National University of Computer & Emerging Sciences (FAST-NU)

PICO: A MapReduce Framework for Mobile GPU Clusters

Project Supervisor
Dr. Jawwad Ahmed Shamsi

Project Team
Ahmad Rahman k122026
Abdul Basit k122290
Muhammad Usman Irfan k122023

2nd May, 2016

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

The Department of Computer Science
National University of Computer & Emerging Sciences (FAST-NU)
Main Campus, Karachi
May 2016

National University of Computer & Emerging Sciences (FAST-NU)

PICO: A MapReduce Framework For Mobile GPUs

Project Supervisor	**Dr. Jawwad Ahmed Shamsi**
Project Manager	**Ahmad Rahman**
Project Team	**Abdul Basit k122290** **Muhammad Usman Irfan k122023**
Submission Date	**2nd May'2016**

Supervisor: Dr. Jawwad Ahmed Shamsi _____

Head of Department: Dr. Jawwad A. Shamsi _____

Document Information

Category	Information
Customer	NUCES-FAST
Project Title	PICO: A MapReduce Framework For Mobile GPUs
Document	FYP II Final Report
Document Version	1.0
Identifier	JS201629 Final Report
Status	Final
Author(s)	Ahmad Rahman, Abdul Basit, M. Usman Irfan
Approver(s)	Dr. Jawwad Ahmed Shamsi
Issue Date	

Definition of Terms, Acronyms and Abbreviations

Term	Description
GPGPU	General Purpose Graphic Processing Unit
MapReduce	A big data processing framework
GPU	Graphic Processing Unit

Table of Contents

1. Context and Preliminary Investigation

1.0. Project Selection

In the modern world information retrieval is a day in and day out task. Daily billions of Internet users retrieve required information through search engines stored in gigantic data collections. To process such a huge amount of data high performance is essential [1]. Today, MapReduce is the most efficient framework to process big data [1][2].

Encouraged by the success of the CPU-based MapReduce frameworks, we are going to develop a MapReduce framework on a GPU based device regarded as the world's first embedded supercomputer, the Jetson TK1, to increase the performance of the system to large scales.

1.1. Project Background

In this section, we present an overview of the GPU as well as a background on MapReduce.

1.1.1 Graphics Processors (GPUs)

GPUs are widely used for graphics processing in personal computers as well as gaming consoles. They work as co-processors for CPU [4]. GPU programming consists of two kinds of code, one that is to be executed on CPU, known as host code and the other that is executed in parallel on GPU, known as kernel code. GPUs have their own physical memories known as device memory. Before execution of kernel code, the data is transferred to the device memory from the main memory. This transfer is considered as a bottleneck for overall performance. Therefore, the most recent GPU machines are coming with unified physical memory. The most common programming language for graphics APIs is OpenGL [5] and the most common general-purpose GPU programming languages are AMD OPENCL and NVIDIA CUDA [6].

1.1.2 GPGPUs (General Purpose computing on GPUs)

GPUs, due to their highly parallel processing power are recently attracting many developers for general-purpose computation. There are a number of applications that are written on GPUs. These include matrix multiplication [7], database applications [8][9], bioinformatics [10] and embedded system design [12]. We refer the reader to go through the survey by Owens et al. [3] for detailed information on how general purpose computing on GPUs is implemented recently.

1.1.3 MapReduce

The MapReduce framework is based on two functional programming primitives, which are defined as follows.

Map: (k1, v1) -> (k2, v2) *.
Reduce: (k2, v2*) -> v3*.

The map function takes all the input key/value pairs and produces the intermediate key/value pairs. The reduce function then takes all the intermediate key/value pairs and produces lists of output values associated with each intermediate key. The system manages the parallel execution of these two tasks at run time. In MapReduce framework there are two APIs, Map and Reduce that must be defined by the programmer.

1.1.4 Jetson TK1

Jetson TK1 is NVIDIA's embedded Linux development platform featuring a Tegra K1 SOC (CPU+GPU+ISP in a single chip), selling for $192 USD. Jetson TK1 comes pre-installed with Linux4Tegra OS (basically Ubuntu 14.04 with pre-configured drivers).

1.1.5 Tegra K1

Tegra K1 is NVIDIA's first mobile processor to have the same advanced features & architecture as a modern desktop GPU while still using the low power draw of a mobile chip. Therefore, Tegra K1 allows embedded devices to use the exact same CUDA code that would

also run on a desktop GPU (used by over 100,000 developers), with similar levels of GPU-accelerated performance as a desktop.

1.2. Literature Review

We now briefly survey the techniques of developing GPGPU primitives, which are building blocks for other applications. Govindaraju et al. [13] presented novel GPU based algorithms for the bitonic sort. Sengupta et al. [24] proposed the segmented scan primitive, which is applied to the quick sort, the sparse matrix vector multiplication. He et al. [15] proposed a multi-pass scheme to improve the scatter and the gather operations, and used the optimized primitives to implement the radix sort, hash searches. Likewise, He et al. [16] further developed a small set of primitives such as map and split for relational databases. These GPU-based primitives improve the programmability of the GPU and reduce the complexity of the GPU programming. However, even with the primitives, developers need to be familiar with the GPU architecture to write the code that is not covered by primitives, or to tune the primitives. For example, tuning the multi-pass scatter and gather requires the knowledge of the memory bandwidth of the GPU [15]. In addition to single-GPU techniques, GPUs are getting more involved in distributed computing projects such as Folding@home [11] and Seti@home [25]. These projects develop their data analysis tasks on large volume of scientific data such as protein using the computation power of multiple CPUs and GPUs. On the other hand, Fan et al. [9] developed a parallel flow simulation on a GPU cluster. Davis et al. [7] accelerated the force modeling and simulation using the GPU cluster. Goddeke et al. [12] explores the system integration and power consumption of a GPU cluster of over 160 nodes. As the tasks become more complicated, there lacks high-level programming abstractions to facilitate developing these data analysis applications on the GPU.

Traits	PICO	I-MapReduce	CellMR	MARS	MapCG	MapD
Iterative Algorithm	×	✓	×	×	×	×
Parallel Algorithm	✓	×	✓	✓	✓	✓
Single Node Compatible	✓	✓	✓	✓	✓	✓
Multiple Node Compatible (Scalability)	✓	x	×	× (Restrictions)	× (Restrictions)	x
User Control Over Scheduling threads, blocks etc.	✓	×	×	×	×	×
Optimizations provided for reducing communication overhead	✓	×	×	×	×	×

Table I. Summary of Related Work in Field

I-MapReduce [10] (formerly CGL MapReduce) is another MapReduce library like MARS, and theoretically the first to ensure the use of data streams instead of hard disk access. Intermediate key value pairs and reduce function outcomes are buffered directly to new mapper and reducer nodes for further computation. By which an efficient, iterative MapReduce algorithm with multiple consecutive MapReduce processes was developed. Recently, paradigm has shifted towards MapReduce to be executed on parallel processors resembling GPUs and IBM's Cell. Catanzaro et al. developed a single-node library for GPUs [11] but the emphasis was on numerous insignificant jobs. Mars [9] was the first significant GPU structure, though its scalability is restricted; for execution Mars uses only one GPU and in-GPU-core tasks. Additional deficiency is that the package, schedules threads and blocks, user has no control over it, making it hard to entirely exploit certain

GPU proficiencies (e.g. inter-block communication). CellMR [13] is a single-node MapReduce library on the Cell Engine that improves the in-core problem of Mars by issuing map function's data in minor portions. In CellMR the MapReduce architecture is distributed into three main portions: Map, Partial Reduction and Global Reduction. Moving on MapCG [12] is another GPU-based MapReduce library. The key objective was to allow for portable multicore MapReduce code that could execute on the GPU. Like Mars, MapCG also has restricted scalability because it uses only one GPU. MapD, exploits the immense computational power available in off-the-shelf graphics cards that can be found in any laptop or PC. But MapD is strictly suitable for data analysis, machine learning, real-time querying, and data visualization, and can cater specific application use cases. Table I further summarizes the comparison between our framework and related libraries.

1.3. Economic Feasibility

University already has the NVidia Jetson TK1 so there is no economic constraint in the main device area. The gigabit Ethernet switch however, the university doesn't have and costs about $192 USD.

1.4. Project Scope

We intend to present our stand-alone Map-Reduce library that leverages the power of GPU for large-scale computing. The framework is responsible to distribute the tasks of map, group and reduce among the hundreds of cores of a GPU in parallel fashion and provide a wrapper for map and reduce to the application developer.

The library will be responsible for creating default number of thread groups and the number of threads within a thread group for map and reduce. However, we will provide configuration parameters to the application developer to override the default configuration. Our framework will handle the sorting of intermediate results as well as the merging of final key-value pairs. Moreover, the framework will also prevent the write conflicts during concurrent read-write operations.

Hence the application developer will be provided with a very simple and familiar MapReduce interface so that he can write his program without being knowing the complexity of the GPU architecture.

1.5. Project Objectives

- **Highly Parallel Computation.**
  We harness the GPU computation power and high memory bandwidth to accelerate MapReduce frameworks, our framework exploits GPU's core level parallelism to fully utilize the computational power of a GPU and also achieve high end performance.

- **Map Reduce Programming at ease.**
  This paradigm reduces the programming complexity so that developers can easily exploit the parallelism in the underlying computing resources for complex tasks, our framework will hide the programming complexity of GPUs behind the simple and familiar MapReduce interface, and automatically manage task partitioning, data distribution, and parallelization on the processors.
  Our work further simplifies GPU programming for MapReduce programmers by providing them with a higher level and more familiar interface than the primitives.

- **Platform for High Performance Complex Computing**
  As stated earlier we intend to present our stand-alone Map-Reduce library that leverages the power of GPU for large-scale computing, the framework will be an efficient accelerator for high-performance computing and can be used for several computation intensive tasks like biometric measurements, Bioinformatics: genome and protein big data analysis, image processing, biomedical signal analysis etc.

- **Scale-out Architecture**
  Our framework will be so universal that to increase processing power one just have to add more servers/GPUs to the prior configuration.

- **Flexibility**
  This framework will be so flexible that it will be applicable to run on NVIDIA GPUs, AMD GPUs, multicore CPUs, and Hadoop-based distributed systems. Additionally, it will make available to the programmer to customize MapReduce configuration like number of threads each block should have, choose between stages of MapReduce to perform or not according to his applications desire etc.

- **One of a kind.**
  Our work is surely one of a kind, as the underlying architecture that we are using to implement our MapReduce frame work, Jetson TK1, was released recently in October 2014. Till now there have been no advancement in developing frameworks for it and MapReduce is one of them.

1.6. Deliverables

The deliverable for this project was defined through a comprehensive requirements collection process. Which are listed below:

- Four Benchmarks for our MapReduce framework that are, Word Count, Matrix Multiplication, K-Means Clustering and Inverted Index.
- Complete Library Code.
- A How to use document (User Manual).
- Proper documentation (Phase I, Phase II).
- MS Project file of Work Breakdown Structure.
- Research Paper.

2. Research

2.0. Primary Research

The device we used for our framework was released recently in Oct'2014 so very few people had info at that time. We didn't conduct any interviews for the requirements but observed various existing MapReduce frameworks and libraries like MARS, how they work, their limitations and areas where we can optimize our framework.

2.1. Academic Research

2.1.1. Development Tools

2.1.1.1. Comparison of Different Tools

The NVIDIA Jetson TK 1 was pre-installed with a slightly modified Linux version called L4T and CUDA 6.0. Due this hardware restriction we were destined to use the tools that are feasible for this hardware and can optimize and enhance its performance.

2.1.1.2. Selection of Appropriate Tools

As stated above due to our constraints we selected C/C++ as our basic programing language with CUDA programing to handle the GPUs. We also selected NVIDIA Nsight for our development purposes.

2.1.1.3. Development Tool Reasoning

NVIDIA Nsight is the ultimate development platform for heterogeneous computing. Work with powerful debugging and profiling tools that enable you to fully optimize the performance of the CPU and GPU. Not only do these feature-rich tools optimize performance, they help you gain a better understanding of your code - identify and analyze bottlenecks and observe the behavior of all system activities.

Moreover, NVIDIA Nsight Eclipse Edition is a full-featured, integrated development environment that lets you easily develop CUDA® applications for either your local (x86) system or a remote (x86 or ARM) target.

Additionally, Nsight supports two remote development modes: cross-compilation and "synchronize projects" mode. Cross-compiling for ARM on your x86

host system requires that all of the ARM libraries with which you will link your application be present on your host system. In synchronize-projects mode, on the other hand, your source code is synchronized between host and target systems and compiled and linked directly on the remote target, which has the advantage that all your libraries get resolved on the target system and need not be present on the host. Neither of these remote development modes requires an NVIDIA GPU to be present in your host system.

2.1.2. Secondary Research

As this project was on cutting edge technology so we faced many challenges in getting relevant information, very limited information was available on internet. We thoroughly researched about the device, Jetson TK 1, understood its complexity and started performing proof of concepts, reviewed research papers that were related to our project domain. For this we searched many websites, research papers and blogs which are stated in the reference section of this document.

3. Requirement Analysis

3.1. System Specifications

3.1.1 Functional Requirements

We intend to present our stand-alone Map-Reduce library that leverages the power of GPU for large-scale computing. The framework is responsible to distribute the tasks of map, group and reduce among the hundreds of cores of a GPU in parallel fashion and provide a wrapper for map and reduce to the application developer.

The library will be responsible for creating default number of thread groups and the number of threads within a thread group for map and reduce. However, we will provide configuration parameters to the application developer to override the default configuration. Our framework will handle the sorting of intermediate results as well as the merging of final key-value pairs. Moreover, the framework will also prevent the write conflicts during concurrent read-write operations.

Hence the application developer will be provided with a very simple and familiar MapReduce interface so that he can write his program without being knowing the complexity of the GPU architecture.

3.1.2 Quality Requirements

3.1.2.1 Maintainability

While coding PICO, we have keep in mind all the Object Oriented Programing and Design Patterns primitives. The code base of our framework is highly organized and simple. We tried to code on interface instead of concrete implementation. PICO has proper hierarchy of abstract, parent and child class, which ensures the maintainability and reusability of code. The framework is highly maintainable and extendable in terms of functionality and features.

3.1.2.2 Simplicity

The framework is easy to use and simple to understand. The application developer will be provided with a very simple and familiar MapReduce interface so that he can write his program without being knowing the complexity of the GPU architecture. This will not only help the developers to easily exploit the power of GPU but will also open GPU computing to larger application domains.

3.1.2.3 Extensibility

Our framework, PICO, is highly extensible, scalable and adaptable. User don't have to ensure any hard configurations, setup or do extensive coding to scale the cluster. It's more like plug and play. All the configurations and setup is dealt by our framework itself by default implementations.

3.1.3 Interface Requirements

3.1.3.1 Hardware Interface

The hardware interface of NVIDIA Jetson TK1 is built around the revolutionary NVIDIA Tegra K1 SoC and uses the same NVIDIA Kepler computing core designed into supercomputers around the world. This gives us a fully functional NVIDIA CUDA platform for quickly developing and deploying compute-intensive systems for computer vision, robotics, medicine, and more.

NVIDIA delivers the entire BSP and software stack, including CUDA, OpenGL 4.4, and Tegra-accelerated OpenCV. With a complete suite of development and profiling tools, plus out-of-the-box support for cameras and other peripherals.

4. Design

4.0 Deliverables of Process Modeling

4.0.1 Context Diagram

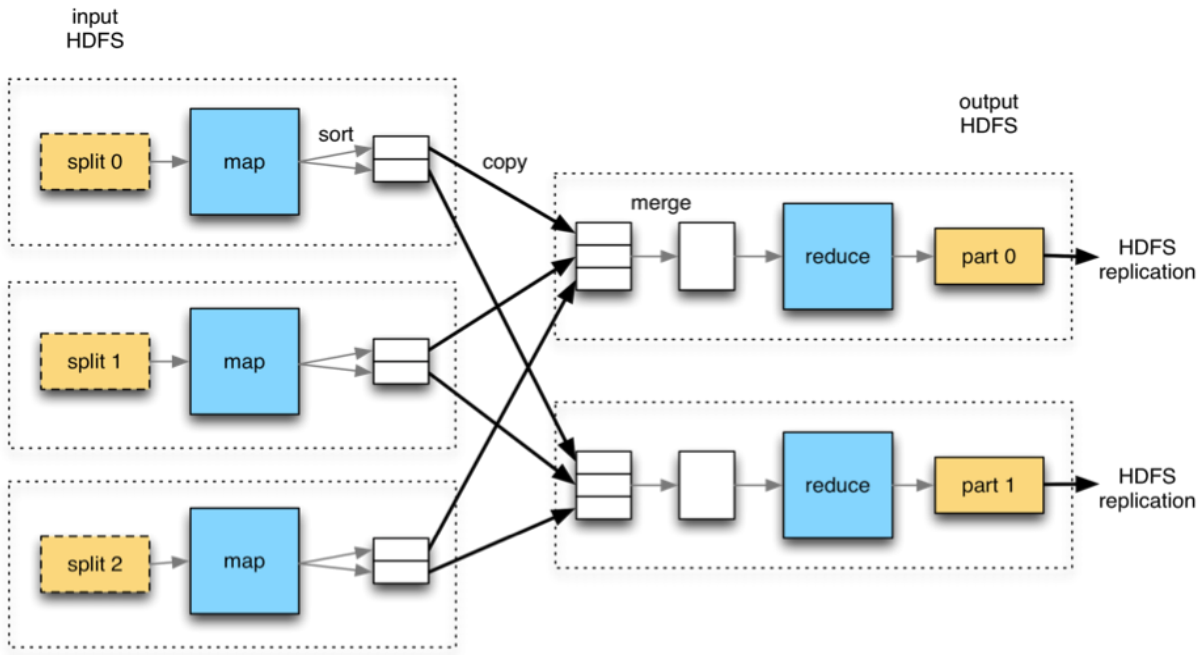


Fig. 1. MapReduce Framework System Context Diagram

4.0.2 DFD (Level 1)

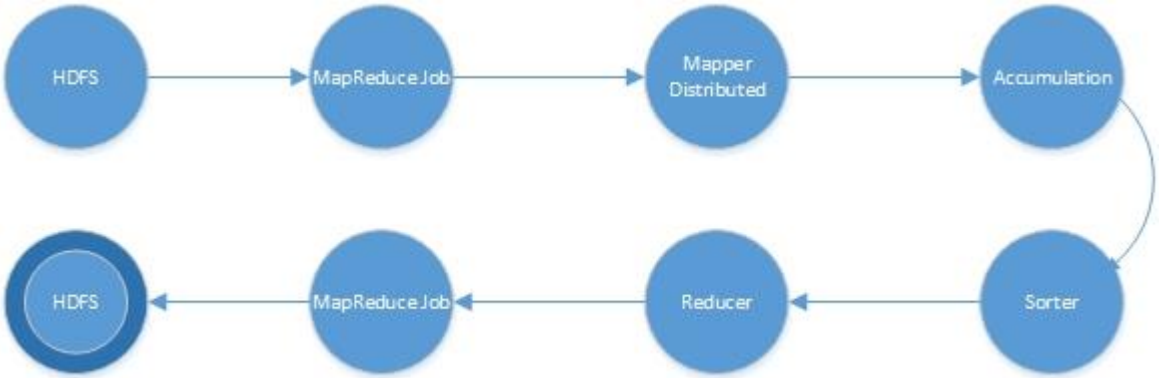


Fig. 2. Dataflow Diagram

4.0.3 System Architecture

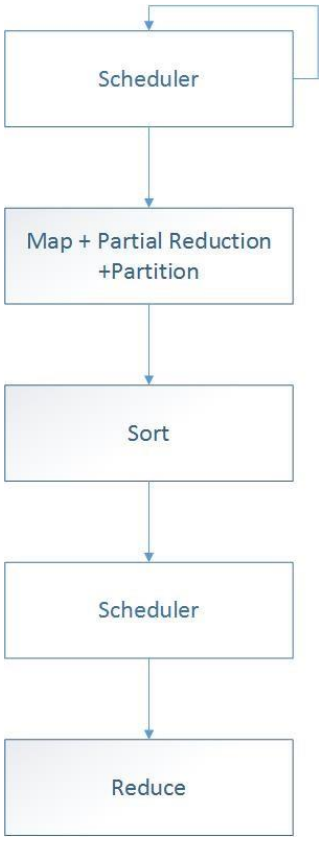


Fig. 3. Pipeline architecture for single node PICO

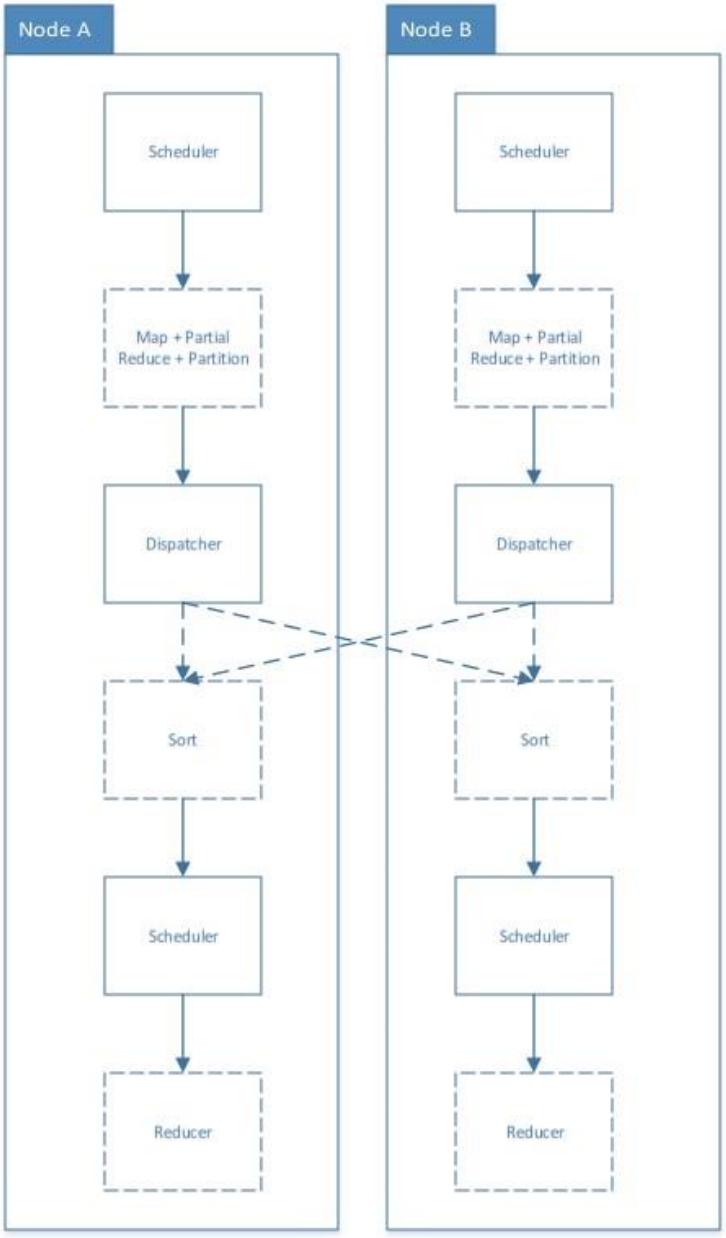


Fig. 4. Pipeline architecture for multi-node PICO

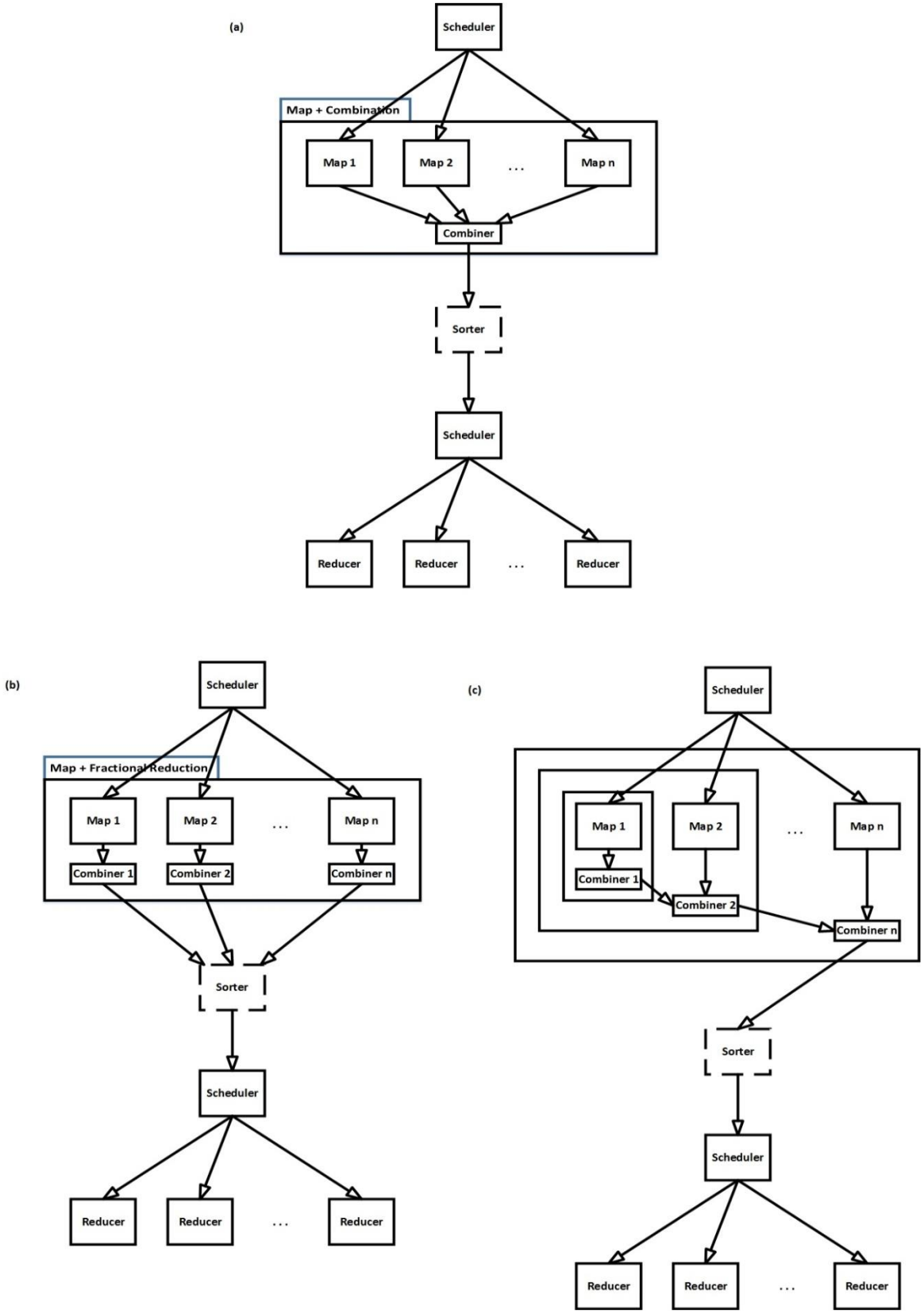


Fig. 5 a) Architecture of MapReduce Combine Phase. This optimization is a variant of an existing MapReduce task, Combine. **b)** Architecture of Fractional Reduction Optimization. **c)** Architecture of Amassing Optimization.

4.0.4 Sequence Diagram

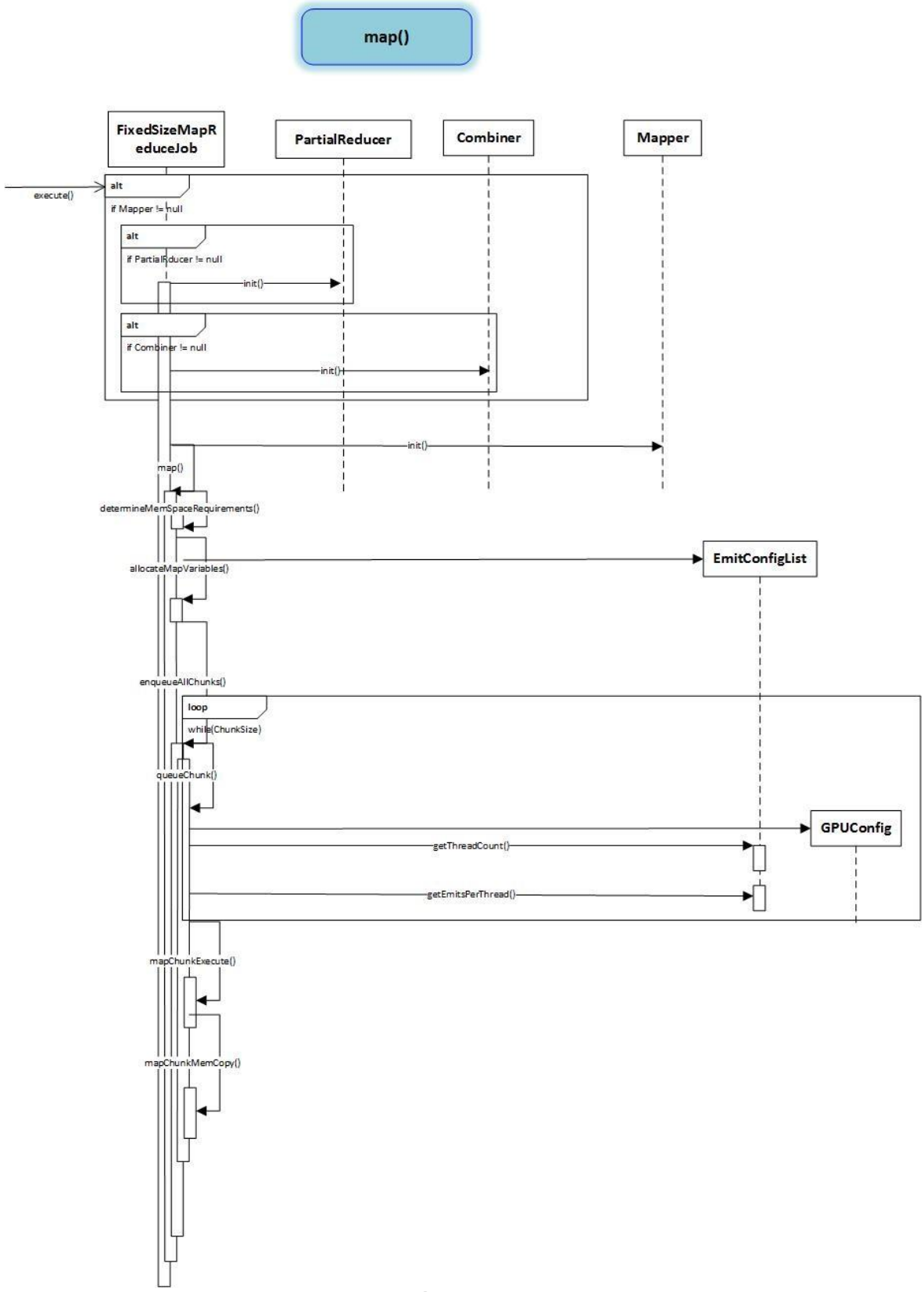


Fig. 6. Map

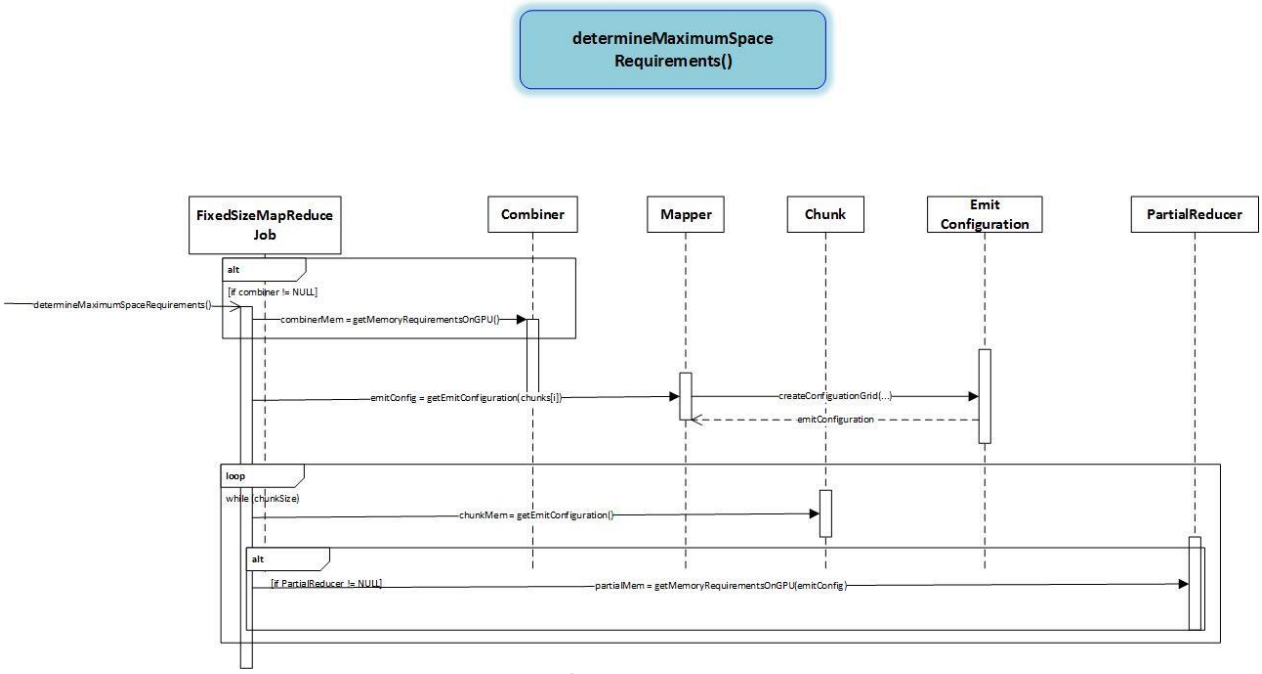


Fig. 6.1. Map

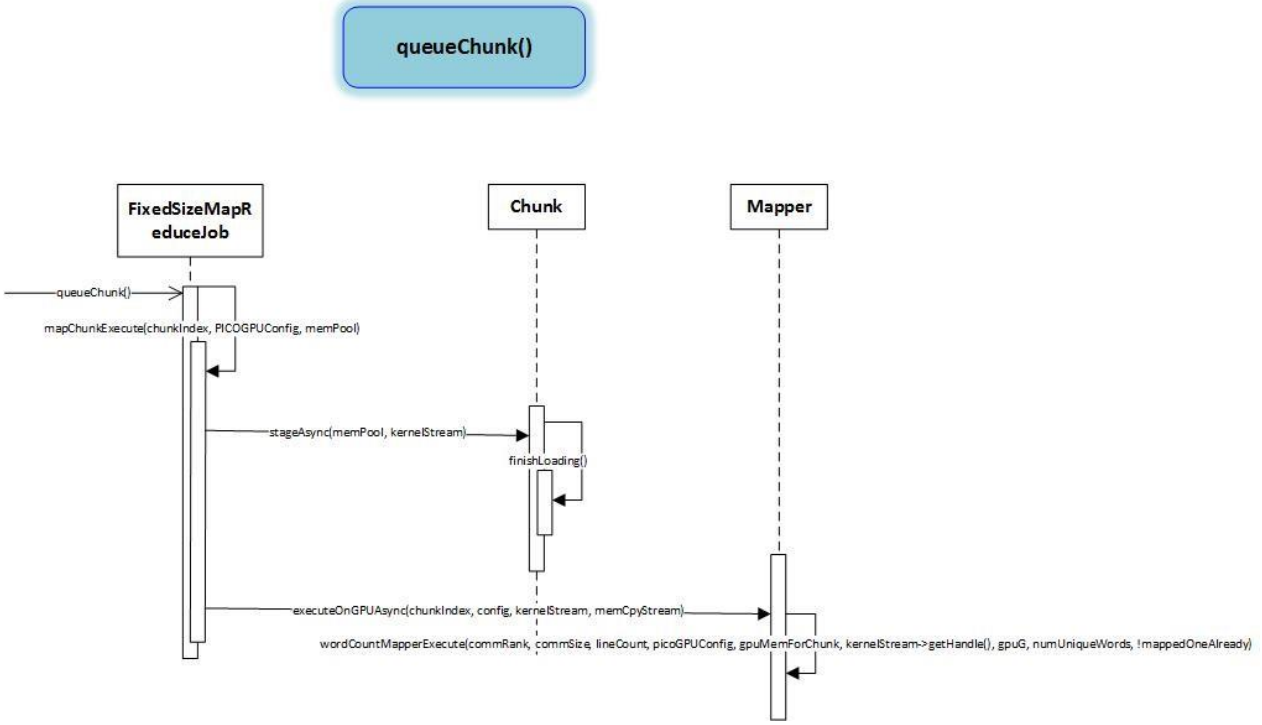


Fig. 6.2. Map

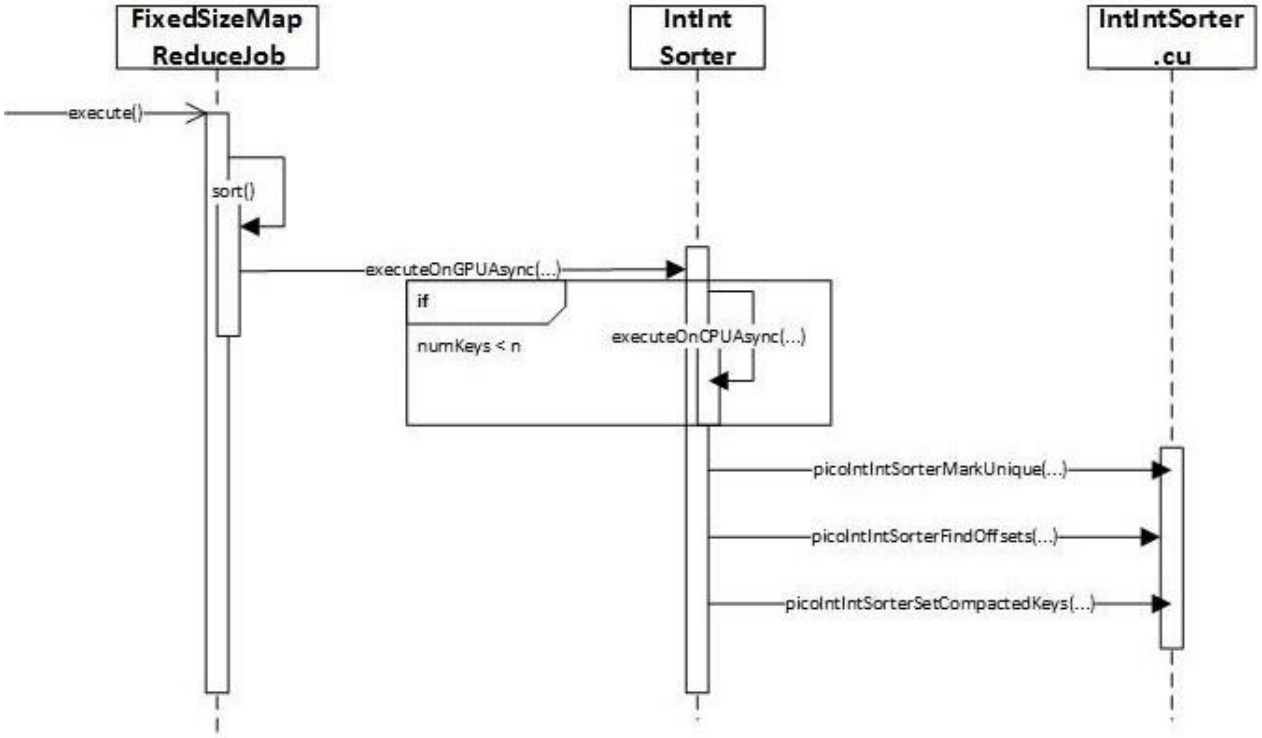

FixedSizeMap ReduceJob
IntInt Sorter
IntIntSorter .cu
execute()
sort()
executeOnGPUAsync(...)
if
numKeys < n
executeOnCPUAsync(...)
picoIntIntSorterMarkUnique(...)
picoIntIntSorterFindOffsets(...)
picoIntIntSorterSetCompactedKeys(...)

Fig. 7. Sort

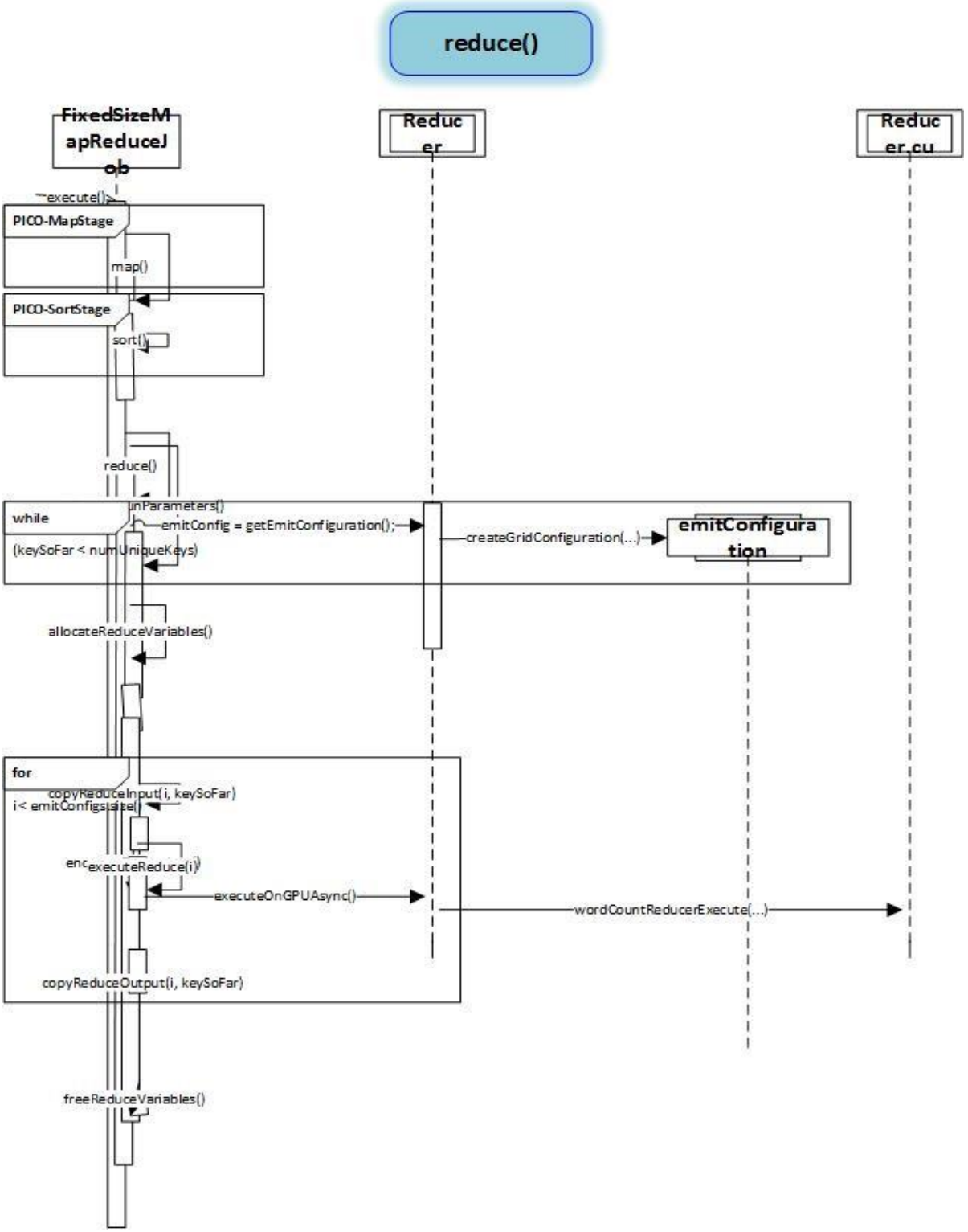


Fig. 8. Reducer

4.1 Problems encountered during the development

We are mentioning few of them here. First, the Jetson tk1 has been launched very recently and there is very little documentation available on this device. Also it has the latest version of CUDA installed, which is slightly different from the previous versions. Moreover, it is the first GPU with unified memory architecture. Secondly we will have to take care of synchronization overhead of our framework. It must be kept very low to enable the system to scale to hundreds of processors. Third, a very deeply planned and efficient load-balancing scheme should be implemented in the framework to utilize the huge thread parallelism of GPU. Finally, we will also have to handle the concurrent reads and writes, file manipulation and string processing very efficiently because all these tasks are unconventional to GPUs.

4.2 Description of the software deliverable

We intend to present our stand-alone Map-Reduce library that leverages the power of GPU for large-scale computing. The framework is responsible to distribute the tasks of map, group and reduce among the hundreds of cores of a GPU in parallel fashion and provide a wrapper for map and reduce to the application developer.

The library will be responsible for creating default number of thread groups and the number of threads within a thread group for map and reduce. However, we will provide configuration parameters to the application developer to override the default configuration. Our framework will handle the sorting of intermediate results as well as the merging of final key-value pairs. Moreover, the framework will also prevent the write conflicts during concurrent read-write operations.

5. Software Testing

5.1. Unit Testing

Definition:

In testing phase of PICO we tested each module as they were developed. Path coverage was used as coverage technique. The following units were tested:

- Mapper
- Sorter
- Partial Reducer
- Combiner
- Partitioner
- Reducer

Participants*:*

- Abdul Basit
- Muhammad Usman Irfan

Methodology:

- Created a Test Plan
- Created Test Cases and Test Data
- Created scripts to run test cases
- Fix the bugs if any and re tested the code
- Repeat the test cycle until the “unit” (module) is free of all bugs

5.2. Integration Testing

Definition:

Once unit tested components/modules are delivered we then integrate them together. These “integrated” components are tested to weed out errors and bugs caused due to the integration.

Participants

- Ahmad Rahman

Methodology:

- Created a Test Plan
- Created Test Cases and Test Data
- Created scripts to run test cases
- Once the components have been integrated executed the test cases
- Fix the bugs if any and re tested the code
- Repeat the test cycle until the components have been successfully integrated

6. Critical Evaluation

6.1. Limitations

One hardware feature that is limiting to PICO is the lack of any interplay between GPUs and network interconnects. We hope that GPU and network vendors work together to allow sourcing and sinking by the GPU for network I/O. This is possible as the PCI-e bus supports peer-to-peer communication, and PICO would benefit by moving intermediate data between nodes without having to route through CPU memory.

6.2. Resources

6.2.1. Human Resource

The project team comprises of three members:

- Ahmad Rahman
- Abdul Basit
- Muhammad Usman Irfan

6.2.2. Hardware Resources

The hardware resources we used in development are listed below:

- NVIDIA Jetson TK 1 (2)
- NVIDIA Tesla K20
- GeForce GTX Titan Black

6.2.3. Software Resources:

The Software resources we used in development are listed below:

- Ubuntu 14.04 L4T (Linux for Tegra)
- NVIDIA Nsight Eclipse Edition

7. Benchmarks Implementation

7.1. Matrix Multiplication

MM is the only application we chose that is, in itself, highly scalable on the GPU. We implemented a straightforward square matrix multiply in two phases. From the very beginning, we had to craft the algorithm carefully. The common CPU MapReduce MM algorithm for multiplying square matrices of dimension M uses M2 vector-vector multiplications in Map, one for each element of the result. There is no Sort or Reduce. This falls short on the GPU in two key ways. First, vector-vector multiply works well on a GPU only when reading row vectors of a matrix (coalescing rules). Second, the GPU has scratchpad memory but not nearly enough to contain both a row vector and a column vector if M is sufficiently large. Due to the limitations of the above algorithm, we used a hierarchical approach from typical cache-oblivious algorithms. We tile each matrix into smaller and smaller pieces until each block in the GPU can fit in shared memory. Then the entire block performs a small matrix multiplication where each thread computes an element of the result as an inner product. Each submatrix of the result is also a tile of inner products. We thus transform the original formulation into N3 uniform sized—and at least 10242—tile multiplications. Each Map tiles this small matrix into a set of 2563 matrices, which are further divided into 162 matrices. Each block performs

a full inner product of tiles by performing many 162 tile multiplications. We stop the division here because a block of 256 threads can read 162 values in a coalesced manner and perform enough computation to keep throughput on the GPU very high. To add all the partial sums of each submatrix and form the result, we bypass Sort and Reduce and implement another Map in a separate MapReduce2. The result is a very scalable, out-of-core implementation of MM that uses the GPU much more efficiently (and runs several orders of magnitude faster) than the direct port of the typical CPU implementation, even on small matrices. Another point that is well illustrated by our algorithm is that PICO allows for a natural use of the GPU by not forcing any specific thread-to-task mapping (e.g. one GPU thread per map task).

7.2. Word Count

WC counts all occurrences of a unique object. Also, the output set for WC is much smaller, leading to a different configuration of the pipeline and drastically different scaling. The input is a collection of text, with words taken from a pre-determined corpus, separated at line boundaries. For our test cases, we used randomly generated text from a forty-three-thousand-word dictionary. Each chunk contains millions of bytes. The typical CPU Mapper implementation has each Map task scan one line of text and emit hW,1i for every found word W. This does not work on a GPU. Sending one line of text to a Mapper is fine, but we should not use strings as keys; strings cannot be read in a single instruction as their length varies, and forcing all strings into a fixed-size area yields poor storage performance in many cases—storing a four-character word in sixteen bytes wastes 75% of the key space. Using variable-sized keys either requires more space for a key directory or more time as PICO must use atomics to emit keys. Instead, we used a minimal perfect hash to assign each key a unique, four-byte integer value. Thus, the GPU Map kernel gives each thread one line of text and scans the text for words, then finishes by hashing W and emitting hash(W),1i. As our dictionary is small (43k integer-integer pairs requires less than 350 kB), we chose to use Accumulation. An initial Map task emits all keys with the value 0. Afterward, whenever we emit a key-value pair, we simply index into the emit space and use a fire-and-forget atomic instruction to increment the associated value.

7.3. K-Means

KMC is used in machine learning. The basic task takes a set of points in space and determines clusters that can best approximate the space. For both the CPU and GPU benchmark, we use a fixed-size random set of cluster centers at job startup. The typical CPU implementation of the Map kernel reads one-point P, finds the index of the closest center C, and emits h index(C), Pi. We implemented this in PICO and saw poor results for three reasons: each thread loads its own point (thus not guaranteeing coalesced reads), we see far too many intermediate key-value pairs, and the size of the emitted pair causes us to issue non-coalesced writes. The biggest change we made was to use persistent threads within the Map stage to process many elements per thread and to use atomic-free Accumulation. The entire block reads points in a coalesced manner and each thread finds the closest center C. The block performs a series of reductions of all points belonging to C. As each reduction completes, the block's master thread accumulates the reduction value to global memory. Because the GPUs we used do not have floating-point atomics, we were forced to use a per-block global-memory pool. After the primary kernel completes, another kernel reduces the values in each block's pool and emits the final values. The CPU implementation emits the h index (cluster center), point i. The GPU emits h C, P dim i for each dimension, as well as an extra key per center (the number of influencing points). This allows for coalesced writes with negligible overhead for a small number of dimensions. If the number of dimensions and centers are large, the typical CPU approach may prove more efficient as it uses less storage.

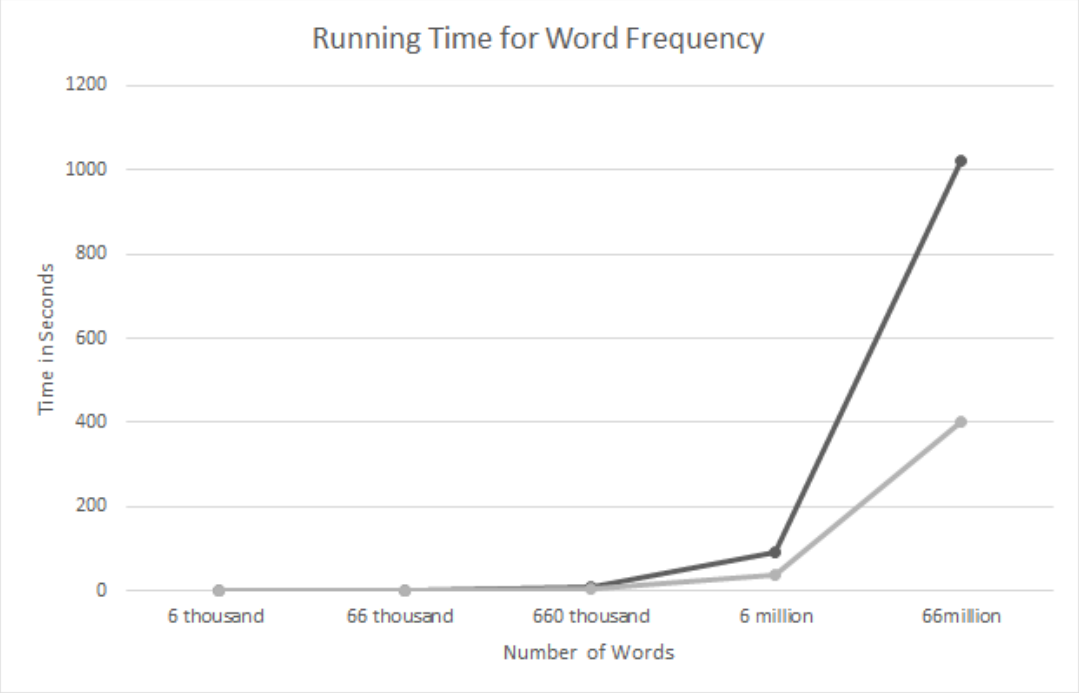
7.4. Linear Regression

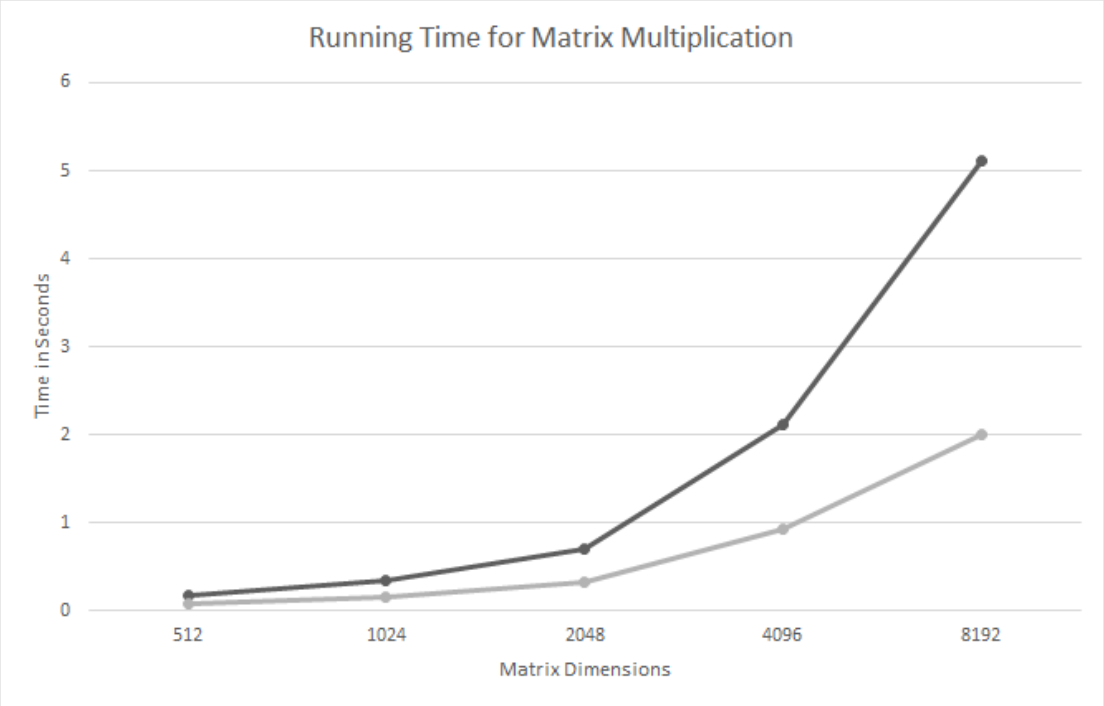
LR models the relationship between two parameters influenced by unknown variables. We store chunks in much the same way as KMC, grouping many points together and tightly-packing them in arrays. In fact, LR is similar to KMC in many ways and the same optimizations work well. We use persistent threads to compute the relationship as well as our own internal Accumulation. As with KMC, we achieve an almost order-of-magnitude speedup over a direct port of the typical CPU implementation that we modeled after the straightforward CPU implementation. The Mapper emits

only six keys upon completion, and thus we do not use Partitioning (the network overhead is minimal in both cases). We use the default sort and perform reductions in a key-per-thread manner (reduction time is virtually nil).

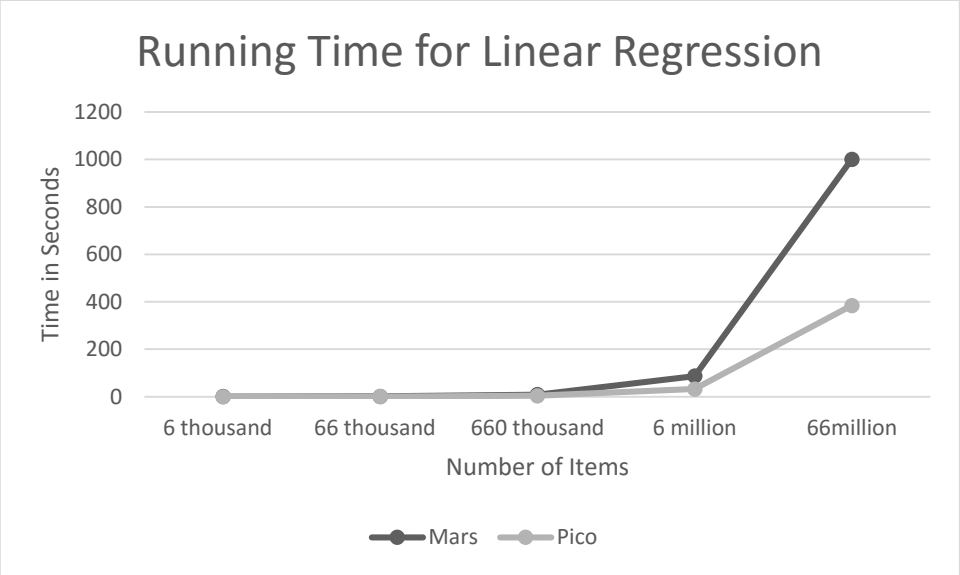
8. Results

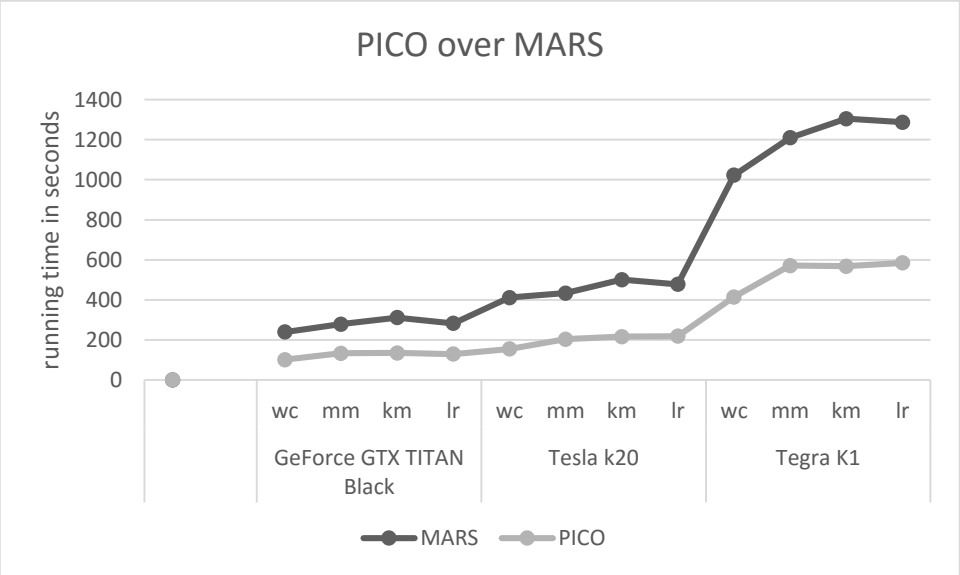
Using PICO, we have written four benchmark applications, the word count or word frequency, Matrix Multiplication, k-Means Clustering and Linear Regression. We have used different subsets of overly simplified data set from Reuter's data set for these benchmarks evaluation. For comparison with PICO, we use the best known GPU-MapReduce framework Mars [2]




Running Time for Matrix Multiplication
Time in Seconds
0
1
2
3
4
5
6
512
1024
2048
4096
8192
Matrix Dimensions


Running Time for k-means Clustering
Time in Seconds
0
100
200
300
400
500
600
700
800
900
6 thousand
66 thousand
660 thousand
6 million
66million
Number of Items


Running Time for Linear Regression
Time in Seconds
1200
1000
800
600
400
200
0
6 thousand
66 thousand
660 thousand
6 million
66million
Number of Items
Mars
Pico


PICO over MARS
running time in seconds
1400
1200
1000
800
600
400
200
0
wc mm km lr wc mm km lr wc mm km lr
GeForce GTX TITAN Black
Tesla k20
Tegra K1
MARS
PICO

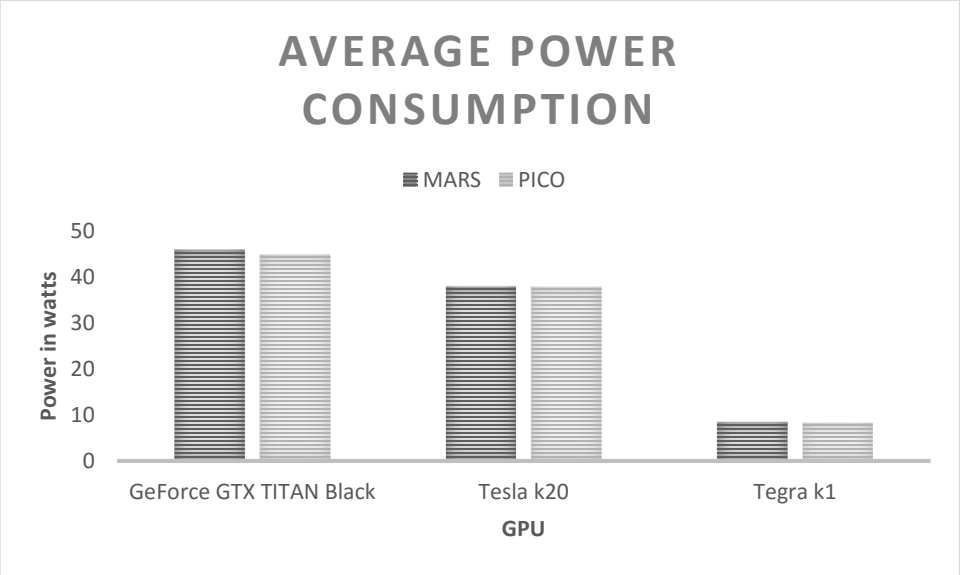


Table II below, shows the comparative analysis of execution times and power consumption on three different GPU architectures (Single Node) using four different benchmark applications. Word Count, Matrix Multiplication, K-Means Clustering and Linear Regression. The data set used for the analysis in the table is **a)** Word Count - 660 Million words. **b)** Matrix Multiplication – dimensions: 4096x4096. **c)** k-Means **-** Google Web Graph, 10 Iterations.

		MARS		PICO	
		Execution Time(sec)	Power - peak (watts)	Execution Time(sec)	Power - peak (watts)
GeForce GTX TITAN Black	wc	240	45.1	102	43.6
	mm	279	48.7	134	45.32
	km	312	46.02	135	46.3
	lr	283	45.3	130	44.34
Tesla k20	wc	412	38.4	155	38.7
	mm	434	39.2	204	37.78
	km	501	37.1	217	37.2
	lr	478	36.8	219	37.07
Tegra K1	wc	1023	8.1	415	8.3
	mm	1210	8.4	571	7.4
	km	1305	8.7	568	8.1
	lr	1287	8.5	585	7.9

Table II. Execution Times and Power Consumption Comparison between MARS and PICO

The results show that PICO has an average speed up of up to 2.275 over Mars with same configuration and datasets for the four benchmarks.

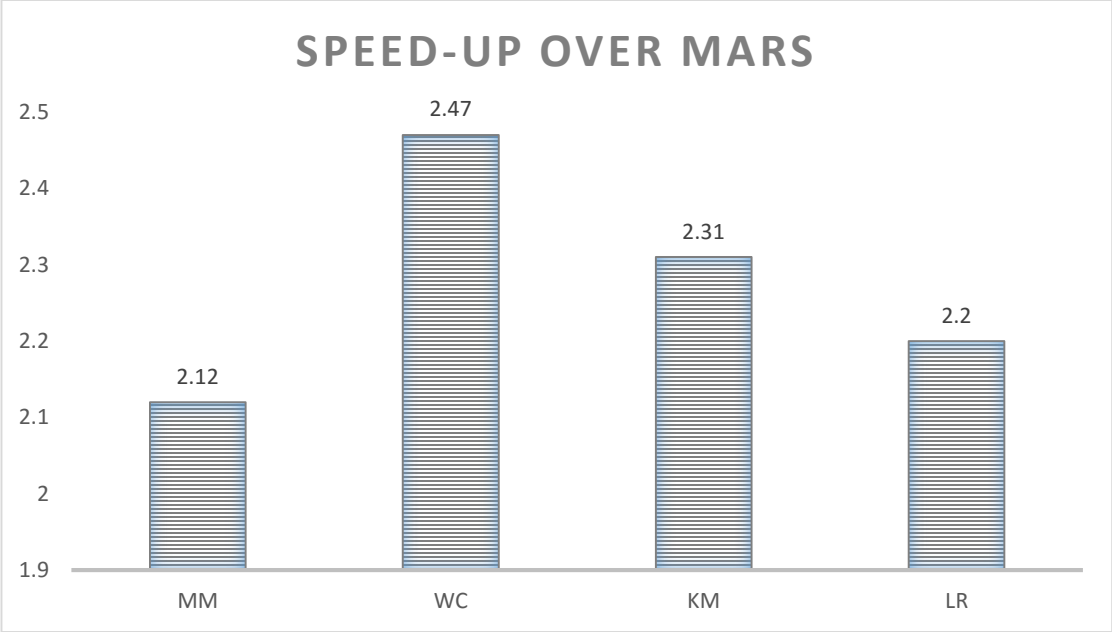


Fig. 9. Speed-up Over MARS

9. References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, 2004.

[2] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. SIGMOD, 2007.

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, Volume 26, 2007.

[4] A. Ailamaki, N. Govindaraju, S. Harizopoulos and D. Manocha. Query co-processing on commodity processors. VLDB, 2006.

[5] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," 2009, http://gpgpu.org/developer/cudpp/.

[6] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in GPU Gems 3, H. Nguyen, Ed. Addison Wesley, Aug. 2007, ch. 39, pp. 851–876.

[7] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.

[8] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multicoreandmultiprocessorsystems,"inInternationalSymposium on High-Performance Computer Architecture. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 13–24.

[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Oct. 2008, pp. 260–269.

[10] J. Ekanayake and G. Fox, "High performance parallel computing with clouds and cloud technologies," in Proceedings of the First International Conference on Cloud Computing, Oct. 2009.

[11] B. Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce framework for programming graphics processors," in Third Workshop on Software Tools for MultiCore Systems, Apr. 2008.

[12] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing parallel program portable between CPU and GPU," in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2010, pp. 217–226.

[13] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "CellMR: A framework for supporting MapReduce on asymmetric Cell-based clusters," in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.

[14] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: Simplified relational data processing on large clusters," in SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Jun. 2007, pp. 1029– 1040.

[15] C. Jin and R. Buyya, “MapReduce programming model for .NET-based cloud computing,” in Euro-Par ’09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Berlin: Springer-Verlag, Aug. 2009, pp. 417–428.

[16] S. Chen and S. W. Schlosser, “Map-Reduce meets wider varieties of applications,” Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05, May 2008. [Online]. Available: http://www.pittsburgh. intel-research.net/~chensm/papers/IRP-TR-08-05.pdf

[17] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, “Map-Reduce as a programming model for custom computing machines,” in FCCM ’08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines, Apr. 2008, pp. 149–159.

[18] R. Lämmel, “Google’s MapReduce programming model—revisited,” Science of Computer Programming, vol. 68, no. 3, pp. 208–237, Oct. 2007.

[19] Y. Gu and R. L. Grossman, “Sector and sphere: the design and implementation of a high-performance data cloud.” Philosophical transactions. Series A, Mathematical, physical, and engineering sciences, vol. 367, no. 1897, pp. 2429–2445, Jun. 2009. [Online]. Available: 10.1098/rsta.2009.0053

[20] H. Prokop, “Cache-oblivious algorithms,” Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Jun. 1999.

[21] R. J. Cichelli, “Minimal perfect hash functions made simple,” Communications of the ACM, vol. 23, pp. 17– 19, January 1980.

[22] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on GPUs,” in Proceedings of High Performance Graphics 2009, Aug. 2009, pp. 145–149.