Department of Computer Science
National University of Computer & Emerging Sciences

Super Computing on the go: MapReduce on mobile GPU cluster

PICO

Final Year Project Report

Submitted By

Ahmad Rahman
K12-2026

Muhammad Usman Irfan
K12-2023

Abdul Basit
K12-2290

Supervisor
Dr. Jawwad Shamsi

Contents

1 Introduction and Motivation

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.

MapReduce is the most successful modal of processing big data [1][2]. MapReduce frameworks for CPUs are proved to be very successful. But currently GPUs are taking limelight from CPUs due to their highly parallel abilities, in domains such as high performance computing [4]. But programming an application on GPU is very difficult because it's necessary to have complete details of GPU before writing a code for it. Here comes the role of our framework. Our framework will provide a simple and familiar MapReduce interface that will help the developers to easily write programs on the GPU using very common programming languages C/C++, hiding the complexity of GPU from them. This will not only help the developers to easily exploit the power of GPU but will also open GPU computing to larger application domains.

Once over coming all challenges and successfully developing our MapReduce framework, we can provide a mobile cluster of 7 GPUS (Jetson TK1) along with its charged power source, containing almost 1400 cores and processing terabytes of data within just few minutes. It will be the first of its kind and will revolutionize the traditional method of processing big data with commodity clusters and Hadoop running on them. Due to its mobility, high parallelism among the cluster nodes and within a node among its hundreds of cores and high memory bandwidth it will quickly dominate the stationary commodity clusters. Researchers and scientists and even business analysts will be able to use a mobile cluster and can easily carry it where so ever. On the other hand, its easy and familiar MapReduce framework will push the developers to exploit the power of GPUs and build extensive data processing applications, such as computer vision, image processing, and business data analysis or web search applications on GPU.

2 Background and Related Work

In this section, we introduce the GPU architecture and discuss the related work on the GPU, and review the MapReduce framework.

Graphics Processors (GPUs)

GPUs are widely available as commodity components in modern machines. Their high computation power and rapidly increasing programmability make them an attractive platform for general-purpose computation. Currently, they are used as co-processors for the CPU [1].

The programming languages include graphics APIs such as OpenGL [21] and DirectX [4], and GPGPU languages such as NVIDIA CUDA [20], AMD CTM [2], Brook [5] and Accelerator [26]. With these APIs, programmers write two kinds of code, the kernel code and the host code. The kernel code is executed in parallel on the GPU. The host code running on the CPU controls the data transfer between the GPU and the main memory, and starts kernels on the GPU. Current co-processing frameworks provide the flexibility of executing the host code and other CPU processes simultaneously. It is desirable to schedule the tasks between the CPU and the GPU to fully exploit their computation power.

The GPU architecture model is illustrated in Figure 1. The GPU consists of many SIMD multi-processors and a large amount of device memory. At any given clock cycle, each processor of a multiprocessor executes the same instruction, but operates on different data. The device memory has high bandwidth and high access latency. For example, the device memory of the NVIDIA G80 has a bandwidth of 86 GB/sec and latency of around 200 cycles. Moreover, the GPU device memory communicates with the main memory, and cannot directly perform the disk I/O. The threads on each multiprocessor are organized into thread groups. The thread groups are dynamically scheduled on the multiprocessors. Threads within a thread group share the computation resources such as registers on a multiprocessor. A thread group is divided into multiple schedule units. Given a kernel program, the occupancy of the GPU is the ratio of active schedule units to the maximum number of schedule units supported on the GPU. A higher occupancy indicates that the computation resources of the GPU are better utilized. The GPU thread is different from the CPU thread. It has low context-switch and low creation time as compared to CPU's. The GPU has a hardware feature called coalesced access to exploit the spatial locality of memory accesses among threads. When the addresses of the memory accesses of the multiple threads in a thread group are consecutive, these memory accesses are grouped into one.

GPGPU (General-Purpose computing on GPUs)

GPUs have been recently used for various applications such as matrix operations [18], FFT computation [17], embedded system design [10], bioinformatics [19] and database applications [13][14]. For additional information on the state-of-the-art GPGPU techniques, we refer the reader to a recent survey by Owens et al. [22]. Most of the existing work adapts or redesigns a specific algorithm on the GPU. In contrast, we focus on developing a generic framework for the ease of the GPU programming. 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 and so on. 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 and so on. 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. Different from the existing GPGPU computation, this study aims at developing a general framework that hides the GPU details from developers so that they can develop their applications correctly, efficiently and easily. Meanwhile, our framework provides the runtime mechanism for the high parallelism and the memory optimizations for a high performance.

PICO also leverages the "CUDA Data-Parallel Primitives" library [5], specifically its scan [6] and sort [7] primitives. Using CUDPP, we were able to design PICO more easily and focus on the API since CUDPP handles many of the difficult aspects.

MapReduce

MapReduce is a programming model that combines two separate higher-order functions in functional programming languages, map and reduce (also known as fold). Programmers specify map and reduce functions. The input to map is a set of data items; each map invocation outputs a sequence of independent key value pairs. All like-keyed values are grouped together and passed to a reduce function which processes them to output a sequence of new values. Google's MapReduce implementation was built for large-scale data processing; large data sets are input to many mapping nodes. Intermediate data is streamed back out to buffer cache, partitioned, and sent to processing nodes to be reduced. Google uses MapReduce for much of its internal data processing, and because of its success Google popularized MapReduce with industry and academia. Since that time, developers and researchers have created many MapReduce packages.

i-MapReduce [10] (previously CGL MapReduce) is another MapReduce package, and potentially the first to use data streams instead of hard disk access. Intermediate data values and reduction results are streamed directly to new mapper and reducer nodes for further processing. This allows for an efficient, iterative MapReduce algorithm with many consecutive MapReduce processes. Recent efforts have gone towards porting MapReduce to parallel processors like GPUs and IBM's Cell. Catanzaro et al. created a single-node library for GPUs [11] but the focus was on many small tasks; the main contribution was creating an efficient small sequence sort on the GPU. Mars was the first largescale GPU system, though its scalability is limited; it uses only one GPU and in-GPU-core tasks. Another shortcoming is that the library, not the user, schedules threads and blocks, making it hard to fully exploit certain GPU capabilities (e.g. inter-block communication). MapCG [12] is another GPU-based MapReduce library. The

main goal was to allow for portable multicore MapReduce code that could run on the GPU. Like Mars, MapCG only offers limited scalability as it uses but one GPU.

CellMR [13] is a single-node implementation of MapReduce on the Cell Engine that alleviates the in-core dilemma of Mars by streaming map data in small pieces. CellMR divides the traditional mapreduce pipeline into three successive steps: map, partial reduction that is on still-resident key-value pairs, and global reduction.

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).

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.

Moving from the CPU to the GPU

The simple in-core, single-node CPU MapReduce implementation is easy. Take a set of indivisible work units (items) and Map them, generating key-value pairs. Sort these pairs by key and Reduce all like keyed pairs and gather the final output. Translating this model to the GPU is straightforward: copy the input data to the GPU; make Map, Sort, and Reduce kernel calls; and copy the output data back to the CPU. Each data item in Map and Reduce is assigned to one GPU thread, and we process many items (a “Chunk”) with a single kernel call. This model is similar to the Mars GPU MapReduce implementation [9], but it ignores two practical scalability problems for GPUs: what happens when the size of the data set exceeds in-core memory? And how do we scale across nodes? Map and Reduce should be distributed across nodes, but then each Map output could potentially feed any/all Reduce instances. Thus we need to Partition our Map output. Both the partitioning and communication implementations must be efficient, but previous GPU MapReduce implementations address neither of these.

We implement several changes and optimizations to this model to build PICO. The first to only expose partitioning and sorting to the programmer (similar to Hadoop) in a manner that allows programmers to optimize for particular workloads and leverage the GPU with minimal PCI-e overhead when desired. For all tasks, we relax the mapping constraint of one item, one-thread. This yields a more flexible mapping; a many-to-one or one-to-many mapping of items to threads might be desirable and more efficient than a one-to-one mapping. We also note that many GPU algorithms use inter-thread cooperation, and relaxing this mapping allows for such. For instance, many GPU computing applications use reductions or parallel prefix operations. Our relaxed mapping gives more flexibility to the user and more efficient higher-level operations.

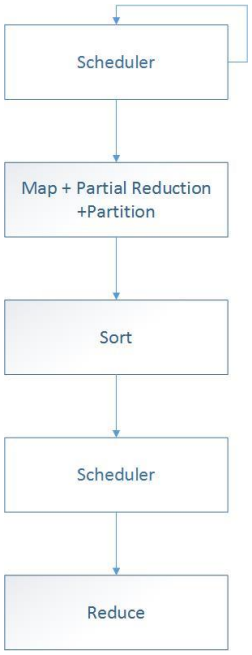

Fig. 1. Pipeline architecture for single node Pico

Processing items in chunks yields additional benefits. Each GPU thread knows the item on which every other GPU thread is working, which allows for more natural programming on the GPU and block-level communication within a chunk. One of the contentious points with chunking is the tradeoff between abstraction and performance. Sticking to the design principles of PICO, we want to give full access to the GPU (i.e. allow block-wide reductions, manually schedule threads per block, etc.) to make PICO as fast and efficient as possible, so chunking simply makes sense.

We cannot speak for every developer, but we believe that chunking provides a good level of abstraction from the original data. And since chunking also allows developers full access to the GPU, they can leverage higher-level GPU operations. We now turn to optimization opportunities from restructuring the pipeline. Any experienced parallel programmer would say that the key to parallel efficiency is to reduce communication times as much as possible and to overlap communication with computation. All of our optimizations focus on reducing communication times at the expense of more computation time. We believe that communication is almost always the bottleneck and GPU computation is relatively cheap.

Each optimization essentially reconfigures the MapReduce pipeline. The first optimization is a variation on an existing optimization in MapReduce, Combine. Before the library transmits pairs to Reducers, like keyed pairs can be Combined, which does not emit any new keys but instead generates a single value to be associated with a key, ensuring that the node only sends one value per key to a Reducer. This is not a new optimization, but it requires a GPU implementation. PICO must use an efficient storage strategy for pre-emitted key-value pairs to stream them back down to the GPU for combination. Unlike in Hadoop, Combine happens only when all Maps complete in order to minimize network traffic as much as possible. This differs from the Hadoop Combine in that it typically only combines values from the same Map instance. The purpose of the PICO Combine sub stage is to reduce network traffic at the expense of added PCI-e transfer time and GPU computation time.

Using chunks allows for two more optimizations: Partial Reduction (similar to what is found in CellMR [13]) and Accumulation. These two stages are similar but differ in a few key ways. They are also mutually exclusive (at most one can be used).

Partial Reduction minimizes communication costs between the CPU and the GPU. As key-value pairs are emitted, PICO stores them on the GPU. Once a mapper finishes with a chunk, the library transfers all emitted key-value pairs from GPU memory back to system memory. To mitigate this cost, the user can execute Partial Reductions on GPU-resident key-value pairs to combine like-keyed pairs, resulting in fewer pairs and reduced transfer cost. The primary purpose of Partial Reduction is to reduce PCI-e communication and network-transfer time at the expense of more GPU computation.

Accumulation is similar to Partial Reduction in that the goal is to reduce the number of key-value pairs and transfer costs, but it is not a logically separate stage of the pipeline. It works by giving each mapper explicit knowledge of the key-value pairs resident on the GPU. When the library Maps the first chunk, it generates an initial set of key-value pairs that remain resident on the GPU, and each subsequent Map kernel combines its output with those pairs. Any combination of the following is done with Accumulation: adding new pairs, reducing the number of pairs, and combining pairs (the overall number of pairs remains the same). Accumulation should result in the final number of key value pairs being lower than if no accumulation were used. The main purpose of using accumulation is to reduce both PCI-e and network costs.

As a general rule, Partial Reduction is more useful when the expected final key-value set is large, and Accumulation should be used when the expected final key-value set is small. Partial Reduction yields gains when reducing like-keyed pairs is faster than transferring them; the user can overlap sending those key-value pairs to reducers with executing more maps. Accumulation works well when the number of final keys is much lower than the number of emitted key value pairs and the user can quickly index keys in the output.

3 METHODOLOGY OR DESIGN OR IMPLEMENTATION

We wrote PICO in C++ and CUDA and designed it to be easy-to-use and extensible while still allowing full access to the GPU. Every part of the MapReduce pipeline is programmable by the user, though we provide default implementations of the Scheduler, and Sorter. Each GPU is controlled by a separate process and each process executes the MapReduce pipeline. The three primary stages to the MapReduce work flow are Map, Sort, and Reduce. Map is divided into many separate sub stages, while Sort and Reduce are each indivisible. All stages and sub stages are customizable by the user. Figure 1 shows a diagram of a typical PICO work flow.

3.1. Map Stage

The Map stage is broken into several sub-stages; the Map itself, Accumulation, Partial Reduction, Combination. Depending on which of these sub stages (described earlier) the user activates, this stage can take many forms. Map processes an input chunk and outputs a set of key-value pairs. The entire chunk is copied to the GPU via a user-supplied function (typically a wrapper for cudaMemcpyAsync) at once and processed by one or more user-supplied kernels. This model allows the raw use of the GPU

while still maintaining the MapReduce model of data-independent elements. Map works on one chunk at a time as PICO assumes each chunk and its output will consume most of the GPU memory.

Chunking also gives us an efficient, out-of-core technique for GPU MapReduce. We can use chunks that are a fraction of the size of available memory, allowing us to Map or Reduce a chunk while simultaneously streaming another chunk to or from the GPU. One particular facet of the Map stage (and the Reduce) is the need for load balancing. PICO tracks the per GPU work in a dynamic queue. If one GPU finishes its work in its local queue and other GPUs have much more work to do, we shift chunks between the local queues. This is important as due to this requirement, chunks must implement a serialization method.

We summarize the effects of the most common pipeline configurations. Using Accumulation eliminates the need for Partial Reduce and Combine. Mapping without Accumulation or Combination, and optionally with Partial Reduce, causes partitioning after every Map completes. Using Accumulation or Combination causes the library to execute only one partition per full Map stage. This happens after the Combine sub-stage/final Accumulation.

3.2. Sort Stage

Sort is relatively straightforward. When possible (with keys that are integer-based), we used radix sort from CUDPP (PICO's default Sorter), and when not, we implemented our own. After the pairs are sorted, PICO discards duplicate keys. Because of the sort, each key's value is stored contiguously. Hence, we only need the number of values and the index of the first value to describe each sequence. We implemented all this functionality on the GPU to be as fast as possible.

3.3. Reduce Stage

Reduce is also relatively straightforward. Key-value sets are divided in a user-driven manner into chunks of their own, such that each chunk fits in GPU memory. PICO does this by issuing a callback to the Reducer that asks how many value sets should PICO copy to the GPU for the next reduction. PICO issues these callbacks until it processes the last sequence of values.

3.4. Mapping Applications to PICO

The most common use case of PICO is the same as that of a CPU-based MapReduce library. The user simply implements a Mapper and optionally a Reducer, and supplies input data. PICO contains default implementations of Sorters. PICO runs all of these on GPUs.

For performance and scalability, however, leveraging the various configurations of the PICO pipeline is vital. Beyond the default versions, users can specify their own implementation of a Sorter, customized for their application; Partial Reducers, and Accumulation can also yield large performance dividends.

Users should use these additional sub-stages, and they should also tune their pipeline and kernels to be as GPU-compute-bound as possible. Minimizing the fraction of communication-only runtime is vital for scalability; compute-bound jobs are scalable because much of their time is computation and overlaps with communication, resulting in high scalability. Real world MapReduce tasks span the gamut from compute bound to communication-bound; PICO gives strong scalability on the former and acceptable scalability on the latter.

3.5. Class Diagram

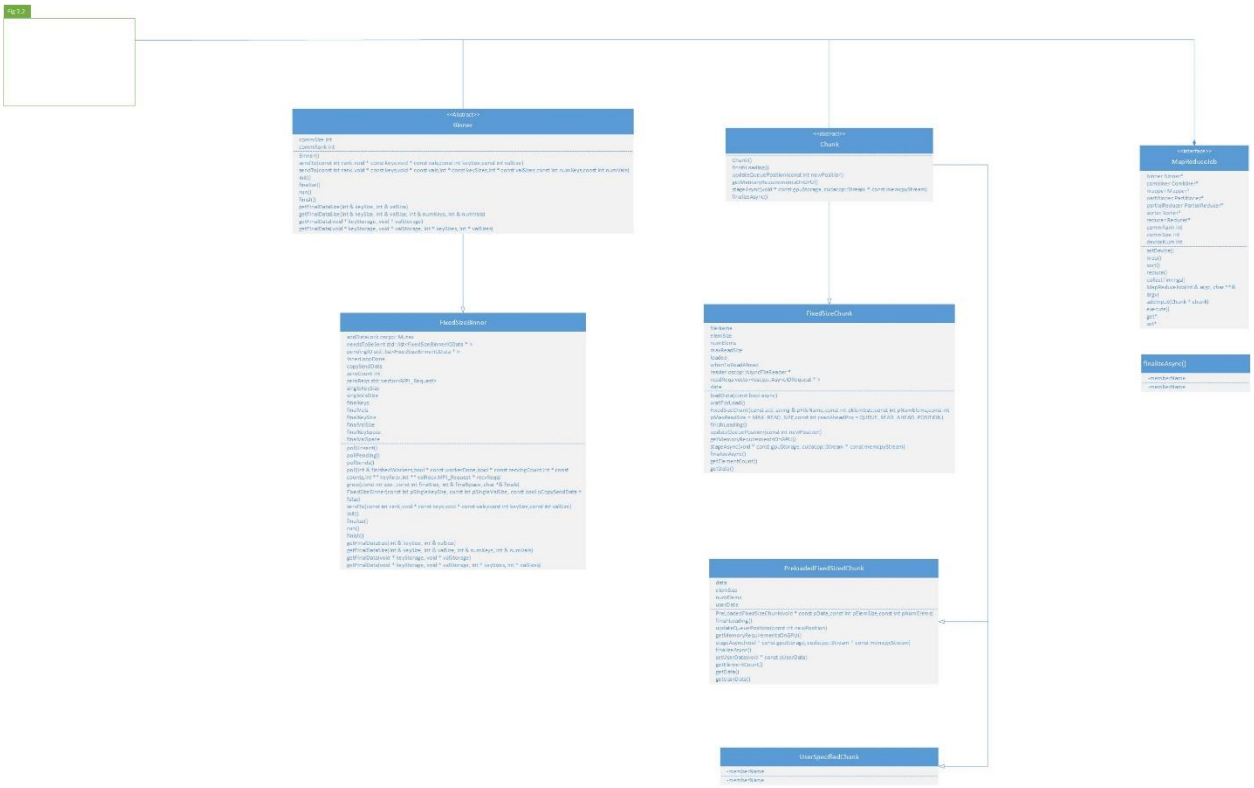

Fig. 2.1 Class diagram

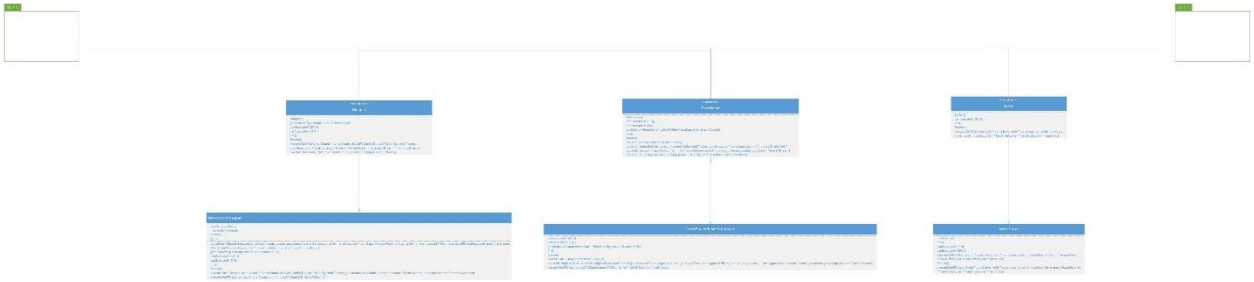

Fig. 2.2 Class diagram

Fig. 2.3 Class diagram

3.5. Sequence Diagrams

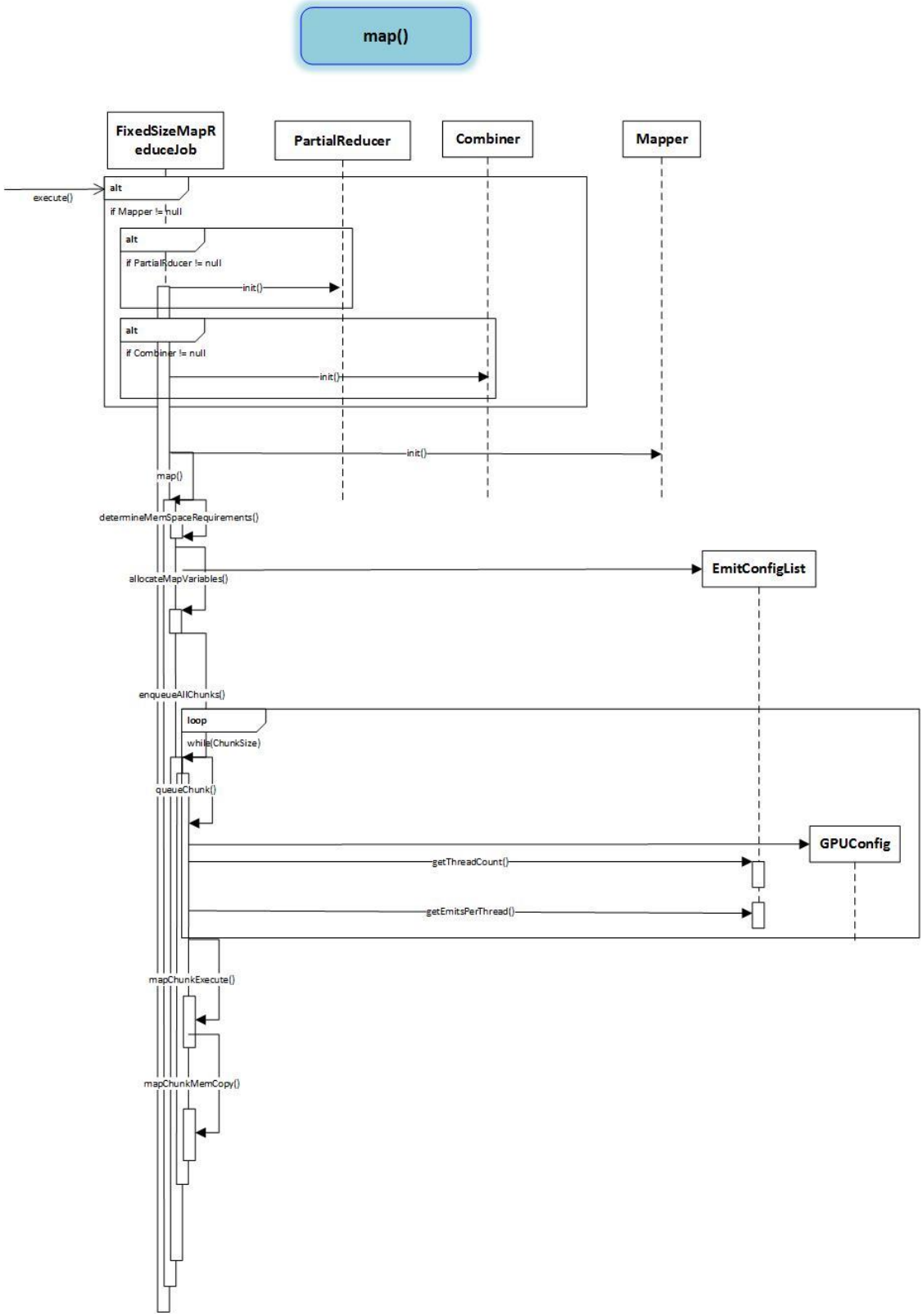

Fig. 3.1 Map

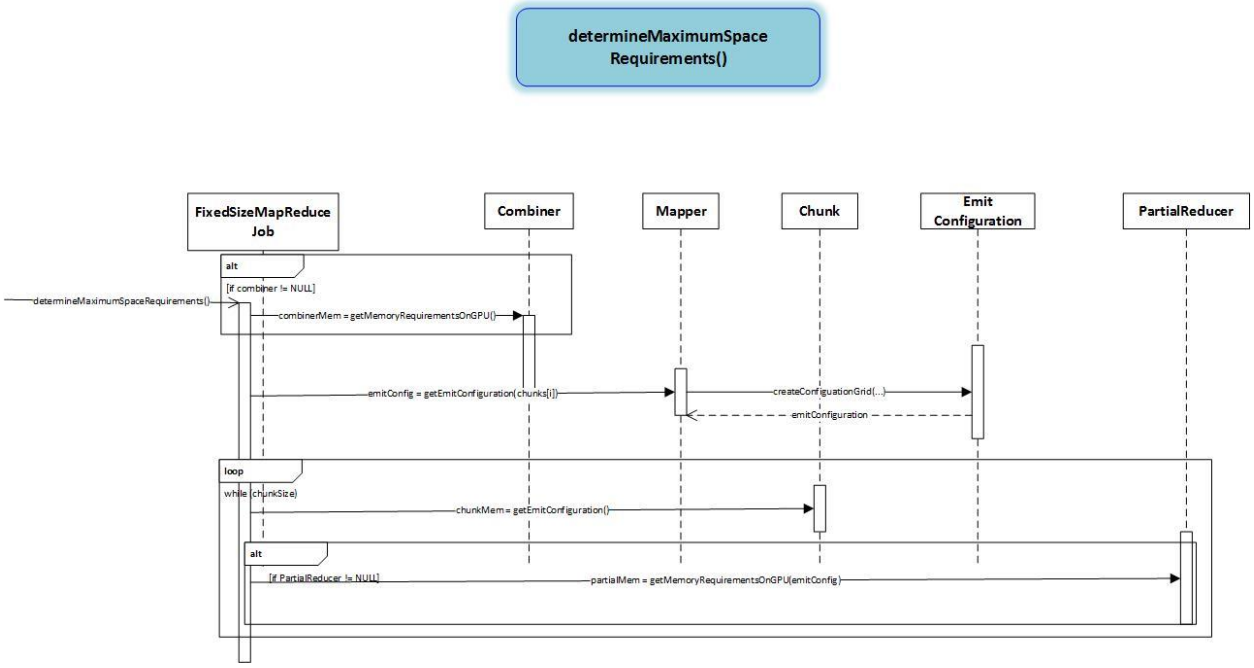

Fig. 3.2 Map

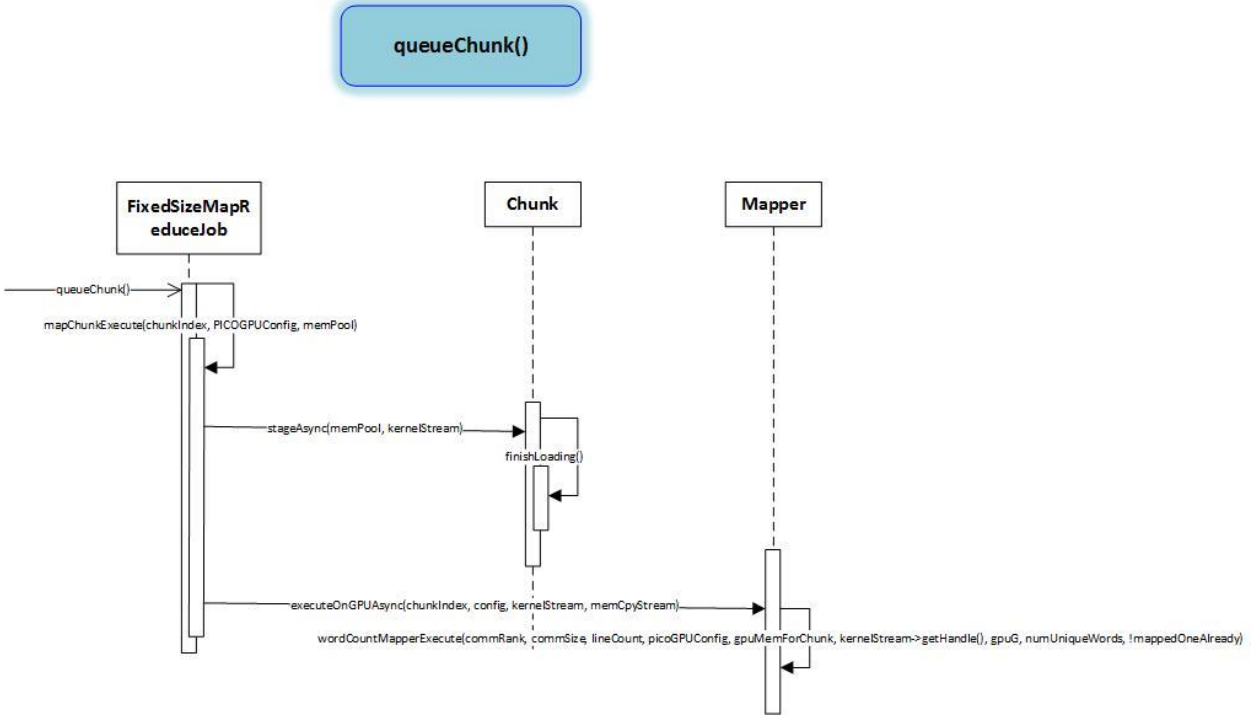

Fig. 3.3 Map

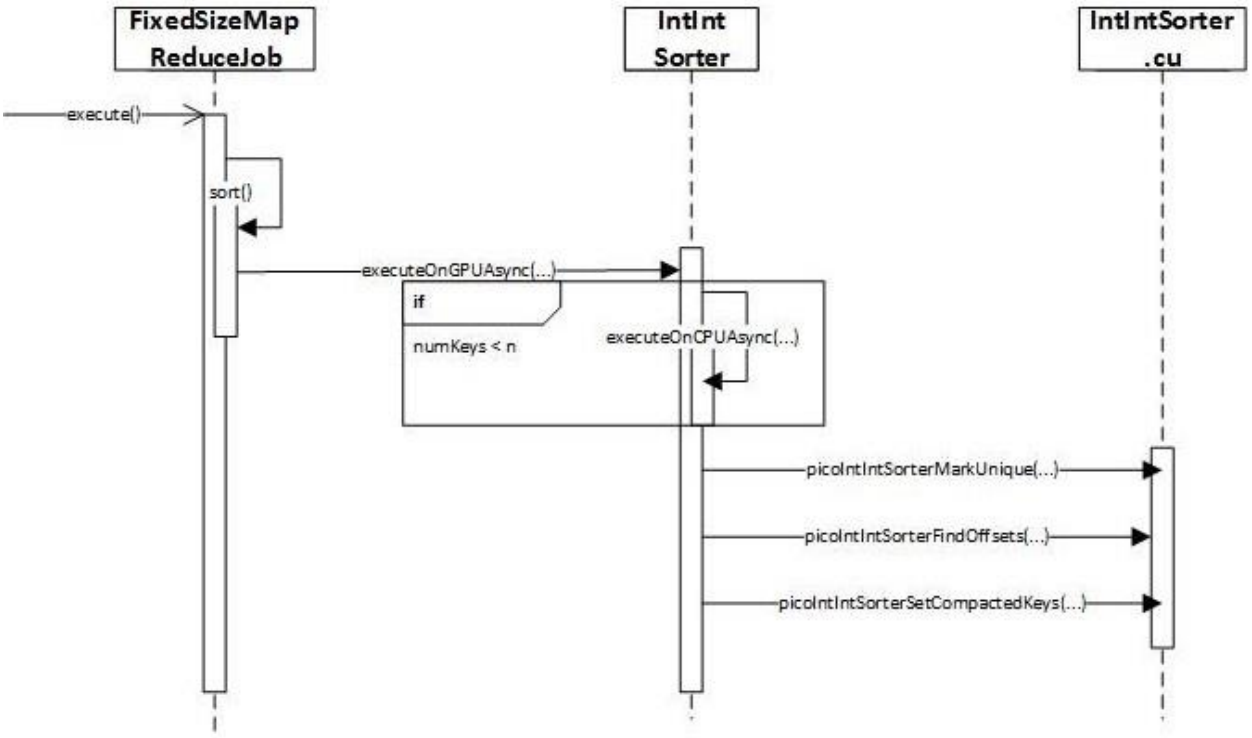
FixedSizeMap ReduceJob
IntInt Sorter
IntIntSorter .cu
execute()
sort()
executeOnGPUAsync(...)
if
numKeys < n
executeOnCPUAsync(...)
picoIntIntSorterMarkUnique(...)
picoIntIntSorterFindOffsets(...)
picoIntIntSorterSetCompactedKeys(...)

Fig. 4. Sort

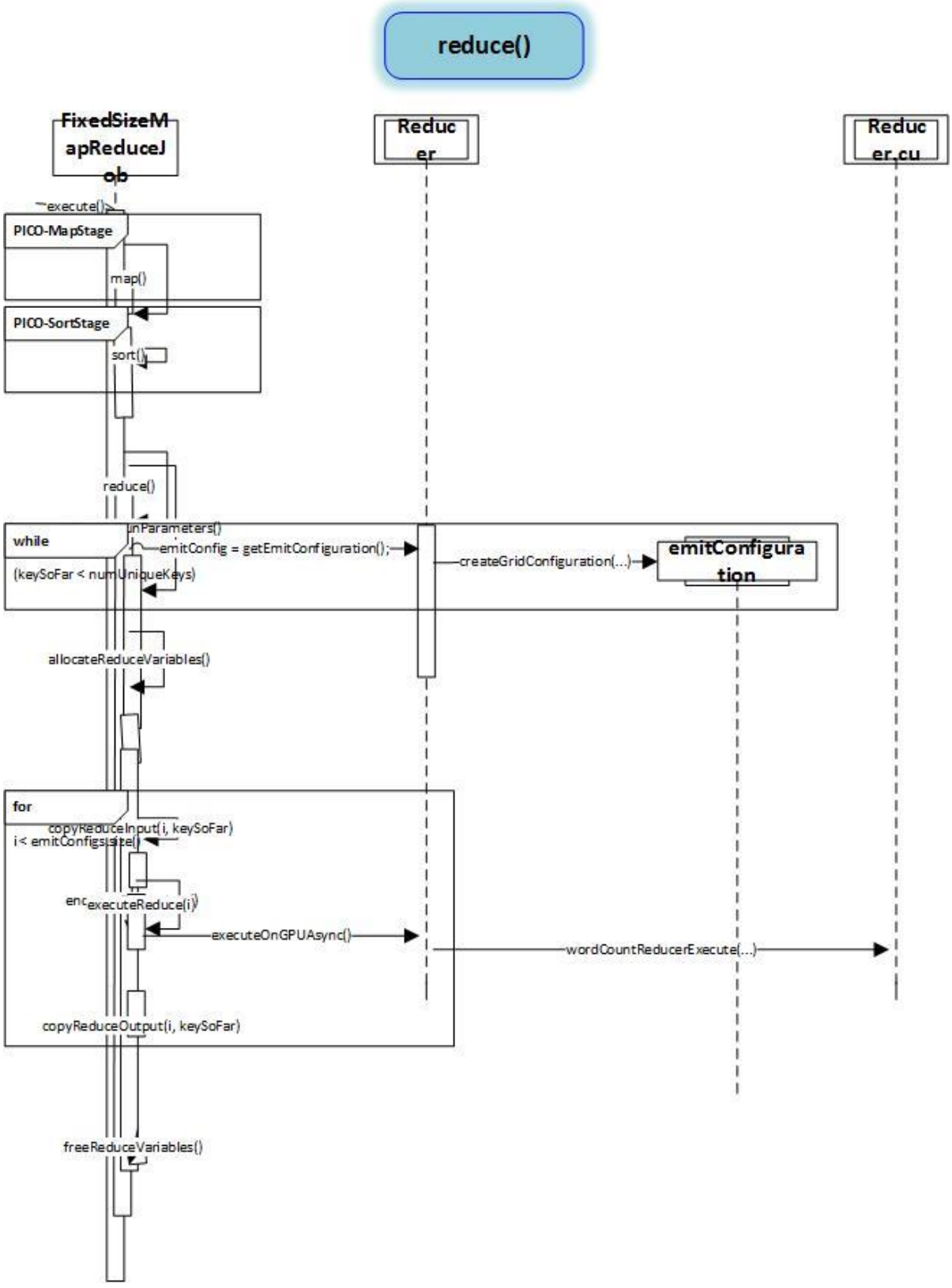

Fig. 5 Reducer

4 EXPERIMENTS AND RESULT

Using Pico, we have written a benchmark application, the word count or word frequency. We have used different subsets of overly simplified data set from Reuter's data set for word count. For comparison with Pico, we use the best known GPU-Mapreduce framework Mars[] .

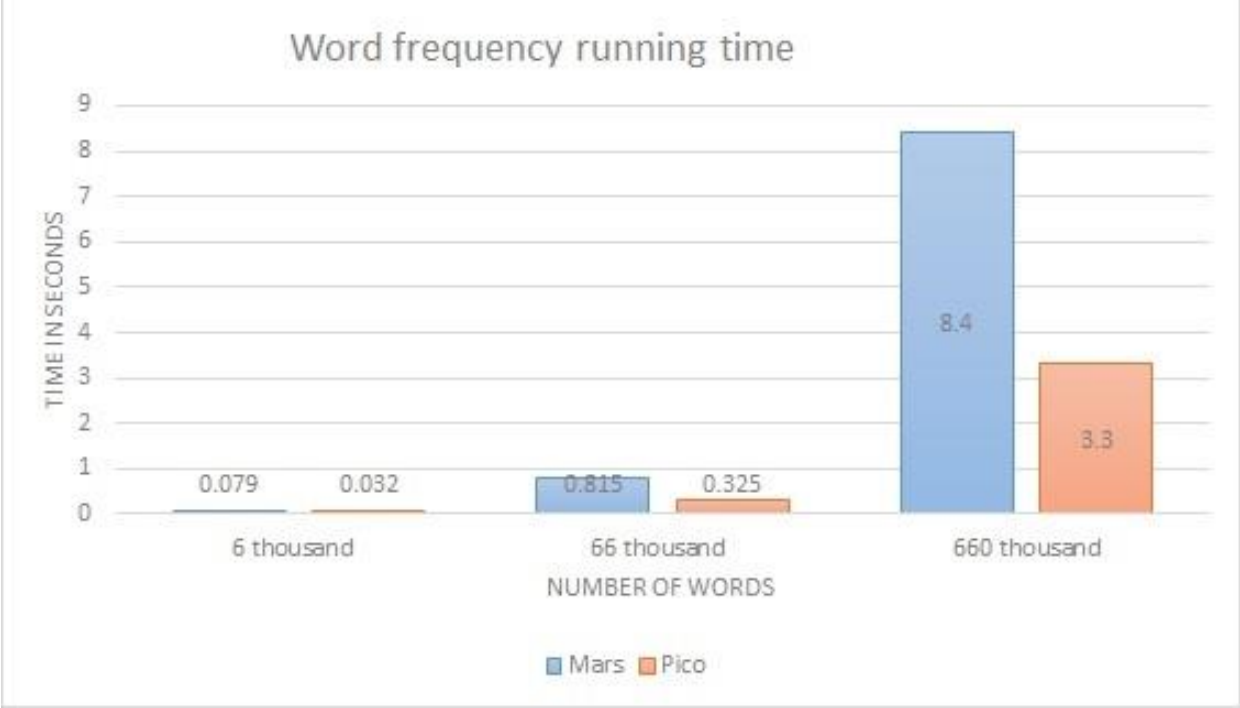

Fig. 2. Word frequency running times for three subsets of dataset with both Pico and Mars on single node Jetson Tk1

The results show that Pico has a speedup of up to 2.47 over Mars with same datasets for the benchmark, word count.

5 GOALS FOR FYP-II

As for the future FYP-II goals we have decided to implement and test PICO, on a multiple nodes cluster of Jetson TK1. The cluster will have a total of 7 nodes, each with an NVIDIA Tegra Jetson TK1. The Jetson TK1 features NVIDIA Kepler GPU GK20a, with 192 CUDA Cores (up to 326 GFLOPS), NVIDIA 4-Plus-1™ Quad-Core 2.3GHz ARM® Cortex™-A15 CPU with 2 GB x16 RAM Memory with 64-bit Width. The nodes will be connected via QDR Inifini band connected to generation-1 PCI-e. 7 Jetsons are considered a small-to medium cluster, but very few existing cluster installations worldwide feature more than these specifications.

Each node in this cluster runs L4T (Linux for Tegra) Linux kernel, and uses the CUDA 6.0 toolkit. We compiled our library and all test software using GCC 4.3.2 and MVAPICH2. To get handsome comparisons and test results we will be executing the following benchmarks on the Jetson cluster: Matrix Multiplication (MM, multiplies two large square matrices); Sparse Integer Occurrence (SIO,

counts the number of times each integer appears in a large dataset); Word Occurrence (WO, counts the number of times each word occurs in a text corpus). These are all legacy benchmarks for testing new MapReduce libraries.

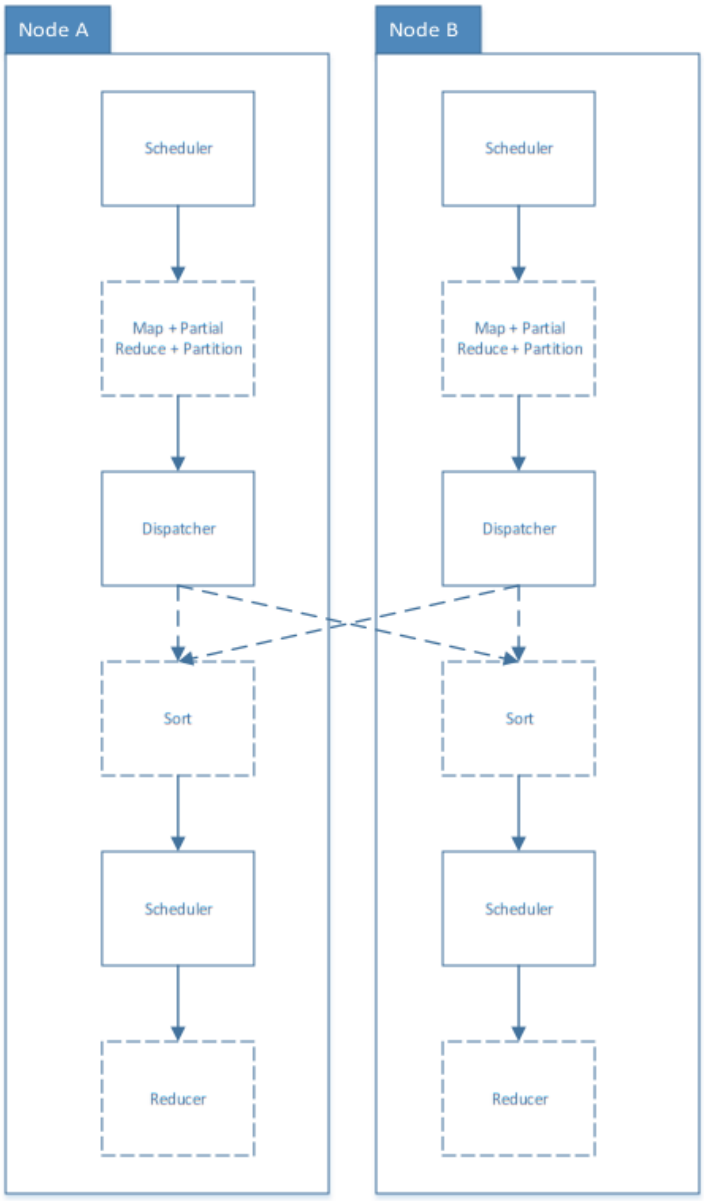

Fig. 3. Pipeline architecture for multi-node Pico

6 WORK DISTRIBUTION

Phase	Team Member
Design	Ahmad Rahman, Abdul Basit, M. Usman Irfan
Implementation-Map	Ahmad Rahman, M. Usman Irfan
Implementation-Sort	Abdul Basit
Implementation-Reduce	Abdul Basit
Testing	Ahmad Rahman, M. Usman Irfan
Documentation	Ahmad Rahman, Abdul Basit

7 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.