs.nvidia.com/introduction-cuda-aware-mpi/) **[2]**[**https://deeplearning4j.org/iterativereduce**](https://deeplearning4j.org/iterativereduce) **[3]**[**http://www.nvidia.com/object/what-is-gpu-computing.html**](http://www.nvidia.com/object/what-is-gpu-computing.html) **[4]**[**http://en.wikipedia.org/wiki/CUDA**](http://en.wikipedia.org/wiki/CUDA)**.**