EFFICIENT TIME-ENERGY EXECUTION OF DATA-PARALLEL APPLICATIONS ON HETEROGENEOUS SYSTEMS WITH GPU

Dumitrel Loghin

(M. Eng., University "Politehnica" of Bucharest, 2012)

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

**DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE**

2017

Supervisor:
Associate Professor Teo Yong Meng

Examiners:
Professor Tan Kian Lee
Associate Professor Wong Weng Fai
Professor Michael O'Boyle, The University of Edinburgh

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Dumitrel Loghin

Dumitrel Loghin
17th September 2017

Abstract

The last decade has seen the exponential growth of data and the advent of data-parallel processing frameworks such as Googles Cloud Dataflow and MapReduce. On the other hand, hardware systems have entered the heterogeneity era where multiple processing units with different performance-to-power ratio are combined into a single system. At the same time, low-power (wimpy) systems traditionally used in mobile devices have made significant improvements in performance and are targeting server system market dominated by high-performance (brawny) x86-64 systems. In this context, it is important to study the efficiency of running data-parallel applications on heterogeneous systems.

In this thesis, we propose techniques for efficient execution of data-parallel processing on heterogeneous systems with GPUs. Our lazy processing technique enables the parallel processing of multiple input records on the GPU in contrast with chunking of a single record among GPU threads. At runtime, our one-time dynamic mapping technique selects the best execution unit for data-parallel processing between the CPU and GPU. This approach is implemented in MoSS, a Hadoop-CUDA framework that we have developed. Compared to Hadoop, MoSS reduces the execution time by a factor of up to 2.3 on brawny systems, and 3.1 on wimpy systems together with a maximum energy reduction of 80% for compute-intensive workloads. On average, MoSS is over 50% faster compared with the chunking approach.

Secondly, we perform a measurement-driven analysis of MapReduce on intra-node heterogeneous systems with (i) ARM big.LITTLE CPU and (ii) discrete and integrated GPU. Our analysis of ARM big.LITTLE systems shows that there is no one size fits all rule for efficient data-parallel processing on these systems. However, small memory size, low memory and I/O bandwidth, and software immaturity concur in canceling the lower-power advantage of ARM systems. Our analysis of heterogeneous systems with both discrete and integrated GPUs reveals that wimpy systems with integrated GPU use the lowest energy due to more energy-efficient hardware and better-balanced system resources. Based on this finding, we establish an equivalence ratio between a single brawny heterogeneous node and multiple wimpy heterogeneous nodes. We show that multiple wimpy nodes achieve the same time performance as a single brawny node, while saving up to two-thirds of the energy.

Thirdly, we design measurement-driven time-energy analytic models to determine the execution time and energy usage of data-parallel execution on both homogeneous systems running Hadoop and heterogeneous systems running MoSS. To the best of our knowledge, we are the first to design an energy usage model for MapReduce execution. Since our modeling approach uses baseline measurements to increase model accuracy, the validation on up to 264 system configurations shows an average model error of less than 15%. Using our models, we analyze the performance of hypothetical scale-out clusters with more than 100 nodes. This analysis shows that heterogeneity always achieves better time-energy performance when the workload consists of a compute-intensive part. In line with our measurement-driven analysis, we show using our models that multiple wimpy nodes not only achieve similar execution times compared to brawny nodes, but also exhibit energy savings of up to 90% for compute-intensive workloads. This, together with MoSS performance results, advocate the potential usage of wimpy systems with integrated GPU for data-parallel processing.

“nil posse creari de nihilo”
– Lucretius

Acknowledgements

Firstly, I am deeply grateful to my supervisor, Professor Yong Meng Teo, for his continuous support, insightful feedback and valuable lessons during these five years that we have worked together. Professor Teo has taught me how to do research by asking the important questions and filtering out the "noise". He had the patience to help me improve my writing skills and my presentation skills. I would also like to thank Professor Teo for nominating me and to acknowledge National University of Singapore for granting me the President's Graduate Fellowship during my candidature.

I would like to thank Professor Kian-Lee Tan and Professor Weng-Fai Wong, members of my thesis committee, for their comments and advice during different stages of my Ph.D. candidature. I would also like to thank Professor Beng Chin Ooi for his insightful comments that helped me improve both the content and the presentation of my articles. Moreover, Professor Ooi has granted me access to state-of-the-art computer systems that were crucial to my research. I want to extend my gratitude to Professor Tulika Mitra and Professor Chin Wei Ngan for granting access to computer systems in key moments for my research.

During my years at National University of Singapore, I was lucky to have been able to work with Professor Hugh Anderson and Professor John L. Gustafson as Teaching Assistant. I have learned much in terms of technical and teaching skills from them both.

I would not have embarked on this PhD journey without the guidance and support of two of the most talented people I have ever met. Thus, I express a special and sincere gratitude to Dr Cristina Carbunaru and Dr Bogdan Marius Tudor. They have helped me during the most difficult times and for that I am most grateful to count them as my closest friends.

A very special acknowledgment for my colleague, Dr Lavanya Ramapantulu, with whom I share an impressive list of co-publications that stands witness to

our fruitful collaboration. I trust that we will continue to collaborate even as we move to different parts of the world. I would also like to thank Dr Bogdan Marius Tudor, Dr Hao Zhang and (soon to be doctor) Sunimal Rathnayake with whom I have collaborated on research projects.

Life in Singapore wouldn't have been so fun without my friends: Bogdan, Cristina, Mihai, Delia, Cristi, Narcisa, Andreea, Yamilet, Khanh, Dan, Kay, Gabor, Tudor, Oana, Sunimal, Thy, Suman, Irvan, Teodora, Mihai.

I would like to express my deepest gratitude to both my families in Singapore and Romania. My dear wife Yamuna, whom I met in Singapore, has always filled me with joy and helped me see the future with optimism. Her family welcomed me and made me feel like home. There are not enough words to express my gratitude for this.

A day away, back in Romania, my parents, uncles, aunties and grandparents have always stood by me and encouraged me to pursue my dreams even if that meant I had to be so far away for long periods of time.

I dedicate this thesis to my grandparents, who have given me a wonderful childhood, and whom I know miss me dearly.

Table of Contents

List of Publications

1. Dumitrel Loghin, Lavanya Ramapantulu, Yong Meng Teo, **Efficient Time-Energy Execution of MapReduce on Heterogeneous Systems with GPU**, submitted, 2017. [MoSS'17]

2. Dumitrel Loghin, Lavanya Ramapantulu, Yong Meng Teo, **On Understanding Time, Energy and Cost Performance of Wimpy Heterogeneous Systems for Edge Computing**, Proc. of 1st IEEE International Conference on Edge Computing, pages 1-8, 2017. [EDGE'17]

3. Dumitrel Loghin, Lavanya Ramapantulu, Oana Barbu, Yong Meng Teo, **A Time-Energy Performance Analysis of MapReduce on Heterogeneous Systems with GPUs**, Performance Evaluation, 91:255-269, 2015. [PEVA'15]

4. Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi and Yong Meng Teo, **A Performance Study of Big Data on Small Nodes**, VLDB Endowment, 8(7):762-773, 2015. [VLDB'15]

5. Dumitrel Loghin, Bogdan Marius Tudor and Yong Meng Teo, **An Approach for Direct Dataflow Execution on Contemporary Multicore Systems**, Proc. of 3rd International Workshop on Dataflow Execution Models for Extreme Scale Computing (in conjunction with PACT), pages 1-8, 2013. [DFM'13]

6. Sunimal Rathnayake, Dumitrel Loghin, Yong Meng Teo, **CELIA: Cost-time Performance of Elastic Applications on Cloud**, Proc. of 46th International Conference on Parallel Processing, pages 342-351, 2017. [ICPP'17]

7. Lavanya Ramapantulu, Thy Dao, Dumitrel Loghin, Nam Thoai, Yong Meng Teo, **Modeling the Energy-Time Performance of MIC Architecture System**, Proc. of 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pages 85-94, 2016. [MASCOTS'16]

8. Lavanya Ramapantulu, Dumitrel Loghin, Yong Meng Teo, **On Energy Proportionality and Time-Energy Performance of Heterogeneous Clusters**, Proc. of 18th IEEE Cluster Conference, pages 221-230, 2016. [CLUSTER'16]

9. Lavanya Ramapantulu, Dumitrel Loghin, Yong Meng Teo, **An Approach for Energy Efficient Execution of Hybrid Parallel Programs**, Proc. of 29th International Parallel and Distributed Processing Symposium, pages 1000-1009, 2015. [IPDPS'15]

10. Lavanya Ramapantulu, Bogdan Marius Tudor, Dumitrel Loghin, Trang Vu and Yong Meng Teo, **Modeling the Energy Efficiency of Heterogeneous Clusters**, Proc. of 43rd International Conference on Parallel Processing, Minneapolis, pages 321-330, 2014. [ICPP'14]

List of Figures

List of Tables

List of Algorithms and Code Listings

Chapter 1

Introduction

In the last few years, we have witnessed the explosion of Big Data analytics triggered by the increasing volume, velocity and variety of collected data. This explosion is driven by the adoption of data-parallel processing frameworks such as MapReduce [33], Hadoop [14], Spark [101], Cloud Dataflow [45] by both the industry and academia. These frameworks are primarily designed for clusters of homogeneous systems where scalability is achieved by increasing the number of cluster nodes. But the high power consumption of traditional server nodes equipped with x86-64 processors [56] and the execution inefficiencies of data-parallel processing frameworks [66] lead to high energy usage.

At the same time, the end of Dennard scaling [37] pushes hardware systems towards heterogeneity by integrating multiple processing units with different performance-to-power ratio (PPR). This heterogeneity permeates the system at *intra-chip*, *intra-node* and *inter-node* levels and exposes a large configuration space that could significantly improve the match between software's dynamic resource demands and heterogeneous system's capacity [81].

Graphics Processing Unit (GPU) is a major driver for heterogeneous computing as it can significantly improve energy efficiency by exploiting massive thread level parallelism (TLP). Additionally, the improvement in GPU programmability over the last decade has accelerated their adoption in supercomputers and datacenters. For example, the number of Top500 systems with Nvidia GPUs has increased from 9 in 2010 to 71 in 2017 [90]. Traditionally, GPUs are exploited in high-performance computing (HPC), but they are being increasingly employed in data analytics or machine learning. For example, Nvidia built a 12-node system with GPUs that can outperform the Google Brain system consisting of 16000 CPU cores [77]. These 12 Nvidia GPUs are 100 times more energy-efficient compared to the Google Brain, making it possible to scale-out machine learning systems that can ultimately model the human brain. This is becoming feasible since modern GPUs are more energy-efficient. For example, Nvidia Maxwell GPUs are at least two times more energy-efficient than previous Kepler generation [51].

Concomitantly, the past few years have seen a spectacular evolution in the performance of ARM-based systems that were traditionally used in smartphones and tablets. Most of these systems have processors with four or eight cores and clock frequencies exceeding 2 GHz, memory sizes of up to 4 GB, and fast flash-based storage. The latest generations of mobile ARM-based systems can run full-fledged operating systems such as Linux or Windows, and the entire stack of user-space applications available for these operating systems. Moreover, supercomputers and datacenters are being increasingly interested in adopting ARM-based hardware [78, 85, 92]. For example, Microsoft has adapted Windows Server for running on ARM-based servers to prepare their integration in Azure cloud [92]. Due to smaller size, smaller power requirements, and lower performance, these systems are often called *small nodes* or *wimpy nodes* as opposed to traditional high-performance *brawny nodes* [49]. However, it remains to be explored if these wimpy systems suitable for Big Data analytics.

With the varying resource demands of mobile apps, these wimpy systems are becoming heterogeneous by integrating CPU cores with different PPRs, such as ARM big.LITTLE [19], and accelerators, such as Nvidia GPUs [75]. The performance improvements of wimpy nodes promote them as an alternative candidate for datacenter computing with the potential to reduce the energy and, thus, datacenter operational costs. Big hardware vendors, such as Dell, AMD, Applied-Micro and Nvidia have already launched server prototypes based on ARM cores and low-power accelerators [30, 75]. In this context, it is fundamental to analyze if heterogeneous systems are more suitable for Big Data analytics compared to traditional homogeneous systems.

Efficiently exploiting heterogeneous systems is a daunting task. On the one hand, explicitly programming in C/C++ with OpenMP, OpenCL, CUDA and MPI, and handling fault tolerance is burdensome. Moreover, adding energy efficiency as a design goal and choosing the most energy-efficient configuration while meeting an execution time deadline is a non-trivial task [79, 80, 81]. On the other hand, frameworks that implicitly handle parallelism, such as Hadoop [14], Spark [101] and Beam [11], are designed for homogeneous systems and may exhibit inefficient execution on heterogeneous systems [66].

In this thesis, we address the challenge of efficiently executing data-parallel applications on heterogeneous systems by (i) proposing techniques to enable efficient time-energy execution of data parallel applications on intra-node and intra-chip heterogeneous systems with GPU [70], (ii) performing in-depth measurement-driven analysis to show that wimpy heterogeneous systems are more energy-efficient compared to brawny systems [71, 72], and (iii) modeling time-energy performance to determine optimal configurations for scale-out workloads and clusters [70]. In the remainder of this section, we present the opportunities and challenges exposed by data-parallel processing and heterogeneous computing, followed by the objective of this thesis.

1.1 Data-Parallel Applications

In the last two decades, the computing landscape has been shifting towards parallel programming since multicore systems became mainstream. More recently, the explosion of Big Data strengthened the need for scalable and fault-tolerant frameworks such as MapReduce [33]. However, with the increasing complexity and velocity of data processing, more flexible programming models, such as dataflow programming, are needed [45, 66]. As a response, academia and industry players developed low-latency data-parallel processing frameworks, such as Spark [100], Spark Streaming [102], Google Cloud Dataflow [4,11,24,45], among others. These frameworks enable fault-tolerant data-parallel execution at large scale.

Parallel applications expose two types of parallelism, namely, task and data parallelism. The former refers to distributing tasks with different functionality among multiple processing units and it corresponds to *Multiple Instruction, Single Data* (MISD) and *Multiple Instruction, Multiple Data* (MIMD) types of computer architectures in Flynn's taxonomy [40]. On the other hand, data parallelism refers to applying the same operation on a large set of data. This corresponds to *Single Instruction, Multiple Data* (SIMD) architecture in Flynn's taxonomy. Moreover, data-parallel processing can be classified into batch and stream processing. Batch data-parallel processing handles large volumes of data, entirely available before execution, and produces consolidated results. The key objective of batch data-parallel systems is to achieve high throughput, rather than low latency. On the other hand, stream data-parallel processing consumes input records that arrive as time passes by, and produces output in a continuous form, aiming for low latency. Independent of the type of parallelism, applications exhibit different system resource demands. We classify an application that stresses computational units such as the CPU or GPU as *compute-intensive*, while an application that

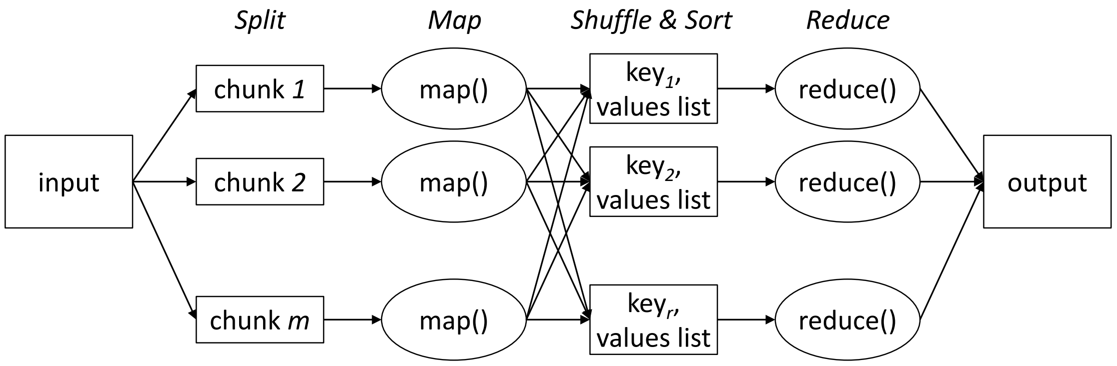


Figure 1.1: MapReduce execution phases

requires many data transfers at memory, storage or network level is *data-intensive.* Nonetheless, some applications have mixed compute- and data-intensive profile. In this thesis, we are focusing on batch data-parallel application with diverse system resource demands and executed on frameworks such as MapReduce (Hadoop) and Google Cloud Dataflow (Beam).

MapReduce was introduced by Google in 2004 as a programming model and an associated framework for processing big amounts of data in a scalable and fault-tolerant way [33]. In 2007, Hadoop was released to become the most popular and widely-used MapReduce implementation [66]. MapReduce processing consists of four steps or phases as depicted in Figure 1.1:

- *Split* - the input is split into several chunks of *records* or *<key, value> pairs.*
- *Map* - each *<key, value>* pair is processed by user-defined *map*() function and, as a result, none, one or more new *<key, value>* pairs are emitted to the next phase.
- *Shuffle* and *Sort* - Map output pairs are organized in *<key, values list>* based on their key. Usually, this is done using a sorting mechanism.
- *Reduce* - each *<key, values list>* is processed by user-defined *reduce*() function which emits new *<key, value>* pairs as the final output.

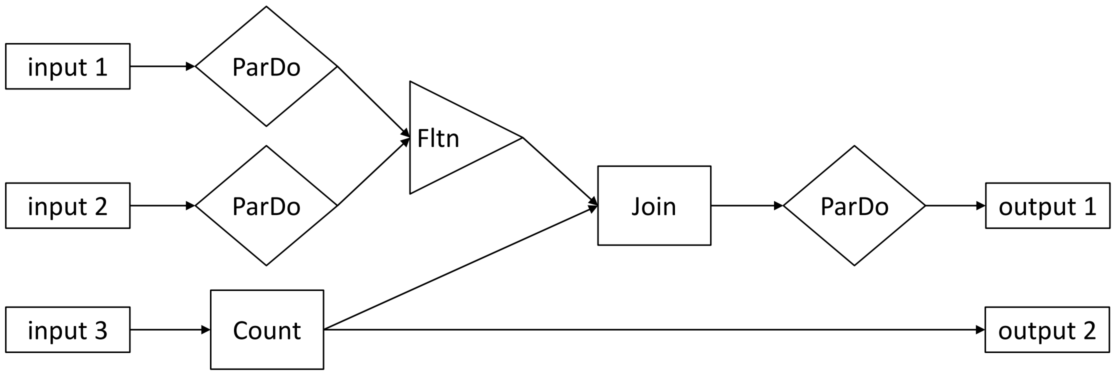


Figure 1.2: Example of Cloud Dataflow processing (adapted from [24])

Among these four steps, only Map and Reduce are exposed to application developers. Split, Shuffle and Sort phases, and Map and Reduce task management are handled by the framework. MapReduce is an expressive programming model which, in conjunction with its associated framework, achieves high scalability and fault tolerance. However, MapReduce may exhibit low runtime efficiency and high energy usage on homogeneous brawny nodes [66].

In 2014, Google announced a new Big Data processing framework, called Google Cloud Dataflow [45]. As an enhancement over MapReduce, this framework:

- allows users to create multi-step pipelines of data processing, as depicted in Figure 1.2.
- allows more types of operators, not only map and reduce. For example, it exposes *ParDo* which is similar to *Map*, *Flatten*, *Join*, *Group*, *Count* and other operators.
- supports low-latency stream processing. For this, data is represented as *(key, value, timestamp)*, compared to just <*key, value*> pairs in MapReduce. The timestamps are similar to tags in the traditional dataflow model.

Compared to MapReduce, Cloud Dataflow exposes more operators and enables pipelining. However, Map and Reduce operators are the building blocks of all other Cloud Dataflow operators [24].

In 2016, Apache released a proposal for an open source implementation of Google Cloud Dataflow under Apache Beam [11]. Some researchers see Cloud Dataflow as the end of MapReduce era. Nevertheless, it is important to note that FlumeJava, the key component of Dataflow, internally represents its computations as pipelines of Map and Reduce operators [24]. Moreover, major industry players, including Amazon [6], IBM [58], Microsoft [74] and even Google [46] are providing cloud-based MapReduce platform-as-a-service.

With the explosion of Big Data, the focus for software developers is on exploiting large scale parallel and distributed systems while ensuring low-latency processing and fault tolerance. However, achieving this depends on addressing overhead management and efficient resource usage. Traditional parallel programming models and languages, such as pthreads, OpenMP and MPI, put the burden of designing parallel programs on application developers, by having to explicitly define the partitioning, synchronization and mapping of the parallel tasks [87]. Hence, scaling applications using such methods is a challenge. On the other hand, models that automatically handle partitioning, synchronization and mapping usually suffer from inefficient execution. For example, Hadoop MapReduce is a scalable, portable and fault-tolerant framework, but suffers from an inefficient execution in terms of time and energy [28, 61, 65, 66]. In this context, we investigate how to combine the performance of traditional programming models with the flexibility of dataflow and data-parallel models to achieve time-energy-efficient execution on heterogeneous systems.

1.2 Heterogeneous Systems

Processors with multiple cores have become the norm in the last decade. With each technological generation, multicore processors have an increasing number of cores integrated per die. However, this trend will soon reach its limit because of transistor power requirements [37]. Due to these power constraints, only part of the chip will be powered at one moment, a phenomenon called *dark silicon*. As a response to this concern, system architects are proposing the usage of heterogeneous systems integrating multiple specialized processing units which can be powered based on applications demands. For example, ARM big.LITTLE architecture was a major milestone in core heterogeneous architectures [19]. The first big.LITTLE chip with 32-bit Cortex-A7 and Cortex-A15 cores was announced in 2011. By the end of 2012, ARM announced the next generation 64-bit big.LITTLE based on Cortex-A53 and Cortex-A57 cores. In 2014, Nvidia introduced Jetson TK1, a heterogeneous wimpy system integrating ARM CPU cores and Nvidia GPU cores on the same die [75].

Another milestone was the launching of Heterogeneous System Architecture (HSA) in 2012 by a consortium of companies led by AMD [7]. This consortium proposes standards and tools for better integration of different processing units, such as CPUs and GPUs, at both hardware and software levels. As part of this effort, they propose HSA Intermediate Language (HSAIL) which can be efficiently mapped by each hardware vendor to its own low-level language, but is flexible enough to be generated once from the source code and to run on different processing units.

With heterogeneity becoming the new norm, we define and classify heterogeneous systems, and present their potential in terms of both time performance and energy efficiency. Heterogeneous systems are defined as having more than one type of processing unit integrated on the same system [9]. Moreover, based on system type, heterogeneity can be classified into:

- *intra-chip heterogeneity* when the system is a single chip integrating different types of processing units such as CPU cores, GPU cores or DSPs. These systems are often called System on a Chip (SoC). An example of intra-chip heterogeneity with *integrated* GPU is the Nvidia Tegra K1 that powers Jetson TK1 system which integrates four ARM Cortex-A15 CPU cores, 192 Nvidia Kepler GPU cores and a shared 2 GB low-power memory [75]. A particular case of intra-chip heterogeneity is when the CPU has multiple cores with different capabilities, such as big and little cores in ARM big.LITTLE architecture [19].
- *intra-node heterogeneity* is a commonly encountered heterogeneous setup involving a multicore CPU and a *discrete* GPU accelerator connected through the PCI-Express interface.
- *inter-node heterogeneity* refers to heterogeneous clusters. In a typical setup, these clusters consist of both low-power systems such as those based on Intel Atom or ARM processors and high-performance systems such as those based on Intel Xeon or AMD Opteron processors. An example of such a system is KnightShift [98] which uses Intel Atom and Intel Xeon by switching between them based on workload demand.

Heterogeneity implies the existence of different processing capabilities, each type of core being specialized for a particular application. For example, CPU cores are specialized in executing sequential code with lots of branches and loops. On the other hand, GPUs are specialized in executing SIMD code where a single operation is applied to multiple values. When executing different applications, these processing resources have different PPR representing the average unit of useful work performed per unit of energy. By efficiently exploiting heterogeneous resources, software applications can achieve energy efficiency. To illustrate this, we present in Table 1.1 the PPRs of two different applications on two different systems, a server-class computer based on Intel Xeon E5-2603 processor and an embedded-class Odroid XU board based on Samsung Exynos 5410 SoC which implements ARM big.LITTLE architecture. Since this architecture is versatile, we present the results for three different configurations, namely, when activating only Cortex-A7 (*little*) cores, when activating only Cortex-A15 (*big*) cores, and when activating both types of cores. The first application is a multi-threaded version of Black-Scholes financial model from PARSEC benchmarks [22]. For this application, we define the PPR as the ratio between the number of processed financial options and the energy used. Even though the execution time on ARM system is three (five on little) times bigger compared to Xeon system, the PPR has the reverse trend. When using only little cores, the PPR of ARM Cortex-A7 is almost five times better than that of Xeon and two times better than that of big Cortex-A15 cores. The second application is Wordcount, a well-known Big Data analytics application implemented using MapReduce programming model [33]. This application computes the number of appearances of words in a series of documents. Since it scans all the input documents, we compute the PPR as their total size in MB divided by the total energy. The workload was run on Hadoop 1.2.1 on a single node cluster. For this workload, the difference in execution time on ARM and Xeon systems is significant, ranging from seven to nine times. Even if the PPR of ARM system is better, the difference between the two types of systems is lower compared to first benchmark.

The advance of heterogeneous hardware platforms introduces a series of challenges for software developers. While many developers advocate the principle of "write once, run everywhere" [31], this approach may expose runtime inefficiencies

Table 1.1: Performance-to-power ratio of heterogeneous systems

Platform	**Processing Unit**	**Blackscholes**		**Wordcount**	
		Time [s]	PPR [Mopt/J]	Time [s]	PPR [MB/J]
Xeon server	Intel Xeon E5-2603	43	0.12	1507	0.10
Odroid XU	ARM Cortex-A7 (LITTLE)	205	0.50	13799	0.18
	ARM Cortex-A15 (big)	132	0.25	11209	0.12
	ARM big.LITTLE	133	0.25	10897	0.12

without proper compilers and runtime engines. For example, programs written in Java are supposed to run on a wide range of hardware platforms through Java Virtual Machine. Nevertheless, this may incur loss of performance on heterogeneous systems since different processing units need different optimizations. Thus, careful design of runtime engines is needed. However, inefficient usage of processing units is not the only source of performance loss. Memory and I/O subsystems can become the bottleneck for a variety of workloads. For example, in ARM-based systems, memory and network I/O are the bottleneck for a series of server workloads, and their energy efficiency is nullified [91].

Heterogeneity introduces a multitude of configurations with different balances of cores, memory, disk and network I/O system resources to meet diverse application needs. As a result, the large configuration space provides new opportunities to achieve time-energy performance trade-offs. These performance trade-offs can satisfy the demands of both users and resource providers. Considering this, our aim is to design model-based techniques to identify optimal configurations of intra-node and intra-chip heterogeneous systems for executing applications with minimum energy while meeting a given time deadline.

Lastly, we present a motivational example of data-parallel processing on heterogeneous systems with GPUs. MapReduce was originally designed for CPU-only server systems and it is non-trivial to run MapReduce applications on heterogeneous CPU+GPU systems. To overcome this, we design a *lazy processing* tech-

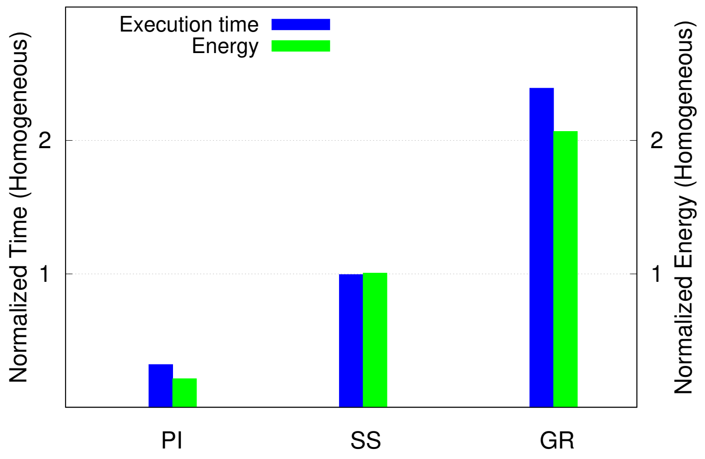


Figure 1.3: Execution time and energy comparison between homogeneous CPU-only systems and heterogeneous systems with GPU

nique that enables the GPU to process multiple <*key, value*> pairs in parallel. We measured the execution time and energy consumed by Hadoop on clusters of 12 Jetson TK1 nodes [75], for both CPU-only (**CPU**) execution and heterogeneous CPU+GPU (**GPU**) execution. Figure 1.3 shows execution time and energy for three applications, Pi estimation (PI), Similarity Score computation (SS) and Grep (GR), normalized to the values for CPU-only execution. The results expose a varied landscape, where some applications greatly benefit from GPU processing, while others exhibit similar or worse performance on GPU compared to CPU-only execution.

This example motivates the need for dynamic mapping techniques for selecting the most suitable heterogeneous execution unit at runtime. This selection could be accomplished by static methods in the absence of runtime variations. However, in practice there are situations where system parameters change and static methods may not achieve optimal performance. For example, if the GPU is used by more applications at the same time, static methods alone do not suffice [47]. In our case, if the code is statically optimized for small-size input records suitable for GPU execution, but in production the input contains unusually large records, GPU processing will be inefficient. In such a case, dynamic techniques are more flexible, even if they incur overhead due to profiling and scheduling.

In summary, heterogeneity introduces opportunities for efficient time-energy execution of data-parallel applications. However, the behavior of current data-parallel processing frameworks on heterogeneous systems needs to be analyzed using measurement and model-based approaches. Moreover, software techniques are required for efficient data-parallel execution on heterogeneous systems.

1.3 Objective, Approach and Contributions

The objective of our work is to efficiently execute batch data-parallel applications on intra-node and intra-chip heterogeneous systems using measurements to analyze software and hardware bottlenecks and analytic models to determine optimal system configurations. The main efficiency criteria we consider are execution time and energy, and their derivatives such as performance-to-power ratio (PPR). As opposed to the majority of related work which focuses on time performance only [2, 61, 83, 84, 103], we also address the energy performance. The energy efficiency of applications running on heterogeneous systems becomes more important as we enter the era of *edge computing* [3].

To achieve our objective, we propose an approach divided into three parts:

1. *techniques* to enable efficient batch data-parallel processing on intra-node and intra-chip heterogeneous systems with GPU

2. identify bottlenecks and expose the limitations of current hardware and software systems through an in-depth *measurement-driven analysis* of batch data-parallel applications on intra-node and intra-chip heterogeneous systems

3. determine scale-out system configuration that efficiently execute batch data-

parallel applications by developing *time-energy analytic models*.

A detailed description of our approach is given in Chapter 3. Next, we highlight the contributions of this thesis:

1. To efficiently execute data-parallel applications on intra-node and intra-chip heterogeneous systems with GPUs, we develop:

    - a *lazy processing* technique that enables the processing of multiple input records on a GPU [70, 71], in contrast with *chunking* which divides a single record among GPU threads [84]. Lazy processing is 54% faster than chunking, on average, and saves 66% of the energy on wimpy heterogeneous systems
    - three *dynamic mapping* techniques to further improve data-parallel processing at runtime [70]. The *one-time* workload profiling approach selects the most suitable processing unit between the CPU and GPU, while *checking* and *callback* are overlapping CPU and GPU processing. Counter to intuition, we show that *one-time* profiling achieves better performance than overlapping the execution on the CPU and the GPU
    - *MoSS* (*M*apReduce *o*n Heterogeneo*S* *S*ystems), an implementation of our techniques using Hadoop and CUDA [70].

2. To analyze time-energy performance, we perform measurements of data-parallel execution on brawny and wimpy intra-node heterogeneous systems and show that there is no "one size fits all" rule for efficient execution on these systems. For intra-node heterogeneous systems with GPU we show that:

    - MoSS improves the execution time up to 2.3 times on brawny systems with GPU, and up to 3.1 times on wimpy systems with GPU along with a maximum of 80% reduction in energy usage for compute-intensive workloads, when compared with Hadoop [70]

   - while compute-intensive applications benefit from heterogeneity, applications where data transfers dominate the execution time, such as Grep and Word-count, exhibit worse time-energy performance on heterogeneous systems compared to homogeneous systems [71]
   - based on the execution time equivalence ratio, multiple wimpy nodes achieve the same performance as one brawny node while saving two thirds of the energy [71].

   For intra-node heterogeneous systems with different CPU cores, such as ARM big.LITTLE we show that:

   - compute-intensive workloads run five times faster on wimpy ARM nodes with minor software modifications [72]
   - it is four times cheaper in terms of total cost to run compute-intensive Map-Reduce on wimpy nodes compared to traditional brawny server nodes [72]
   - software immaturity and limited memory size and bandwidth are the main issues that affect data-parallel execution performance of wimpy nodes [72].

   The detailed time-energy performance analysis of data-parallel execution on ARM big.LITTLE is presented in Appendix B.

3. To analyze the time-energy performance of hypothetical or scale-out systems:

   - we develop *measurement-driven time-energy analytic models* to determine the execution time and energy usage of MapReduce data-parallel execution on both homogeneous systems running Hadoop and heterogeneous systems with GPU running MoSS. Validated against real measurements on up to 264 configurations, our models exhibit an average error of less than 15%.

- we use our models to study the time-energy performance of scale-out clusters of more than 100 nodes and we show that (i) heterogeneous systems almost always achieve better time performance and save energy for workloads with at least 10% compute-intensive fraction, and that (ii) an execution time equivalence ratio between brawny and wimpy systems exists for scale-out clusters, where wimpy systems can save up to 90% of the energy for compute-intensive workloads.

To the best of our knowledge, we are the first to design an energy usage model for data-parallel MapReduce execution and the first to present a time-energy analysis of data-parallel processing on wimpy heterogeneous systems. In addition, we perform a detailed system characterization at CPU, GPU, memory, storage and networking level for all systems used in our research, as detailed in Appendix A.

1.4 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we present the related work on data-parallel processing on heterogeneous systems, with focus on the time-energy performance. We first present the work done on improving the time performance of data-parallel processing on different types of heterogeneous systems, followed by studies on improving the energy efficiency. Next, we present existing performance models and show that there is no energy usage model for data-parallel processing. Lastly, we expose related work limitations in contrast with the contributions of this thesis.

In Chapter 3, we first present our overall approach and then focus on the techniques for efficient data-parallel execution on heterogeneous systems with GPUs, in contrast with related work. Next, we present the implementation of these techniques in MoSS using Hadoop and CUDA.

In Chapter 4, we evaluate MoSS and analyze the time-energy performance of heterogeneous systems with GPUs using a measurement-driven approach. We start with the time-energy performance at cluster level, and continue with an in-depth analysis at single node level. For this analysis, we use both brawny and wimpy systems. Brawny systems are represented by cloud-based Amazon EC2 [1] instances and self-hosted server systems with i7 or Xeon CPU and Nvidia GPU. Wimpy systems include both discrete GPUs, such as Kayla [30], and integrated GPUs, such as Jetson TK1 [75]. Using this measurement-driven analysis, we establish an equivalence ratio between one brawny node and multiple wimpy nodes, such that they achieve the same time performance while saving energy.

In Chapter 5, we introduce our hybrid time-energy analytic models that are using measured parameters to improve accuracy. To the best of our knowledge, we are the first to develop an energy usage model for data-parallel MapReduce execution. Using these models, we present a time-energy performance analysis of scale-out clusters and workloads. This analysis confirms the measurements results in Chapter 4 and shows that wimpy heterogeneous systems are an energy-efficient alternative for Big Data analytics.

We conclude by briefly discussing future directions in Chapter 6.

Chapter 2

Related Work

With the explosion of Big Data analytics, there is demand for scalable and fault-tolerant data-parallel processing. Traditional parallel programming models and languages, such as C/C++ with pthreads, OpenMP, MPI and CUDA are burdensome for application developers since they have to explicitly define the partitioning, synchronization and mapping of parallel tasks [42, 87]. Adding scalability and fault tolerance to data-parallel applications makes it even more difficult to use these traditional programming models [33]. On the other hand, models and frameworks that automatically handle partitioning, synchronization and mapping usually suffer from inefficient time-energy execution [66]. Moreover, most of these frameworks, such as MapReduce [33], Spark [101] and Cloud Dataflow [45], are designed for traditional brawny server systems that incur high energy usage.

Based on the degree of heterogeneity of the target system, we present related works and highlight our work in Figure 2.1. We classify the target systems into homogeneous and heterogeneous. We further classify heterogeneous systems into intra-node systems with discrete processing units, intra-chip systems with processing units integrated into the same die, and inter-node systems with different types of cluster nodes. The bulk of work targets homogeneous brawny x86-64

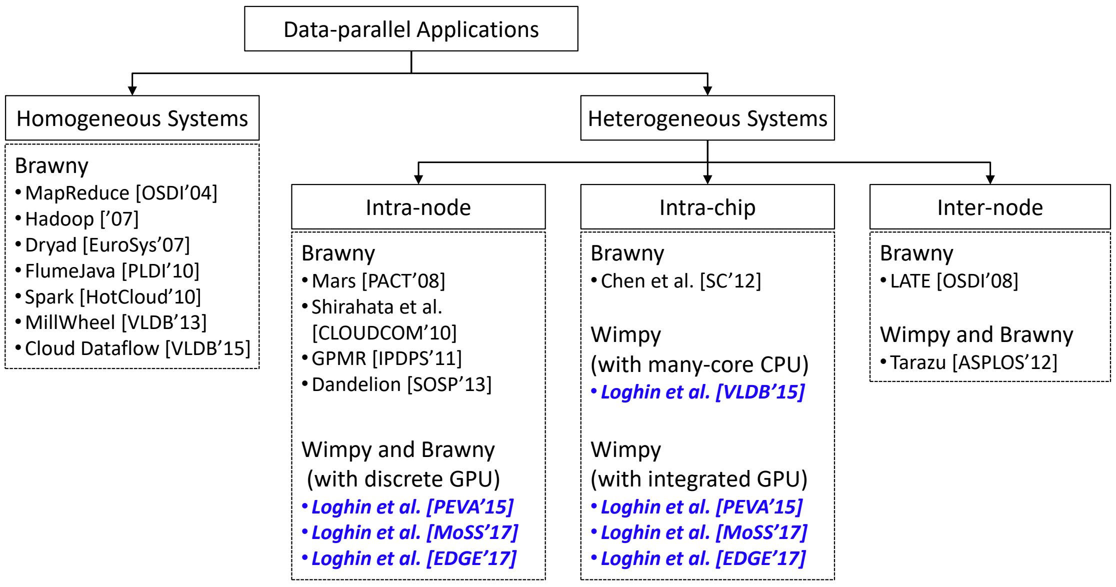

Data-parallel Applications
Homogeneous Systems
Brawny
• MapReduce [OSDI'04]
• Hadoop ['07]
• Dryad [EuroSys'07]
• FlumeJava [PLDI'10]
• Spark [HotCloud'10]
• MillWheel [VLDB'13]
• Cloud Dataflow [VLDB'15]
Heterogeneous Systems
Intra-node
Brawny
• Mars [PACT'08]
• Shirahata et al. [CLOUDCOM'10]
• GPMR [IPDPS'11]
• Dandelion [SOSP'13]
Wimpy and Brawny (with discrete GPU)
• Loghin et al. [PEVA'15]
• Loghin et al. [MoSS'17]
• Loghin et al. [EDGE'17]
Intra-chip
Brawny
• Chen et al. [SC'12]
Wimpy (with many-core CPU)
• Loghin et al. [VLDB'15]
Wimpy (with integrated GPU)
• Loghin et al. [PEVA'15]
• Loghin et al. [MoSS'17]
• Loghin et al. [EDGE'17]
Inter-node
Brawny
• LATE [OSDI'08]
Wimpy and Brawny
• Tarazu [ASPLOS'12]

Figure 2.1: Related work and the contributions of this thesis

server systems since traditional clusters and datacenters consist of such systems. With the emergence of heterogeneous systems, some research projects propose enhancements of data-parallel frameworks for these platforms.

In our work, we propose techniques for efficient data-parallel processing on intra-node and intra-chip heterogeneous systems, focusing on systems with discrete and integrated GPUs. We analyze and model the time and energy performance of these heterogeneous systems in comparison with homogeneous systems. To the best of our knowledge, we are the first to analyze the energy efficiency of wimpy heterogeneous systems with discrete and integrated GPU [71] or with ARM big.LITTLE CPU [72], and the first to propose an energy usage model for MapReduce on both homogeneous and heterogeneous systems [70].

In the remainder of this chapter, we first present techniques and frameworks for data-parallel processing based on the type of systems they are designed for. We continue with performance models and then discuss techniques for improving the energy efficiency of data-parallel processing. Since none of the related works on heterogeneous systems consider the energy efficiency of data-parallel frameworks, we present works on energy optimizations for homogeneous systems only. We end by presenting the limitations of related work in contrast with the contributions of this thesis.

2.1 Techniques for Data-parallel Processing

In this section, we begin by presenting works targeting homogeneous systems. Afterwards, we present projects on inter- and intra-node heterogeneous systems.

2.1.1 On Homogeneous Systems

In the last decade, MapReduce [33] and Hadoop [14], its open-source implementation, were the most widely used data-parallel frameworks. As explained in Chapter 1 and depicted in Figure 1.1, MapReduce execution consists of four stages, namely, Split, Map, Shuffle and Reduce. Among these four stages, only *Map* and *Reduce* are exposed to application developers. MapReduce is an expressive programming model which, in conjunction with its associated framework, achieves high scalability and fault tolerance. But this programming model is a one-stage, fixed dataflow, batch processing model [66]. Hence, it is not suitable for iterative or interactive jobs [100]. Recently, academia and industry are investigating more flexible and efficient data-parallel programming models [4, 24, 83, 100].

Google has announced Cloud Dataflow [45], a replacement for MapReduce, based on two previous projects, namely, FlumeJava [24] and MillWheel [4]. FlumeJava alleviates the one-stage processing model of MapReduce. Users can define more complex computation pipelines without manually linking MapReduce stages. FlumeJava exposes a series of operators (e.g. *parallelDo*, *groupByKey*, *join*) which can be applied on immutable collections of objects (*PCollections*). At runtime, it employs a lazy evaluation technique by first computing an optimized dataflow execution plan and then starting the actual execution. The employed optimizations are based on node fusion. For example, *MSCR* fusion produces a single stage MapReduce from a series of *paralellDo* and *groupByKey* nodes. With these optimizations, FlumeJava achieves the execution time performance of hand-optimized MapReduce pipelines, but with less programming effort.

The other component of Cloud Dataflow, MillWheel [4], is a framework providing scalable low-latency stream processing. Internally, the processing is represented as a dataflow graph where users provide the code for each node. The data flowing through this graph is represented as a *(key, value, timestamp)* tuple. The timestamp is needed in stream processing to distinguish values arriving at different moments, which can be out-of-order. These timestamps are similar to tagged-tokens or colors in traditional dataflow architectures [20]. Another similarity with traditional dataflow is that application developers can define their one code inside each node of the dataflow graph. MillWheel processing is distributed by assigning a key interval to each cluster node.

More recently, Google has unified batch processing represented by MapReduce [33] and FlumeJava [24], and stream processing represented by MillWheel [4] under a single API named Cloud Dataflow [5]. This API that allows application developers to easily express their data-parallel pipelines has become open source under Apache Beam project [12], where the execution engine is provided by Spark [15, 100, 101] or Flink [13]. With the adoption of Cloud Dataflow, many researchers foresee the end of MapReduce. However, we note that FlumeJava, a key component of Dataflow, internally represents its computations as pipelines of Map and Reduce operators [24]. Moreover, major industry players, including Amazon [6], IBM [58], Microsoft [74] and even Google [46] are providing cloud-based MapReduce platform-as-a-service.

Resilient Distributed Datasets (RDD) [100] and Spark [101] offer a fault-tolerant, in-memory data-parallel processing framework that is able to speedup analytics jobs by one order of magnitude. This framework primarily addresses iterative and interactive programs and achieves an impressive speedup because of keeping the datasets in RAM. The concepts used in this work are similar to those in FlumeJava [24] and Dryad [59]. RDDs are similar to PCollections, but they (i) are mainly in-memory data structures and (ii) can be shared across different queries. The runtime uses a lazy evaluation strategy by first constructing a directed acyclic graph (DAG) of tasks and then distributing these tasks across cluster nodes. However, as opposed to FlumeJava, Spark does not perform optimizations on the DAG. Spark with RDDs can achieve a speedup of up to 40 compared to Hadoop, because of its in-memory processing and its multi-query support. As Spark targets traditional x86-64 cluster systems, it remains to be evaluated how it performs on emerging low-power systems with limited amount of RAM.

Discretized Streams (D-Streams) [102] is a scalable framework that unifies batch and stream processing. It uses the scalability and fault tolerance of data-parallel batch processing frameworks and alleviates their high latency by employing in-memory storage. D-Streams key idea is to execute stream processing as batch jobs in very short time intervals, thus, discretizing the execution. For achieving this, D-Streams uses the RDDs [100] and the Spark runtime [101]. As a result, batch, interactive and stream processing are unified under the same framework. D-Streams achieves a processing latency of less than one second. However, as opposed to MillWheel, users are limited to the operators allowed on RDDs, such as *map*, *groupBy*, *join*. In contrast, MillWheel allows user to define their own code for each node in the dataflow graph.

2.1.2 On Heterogeneous Systems

2.1.2.1 On Inter-node Heterogeneous Systems

Zaharia et al. [103] are the first to investigate Hadoop performance on heterogeneous clusters. They propose Longest Approximate Time to End (LATE) scheduling algorithm which halves Hadoop execution time in heterogeneous clusters of virtual machines. This work is motivated by the fact that Hadoop assumes all nodes in a cluster to be homogeneous. However, heterogeneity may be present in clusters of systems with different loads or belonging to different generations. Many real world clusters deploy multiple virtual machines on top of a single physical system, thus, introducing more degrees of heterogeneity. In this environment, some nodes, called stragglers, execute their task very slowly and delay the entire MapReduce execution. LATE manages these stragglers in a more robust way than Hadoop's default scheduler. However, the definition of heterogeneity used by LATE is slightly different from ours. We consider nodes with different processing units and different PPRs as being heterogeneous, but in LATE evaluation heterogeneity and straggler effect are artificially obtained by executing additional CPU or I/O intensive workloads to slow down the systems.

Ahmad et al. [2] investigate Hadoop MapReduce execution on heterogeneous clusters with high-performance Intel Xeon and low-power Intel Atom CPUs. This work is motivated by the slower Hadoop execution on a heterogeneous cluster compared to that on either of the homogeneous clusters. The authors identify two reasons for this behavior: (i) high network utilization during Map phase due to Hadoop load balancer which steals work from slower low-power nodes and (ii) the imbalance in the Reduce phase due to equally distributed work among heterogeneous nodes with different compute capabilities. In order to alleviate these issues, the authors propose three optimizations: a communication-aware load balancer (CLAB) for Map phase, a communication-aware scheduler for Map phase (CAS) and a predictive load balancer for Reduce phase (PLB). Firstly, CALB decides if Map or Shuffle is on the critical path. Secondly, CAS decides how many remote Map tasks should be created and how should they be executed. Lastly, PLB splits the key space into smaller chunks and assigns them to nodes based on their processing capabilities. These optimizations are implemented in Hadoop and evaluated on a cluster with 10 Xeon nodes and 80 Atom nodes. The average speedup is 1.7 when compared to Hadoop and 1.4 when compared to LATE.

2.1.2.2 On Intra-node Heterogeneous Systems

We show in Table 2.1 related works that propose MapReduce frameworks for intra-node and intra-chip heterogeneous systems. Among the first MapReduce frameworks targeting intra-node heterogeneous systems with GPUs, Mars [52] contributes with the design and implementation of MapReduce for Nvidia GPUs. In addition, the authors developed a string processing library for GPUs that achieves a speedup of up to nine compared to standard C/C++ string processing on CPUs. Compared to Phoenix [82], which is a state-of-the-art, CPU-only, C++ implementation of MapReduce, Mars achieves a maximum speedup of 16. However, Mars has some limitations. Even though Mars supports both map and reduce phases on GPU, only one of the six analyzed benchmarks has a reduce phase. Secondly, Mars has a shared-memory implementation that does not support cluster execution. Nevertheless, Mars represents a breakthrough and was followed by a series of works proposing improvements for MapReduce execution on single-node systems with GPUs [26, 36, 57].

MapCG [57] adopts the idea of writing a single version of a program using MapReduce model and running it efficiently on both the CPU and GPU. This

Table 2.1: Related work on heterogeneous systems with GPU

Work	Year	Multi-node	Hadoop-based	GPU programming
Mars [52]	2008	no	no	CUDA
Merge [68]	2008	no	no	EXOCHI
MITHRA [38]	2009	yes	yes	CUDA
Shirahata at al. [84]	2010	yes	yes	CUDA
MapCG [57]	2010	no	no	CUDA
StreamMR [36]	2011	no	no	OpenCL
Ji et al. [60]	2011	no	no	CUDA
GPMR [86]	2011	yes	no	CUDA
Chen et al. [26]	2012	no	no	CUDA
Chen et al. [27]	2012	no	no	OpenCL
HadoopCL [48]	2013	yes	yes	Java/OpenCL
Glasswing [35]	2014	yes	no	OpenCL
Hadoop+ [53]	2015	yes	yes	CUDA/OpenCL

single-node framework improves on Mars by avoiding redundant counting phases with the help of a light-weight memory allocator and by replacing sorting of intermediate keys with a custom hash table approach on GPUs.

Targeting systems with AMD GPUs, StreamMR [36] is a single-node MapReduce implementation that uses lock-free and atomic-free data structures to improve GPU execution. Similar to MapCG [57], StreamMR implements a hash table for intermediate <*key, value*> storage, and, thus, achieves better performance than Mars [52] which uses a sorting mechanism.

Merge [68] proposes a MapReduce-based programming model and applies the idea of distributing work in the form of specialized code for different heterogeneous resources. While this idea is similar to ours, implementing new workloads into Merge is time-consuming. Moreover, Merge implementation is based on Intel compiler and Threading Building Blocks (TBB) which reduce its portability. Similar to us, the authors discovered that overlapping CPU and GPU does not speedup the execution.

Ji et al. [60] show that MapReduce execution on GPUs can be improved by exploiting the small but fast shared memory. They use this shared memory as a buffer for map and reduce input and output data to increase data transfer efficiency. Chen et al. [26] use the same shared memory to implement the immediate reduction of a <*key, value*> pair emitted by Map phase. In contrast, we use shared memory to keep additional data needed by `map` function, such as the string to be searched for Grep or the initial cluster centroids for Kmeans.

Motivated by the emergence of integrated CPU and GPU chips, Chen et al. [27] have designed a MapReduce framework targeting AMD Fusion architecture. In order to utilize both the CPU and GPU, the authors use two different approaches: (i) a map-dividing approach in which both devices run map and reduce tasks and (ii) a pipelining approach in which one device executes map tasks and the other runs the reduce tasks. Using scheduling and tuning techniques, their CPU-GPU framework achieves a maximum speedup of 2 compared to the best of CPU- and GPU-only approach. In contrast, we show that overlapping CPU and GPU does not improve the execution.

Motivated by the lack of stand-alone MapReduce frameworks targeting clusters of heterogeneous systems with GPUs, Stuart et al. introduced GPMR [86]. In addition to the framework design and implementation, the authors highlight a series of optimization techniques for implementing MapReduce on GPUs, such as partial reductions and accumulation of <*key, value*> pairs on GPU. GPMR is evaluated on a cluster with 64 Nvidia Tesla GPUs and shows poor scalability when running on more than eight GPUs. The other stand-alone MapReduce framework for clusters, Glasswing [35] relies on a pipeline with five stages to overlap communication and computation in map and reduce phases. In order to provide vertical scaling using heterogeneous core inside each node, Glasswing exposes a MapReduce-style OpenCL API. While these frameworks achieve significant performance, they are impractical to use in real-world deployment due to lack of fault tolerance. Moreover, cluster-level input and output data management is fuzzy.

With Hadoop being the most employed MapReduce framework, few works extend it for clusters of heterogeneous systems with GPUs. MITHRA [38] is among the first works that combine Hadoop and CUDA to improve the execution of embarrassingly parallel applications on clusters with GPUs. Their implementation of BlackScholes pricing model on four nodes with GPU achieves better execution time compared to a cluster of 62 CPU-only nodes.

Shirahata et al. [84] propose first hybrid CPU-GPU Map task implementation in Hadoop. They implement the Map phase of Kmeans clustering algorithm in C++ and CUDA and integrate it on Hadoop through Pipes mechanism. In order to efficiently schedule Map tasks on both CPU and GPU, they propose to solve a minimization problem for task execution. The input parameters for this problem, such as map execution time and GPU acceleration, are obtained after profiling map execution. Using this proposed scheduling technique, the maximum speedup over Hadoop is 1.93 obtained on 64 nodes with GPUs. However, these works analyze only one application and do not present an evaluation of the energy usage.

Aiming to leverage the programmability of heterogeneous systems in general, HadoopCL [48] integrates Hadoop and OpenCL by automatically generating OpenCL code from Java using APARAPI tool. But this translation tool supports only a subset of Java types and library methods and hence limits the usability of HadoopCL. Even if energy efficiency is one of the main motivations for this work, the authors mention that no energy usage results are presented due to the lack of infrastructure to measure power and energy.

Hadoop+ [53] is another Hadoop-based framework that enables concurrent CPU and GPU MapReduce execution. Additionally, a simple model is proposed to help users select the most cost-effective system configuration. However, Hadoop+ and its presentation have some limitations. Firstly, Reduce phase execution on the GPU is not clear. Based on our experience, Reduce phase on the GPU requires additional mechanisms to collect intermediate results and to process them using reductions. These mechanisms require changing CPU program structure to adapt it for GPU execution. Secondly, the model assumes that only one GPU task runs on the GPU at one time. It is not clear how many GPU threads are used for this task. GPU configuration space consisting of number of thread blocks and number of threads per block is not explored. In contrast, we perform both an empirical and a formal analysis of GPU configuration space.

In the end, we present non-MapReduce data-parallel frameworks on intra-node heterogeneous systems. Dandelion [83] is a data-parallel processing framework targeting small and medium clusters of heterogeneous CPU-GPU systems. Users write sequential C# code and Dandelion compiler and runtime represent it as a multi-level dataflow graph. This representation comprises three dataflow graphs: cluster level, multicore system level and GPU level graph. These three graphs are executed on three runtime dataflow engines residing at each system level. Firstly, cluster dataflow engine implements the same principles (e.g. in-memory caching, checkpointing) as other data-parallel frameworks [24, 59, 100]. Secondly, the multicore system dataflow engine handles the parallelization among CPU cores and the off-loading of some processing to the GPU. Thirdly, GPU dataflow engine uses the tag-tokens model to handle iterative and stream processing. As side contributions, this work develops a compiler from C# to CUDA code and a GPU dataflow engine as a library. Dandelion achieves a mean speedup of 6 on a single system when compared to sequential C# code. On a 10-node cluster with GPUs, it exhibits a speedup of 2 compared with Dryad framework [59]. However, even if energy efficiency is cited as a motivation, it is not discussed in the evaluation.

With the exponential increase in data to be analyzed, energy usage becomes an issue. Even if some of the presented works are motivated by the potential energy efficiency of nodes with GPUs [48, 84], none of them include an energy analysis.

Hence, it is not clear if GPU improves the energy usage of a system running MapReduce workloads. In contrast, we present an analysis of low-power systems with GPUs which shows that GPU improves the energy efficiency of compute-intensive workloads.

2.2 Performance Models for Data-parallel Processing

With the adoption of MapReduce and Hadoop on both cloud and self-hosted clusters, there is a higher demand to optimize software parameters and system resources. It is well-known that manually tuning Hadoop is a daunting task because the framework has many software tuning parameters [55]. It is equally challenging to select the most suitable system resources in terms of node type, node settings, and number of nodes [54]. In this context, there are many research works that propose MapReduce performance models to predict the execution time of scale-out workloads and to select suitable software and hardware configurations. We summarize these related works in Table 2.2 and present them bellow.

Elastisizer [54] provides an automated approach to configure cloud cluster sizes and Hadoop framework parameters. To predict Hadoop execution, it employs analytic, Machine Learning and simulation techniques. However, its average validation error is relatively high, at 20%. Elastisizer is based on Starfish [55], an approach to model Hadoop execution in detail using an instrumentation-based fine-grain profiling of MapReduce phases. This profiling target MapReduce subphases, such as reading the input, map, collect, spill, merge, shuffle, reduce and writing to the output. Both Elastisizer and Starfish overestimate job execution time due to high profiling overhead. In contrast, our profiling targets only main phases such as Map, Shuffle and Reduce to achieve higher accuracy at lower pro-

Table 2.2: MapReduce performance models

Work	**Year**	**Model type**	**Model error [%]**	**Validation configurations**
Elastisizer [54]	2011	Analytic/ML	20	24
ARIA [93]	2011	Analytic	10	108
Verma et al. [94]	2011	Analytic	10	113
Tian et al. [88]	2011	Analytic	9-19	100-240
Grey-box [62]	2012	ML	12	12
HP [104]	2013	Analytic	10	24
Zhang et al. [105]	2013	Analytic	10	26
ARIEL [95]	2014	Analytic	10	12
CRESP [25]	2014	Analytic/ML	5-16	360-640
HP+ [64]	2016	Analytic	5	40

filing overhead.

ARIA [93] and subsequent works [94, 95, 104, 105], propose a modeling approach to determine lower and upper bounds for Hadoop's execution time using a "makespan theorem". The approach consists of three steps, (i) job profiling, (ii) job modeling and (iii) job scheduling or capacity planning. Job profiling is done in-depth, at sub-phase level, similar to Starfish approach, and in contrast with our profiling of main execution phases. Improving over these works, HP+ [64] employs a Locally Weighted Linear Regression (LWLR) technique resulting in very high modeling accuracy. However, the model is validated only on two workloads, Wordcount and Sort, on a total of 40 system configurations.

Tian et al. [88] propose a simple regression-based model which is parametrized using baseline runs on small inputs and is validated using four applications and up to 60 configurations for each application. Improving over this work, CRESP [25] employs a hybrid analytic and Machine Learning (ML) approach to increase accuracy. One limitation of these works is that Shuffle phase is not modeled. Similar to us, they validate the model on both in-house and cloud-based clusters.

Kadirvel et al. [62] propose a grey-box approach based on Machine Learning for modeling MapReduce execution. This grey-box approach uses low-level system information similar to white-box analytic modeling to improve the accuracy of black-box Machine Learning techniques that are system-agnostic. In this work, the authors evaluate 20 learning techniques and select four that achieve an average accuracy of 12%, comparable to the accuracy of our measurement-driven analytic models.

While all previous works model execution time and determine suitable cluster sizes and framework parameters, none of them is modeling or analyzing energy usage. To the best of our knowledge, we are the first to (i) provide an energy usage model for Hadoop MapReduce execution and (ii) use the time-energy models to analyze wimpy in contrast with brawny nodes and heterogeneous in contrast with homogeneous clusters [70].

2.3 Energy Efficiency of Data-parallel Processing

Since previous works of MapReduce and Hadoop on heterogeneous systems do not address the issue of energy usage, we present projects on the energy efficiency of Hadoop on traditional homogeneous systems. These works include two techniques for shutting-down systems during low-utilization periods. These techniques are called Covering Set (CS) [67] and All-In Strategy (AIS) [65]. CS shuts-down all the nodes in the cluster during low utilization intervals, except a small set (the Covering Set) of nodes which store at least one replica of each HDFS block. On the other hand, AIS claims that is more energy-efficient to use all the nodes and, thus, finish MapReduce jobs faster and shut-down all nodes afterwards.

Berkeley Energy Efficient MapReduce (BEEMR) [28], proposes to split a MapReduce cluster into interactive and batch zones. The nodes in batch zone are kept in a low-power state when inactive. This technique is based on the insights from MapReduce with Interactive Analysis (MIA) workloads. For this kind of workloads, interactive MapReduce jobs tend to access only a small part of data. Hence, an interactive cluster zone can be obtained by identifying these interactive jobs and their required input data. The rest of the jobs are executed on the batch zone at defined time intervals. Using both simulation and validation on Amazon EC2, BEEMR reports energy savings of up to 50%.

Feller et al. [39] studied the performance and power consumption of Hadoop on clusters with collocated and separated data and compute nodes. They highlight two unsurprising findings: (i) the PPR of collocated data and compute nodes is better compared to a separated deployment and (ii) power is dependent on MapReduce phases.

With an environmental-friendly approach, GreenHadoop [44] improves Hadoop scheduling to minimize energy usage in a datacenter power by both solar energy and conventional grid-based energy. While solar energy availability is estimated using a model, MapReduce energy requirements are estimated using historical energy measurements per job. However, this approach assumes that jobs with similar characteristics are run over time in the datacenter. Our measurement- and model-driven approaches are orthogonal to this work and could be used to improve the performance of GreenHadoop's scheduling algorithm.

2.4 Summary and Limitations

As data increases to petabytes and clusters scale to thousands of nodes, there is a need for scalable and fault-tolerant frameworks such as Google Cloud Dataflow and MapReduce. These frameworks are designed for homogeneous brawny server systems that incur high energy usage [66]. At the same time, heterogeneous systems with GPU are increasingly being used in datacenters [89] and low-power wimpy heterogeneous systems are to enter server market [8]. But there is a lack of

Table 2.3: Related work summary

Related Work	**This Thesis**
Intra-node heterogeneous systems [26, 27, 35, 38, 52, 53, 57, 60, 83, 84, 86]	1. intra-node and intra-chip heterogeneous systems with GPU 2. wimpy heterogeneous systems with GPU and with many-core CPU
Energy efficiency [28, 39, 65, 67]	3. time-energy efficiency 4. time-energy modeling

studies on the time-energy performance of batch data-processing on wimpy heterogeneous systems, as shown in Figure 2.1 and Table 2.3. Thus, our aim is to analyze the time-energy performance of data-parallel applications on these heterogeneous platforms, and to propose techniques to improve the time-energy performance. In this thesis, we focus on wimpy heterogeneous systems with both discrete and integrated GPUs, such as Kayla [30] and Jetson TK1 [75] from Nvidia, and on wimpy heterogeneous systems with many-core CPUs represented by ARM big.LITTLE architecture [19].

In the contemporary context of green computing, related work suffers from the lack of energy efficiency evaluation. Even among recently proposed data-parallel processing frameworks, there are few works that consider energy efficiency, as discussed in Section 2.3. We believe that energy-efficient execution can be achieved using adequate techniques for mapping data-parallel tasks on heterogeneous execution units. We address this by developing such techniques and by conducting studies on the energy performance of data-parallel applications on heterogeneous systems.

Chapter 3

Techniques for Efficient Data-parallel Execution

This chapter presents our general approach, followed by the proposed techniques for efficient data-parallel execution on heterogeneous systems with GPU.

3.1 Approach Overview

Given data-parallel applications, such as Google Cloud Dataflow or MapReduce applications, our approach targeting intra-node and intra-chip heterogeneous systems consists of three parts, as depicted in Figure 3.1.

1. To enable data-parallel application processing on heterogeneous systems with accelerators, we design *techniques for efficient data-parallel execution* on heterogeneous systems with GPUs and implement them in MoSS, our Hadoop-CUDA framework. Firstly, our *lazy processing* technique enables the processing of multiple input records on a GPU, in contrast with chunking which divides a single record among GPU threads. Secondly, we propose three *dynamic mapping* techniques to overlap or select the most suitable

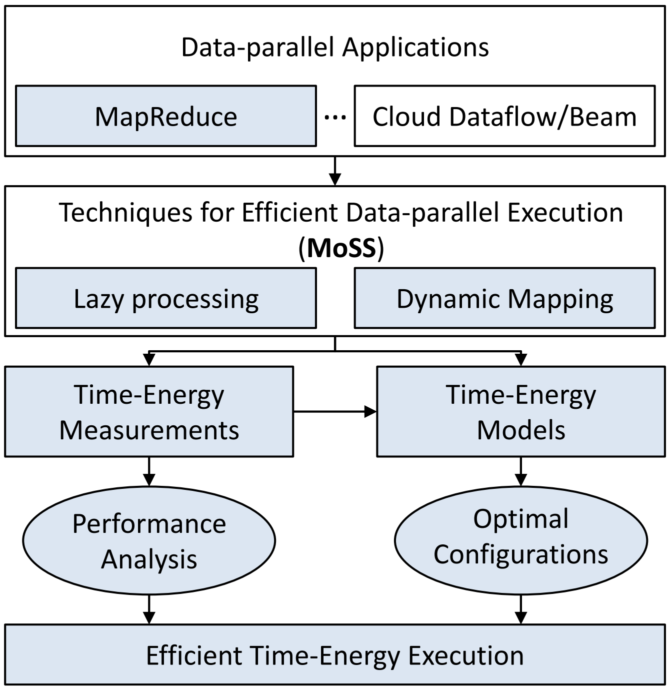


Figure 3.1: Approach

processing unit between the CPU and GPU at runtime.

2. To evaluate the time-energy performance of brawny and wimpy intra-node and intra-chip heterogeneous systems, we perform a novel *measurement-driven time-energy performance analysis* to expose issues and bottlenecks. Our performance analysis is performed on both intra-chip heterogeneous systems with many-core CPUs, such as ARM big.LITTLE, and intra-node heterogeneous systems with GPUs, such as servers with discrete GPUs or systems with integrated GPUs. We show that heterogeneity is more time-energy-efficient for compute-intensive applications. Moreover, we compare brawny and wimpy heterogeneous systems and derive performance equivalence ratios that lead to energy savings on wimpy heterogeneous clusters.

3. To determine efficient configurations for large scale deployments, we design *time-energy performance models* for data-parallel execution on heterogeneous systems using insights from our measurement-driven analysis. Using

workload and system parameters, and baseline runs to capture the runtime behavior, we apply the models to derive optimal system configurations. In addition, the models reveal bottlenecks in both the application and the platform, hence, offering useful insights to both application developers and hardware designers.

Next, we discuss each of these three parts, while the details are presented in Chapter 3, Chapter 4 and Chapter 5, respectively.

Techniques for Efficient Data-parallel Execution. With heterogeneous systems becoming the norm, our aim is to efficiently execute batch data-parallel applications, such as MapReduce applications, on heterogeneous systems with accelerators, while maintaining their program logic structure. Since heterogeneous systems incorporate processing units of different architecture, the same software approach may not achieve good performance on all these units. For example, a modern CPU has MIMD architecture, while a GPU has SIMD architecture being able to run the same program on multiple data at a time. At intra-node memory level, these processing units may share the same memory, as in systems with

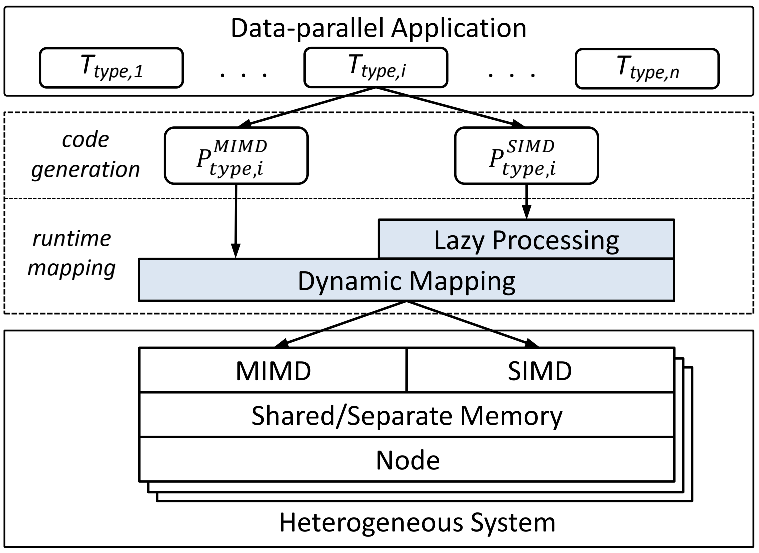


Figure 3.2: Techniques for data-parallel execution on heterogeneous systems

integrated GPU, or have separate memories such as systems with discrete GPU. Given an application which is statically split into tasks of different types ($T_{type,i}$), our high-level approach consists of (i) a static step when binary code is generated for different processing units having different Instruction Set Architecture (ISA) and (ii) a dynamic step consisting of runtime mapping techniques, as depicted in Figure 3.2. In the dynamic step, multiple task-instances are spawned during program execution, and one or multiple instances are distributed to processing units based on their architecture. Inside each task-instance, input records are processed differently based on the processing unit. On a CPU core, only one record is processed at a moment, while on a GPU, multiple records can be processed at the same time. To enable this execution of multiple records on the GPU, we design a *lazy processing* technique that is presented in detail in Section 3.2. However, the GPU may not be the most suitable processing unit for all types of applications, as motivated in the Introduction. Thus, we present in Section 3.3 the design of dynamic mapping techniques to select the most suitable processing unit or to overlap the processing on multiple processing units. We implement these techniques for heterogeneous systems with Nvidia GPUs in MoSS using Hadoop [14] and Nvidia'a CUDA programming model [76]. MoSS implementation is discussed in Section 3.4, while the API and a programming example are presented in Appendix C.

Measurement-driven Performance Analysis. With the proliferation of both heterogeneous systems and wimpy, low-power systems, we aim to answer two fundamental research questions. Firstly, are heterogeneous systems more time-energy-efficient than homogeneous, CPU-only systems, for data-parallel processing? Secondly, are wimpy systems more energy-efficient than brawny systems when performing data-parallel processing? To answer these questions, we perform an in-depth measurement-driven analysis of the execution time and power usage of

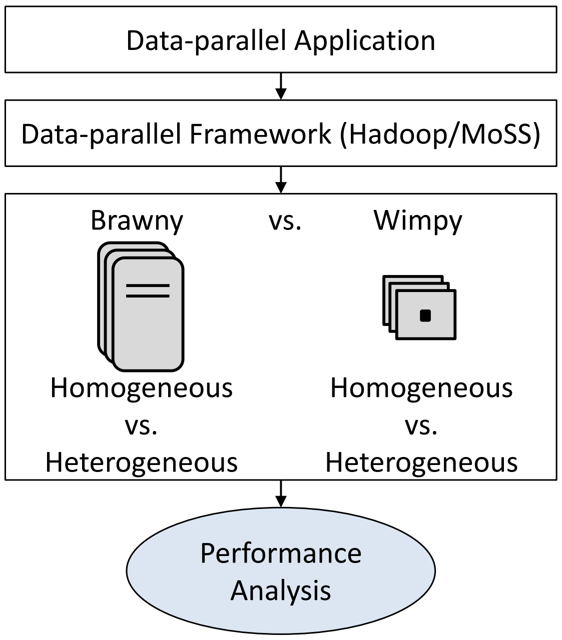

Data-parallel Application
Data-parallel Framework (Hadoop/MoSS)
Brawny
vs.
Wimpy
Homogeneous vs. Heterogeneous
Homogeneous vs. Heterogeneous
Performance Analysis

Figure 3.3: Measurements-driven time-energy performance analysis

heterogeneous systems with GPU at single-node and cluster level using MoSS and Hadoop, as depicted in Figure 3.3. This measurement-driven analysis is presented in detail in Chapter 4. For this analysis, we have selected four representative systems covering self-hosted and cloud-based brawny nodes with discrete Nvidia GPUs and x86-64 Intel/AMD CPUs, and self-hosted wimpy nodes with discrete and integrated Nvidia GPUs and ARM CPUs. A detailed characterization of the systems used in our measurement-driven analysis is presented in Appendix A. In addition to heterogeneous systems with GPUs, we study the time, energy and cost performance of many-core heterogeneous wimpy systems represented by ARM big.LITTLE [19], in comparison with traditional Xeon-based server systems for data-parallel processing. This performance study [72] is presented in detail in Appendix B.

Model-driven Performance Analysis. While measurement-driven analysis is tractable for relatively small workloads and small clusters, to analyze scale-out workloads and clusters we develop execution time and energy usage analytic

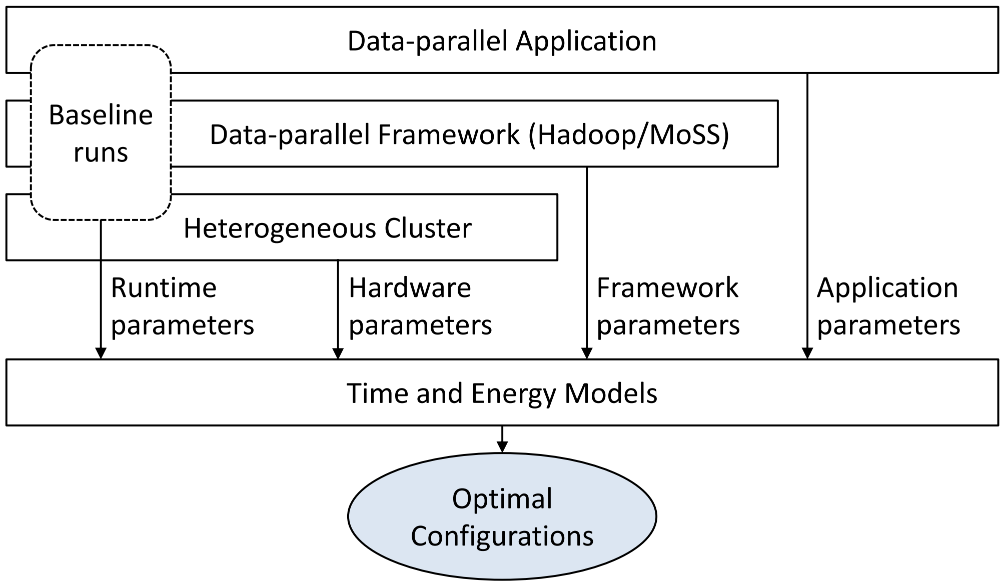


Figure 3.4: Modeling the time-energy performance of data-parallel processing

models based on measured parameters. Moreover, we use these models to analyze hypothetical systems, a task that is impossible using only measurements. Our execution time and energy models take input parameters from application level, such as input size, framework level, such as input split size, and systems level, such as number of cluster nodes, as depicted in Figure 3.4. To increase accuracy, our models are parametrized using measurements from baseline runs of small versions of the workload on a single cluster node. These baseline runs determine application characteristics such as Map phase output profile, Map and Reduce processing time of one record and utilization and power consumption during each MapReduce phase. We present the models and the model-based analysis in Chapter 5.

In the remainder of this section, we present our proposed techniques for efficient data-parallel execution on heterogeneous systems and their implementation in MoSS using Hadoop and CUDA.

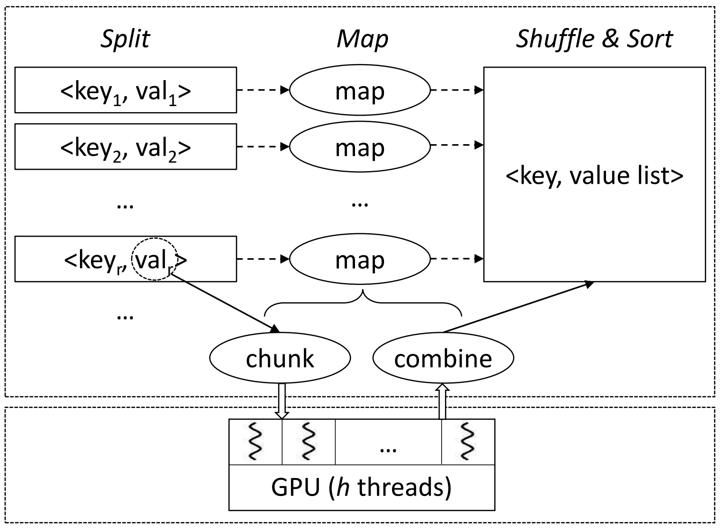
3.2 Lazy Processing

As stated in Section 1.1 of the Introduction, we classify data-parallel processing into (i) batch data-parallel processing and (ii) stream data-parallel processing. In batch data-parallel processing, the entire input data is known *a priori* and data operators can be applied to input records at the same time, in parallel. However, this approach requires a number of processing units equal to the number of input records. In practice, the input is split into chunks, and each chunk is handled by a parallel task-instance. Multiple task-instances are processed in parallel by the available parallel processing units. Depending on their architecture, processing units may process chunked records sequentially, one-by-one, or in parallel, multiple-at-a-time.

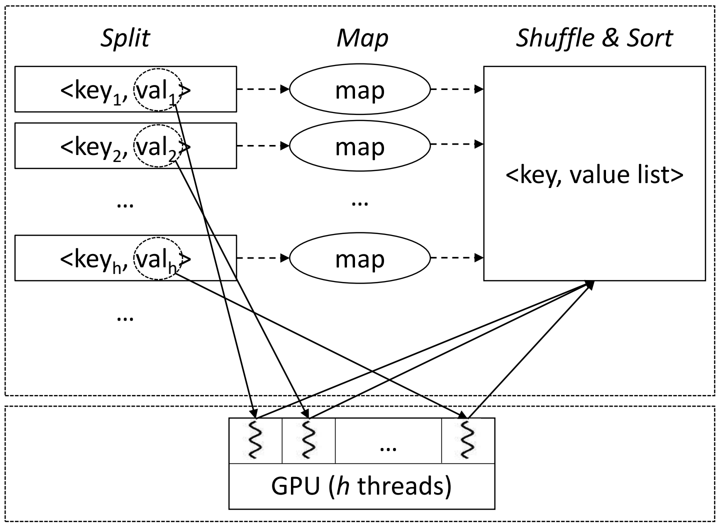
In typical data-parallel frameworks [11,14,33,101], multiple task-instances are spawned during runtime and each instance processes its data records sequentially. This sequential processing is suitable for CPU cores, but the GPU requires a different approach. One alternative implemented in related work [84] is to further chunk each record's value and process these chunks in parallel on the GPU, as depicted in Figure 3.5a. The related article [84] does not explicitly describe this *chunking* approach, since its focus is on scheduling Map tasks on CPUs and GPUs. However, we have analyzed the related source code[1] implementing Kmeans application, and we attribute the name “chunking” to this approach.

We further describe chunking technique applied on Kmeans and later contrast it with our *lazy processing* approach. Kmeans is a data analytics application that groups n points with m dimensions into k clusters based on a distance metric. Each cluster has a virtual center point, called centroid. Starting from k centroids, the application determines the closest centroid for each point and assigns the point

[1]Source code is available at `https://github.com/koichi626/hadoop-gpu`



(a) Chunking



(b) Lazy Processing

Figure 3.5: Comparison between chunking and lazy processing

Algorithm 3.1 Kmeans using chunking

```
 1: # Data processing (runs on GPU)
 2: function ComputeClustersOnGPU($h, k, n, centroids, points$)
 3:     $tid \leftarrow get\_thread\_id(1, h)$
 4:     for each $point$ in $points$ assigned to thread $tid$ do
 5:         $cluster\_id \leftarrow get\_closest\_cluster(k, centroids, point)$
 6:         assign $cluster\_id$ to $point$
 7:     end for
 8: end function
 9: function ComputeNewCentroids($h, k, n, new\_centroids, points$)
10:     $tid \leftarrow get\_thread\_id(1, h)$
11:     $new\_centroids[tid] \leftarrow compute\_new\_centroids(tid, points)$
12: end function
13:
14: # Map using "chunking" (runs on CPU)
15: function Chunking
16:     $< key, value >\leftarrow get\_input\_record()$
17:     $k \leftarrow parse\_k(value)$
18:     $n \leftarrow parse\_n(value)$
19:     $centroids \leftarrow parse\_centroids(value)$
20:     $points \leftarrow parse\_points(value)$
21:     $new\_centroids \leftarrow [0, \dots, 0]$
22:     $iter \leftarrow 1$
23:     while $(new\_centroids \neq centroids) \wedge (iter \leq 100)$ do
24:         transfer $centroids$ and $points$ from CPU to GPU
25:         ComputeClustersOnGPU($h, k, n, centroids, points$)
26:         transfer $points$ with cluster assignment from GPU to CPU
27:         sort $points$ based on cluster assignment
28:         transfer $points$ from CPU to GPU
29:         ComputeNewCentroids($k, k, n, new\_centroids, points$)
30:         transfer $new\_centroids$ from GPU to CPU
31:         $iter \leftarrow iter + 1$
32:     end while
33:     emit $< key, new\_centroids >$
34: end function
```

to the cluster with the closest centroid. At the end, new centroids are computed for each cluster, based on the assigned points. These steps are repeated until the clustering is stable, that is, two consecutive iterations produce the same clusters and centroids, or until the application exceeds a certain number of iterations.

Typical MapReduce Kmeans programs implement a single iteration of the algorithm described above [2], where an input record represents a point with m dimensions. However, in [84], an input record represents an instance of Kmeans, with k centroids and n 2D points. Map function gets an input record, parses the points, runs the Kmeans steps and emits k centroids, as presented in Algorithm 3.1. During each Kmeans iteration, the GPU is used firstly to compute the closest centroid for each point to determine cluster allocation and, secondly, to compute the new centroids for the k clusters. On the first GPU invocation, the points are split into chunks and each GPU thread processes one chunk. On the second invocation, there are k GPU threads where one thread computes the new centroid for one cluster. Using this approach, the user must explicitly split or chunk the input and combine the results for the final output, as shown in Figure 3.5a. Moreover, this approach is suitable for large input records [84], but in practice workloads have small input records [29, 63].

In contrast with chunking, we propose a *lazy processing* technique, as depicted in Figure 3.5b. Given a GPU with g cores, we select a thread count value h larger than g to achieve high occupancy on the GPU. The *lazy processing* buffers h records and sends them to the GPU for processing at the same time. Contrary to chunking where the record is immediately split and processed, our approach is "lazy" as it waits for h records before beginning the processing. A high-level form of the lazy processing technique is presented in Algorithm 3.2. The value of h is empirically determined using small input sizes based on GPU capabilities, as described in Section 4.3.1. After record processing on the GPU, the results are

Algorithm 3.2 Lazy processing

```
1: # Data processing (runs on GPU)
2: function ProcessOnGPU($h$)
3:     process buffered records using $h$ GPU threads
4: end function
5:
6: # Lazy Processing (runs on CPU)
7: function LazyProcessing($r$, $h$)
8:     while $r > 0$ do
9:         $cnt \leftarrow 0$
10:        while $cnt < h$ do
11:            buffer current record
12:            $cnt \leftarrow cnt + 1$
13:        end while
14:        transfer buffered records from CPU to GPU
15:        ProcessOnGPU($h$)
16:        transfer the results from GPU to CPU
17:        emit the results using the CPU
18:        $r \leftarrow r - h$
19:    end while
20: end function
```

sent to the Shuffle phase using the CPU. This result outputting cannot be done on the GPU because it requires disk and networking access.

We present Kmeans using *lazy processing* in Algorithm 3.3. The entire processing of an input record is done on the GPU, while the CPU handles record buffering, CPU-GPU data transfers and the output of the results. The user can focus on program logic by writing the GPU code without thinking of how to split the data or how to combine the results.

While our approach can be applied in general to batch data-parallel processing, we describe it in detail for MapReduce, one of the most popular data-parallel processing frameworks. Given a MapReduce application that is executed on an intra-node heterogeneous system with CPU and GPU, the first step consists of writing Map and Reduce code for both the CPU and GPU. Since our aim is to retain the program logic of MapReduce application, GPU code is similar to CPU

Algorithm 3.3 Kmeans using lazy processing

```
# Data processing (runs on GPU)
function ProcessOnGPU($h$)
    $tid \leftarrow get\_thread\_id(1, h)$
    $< key, value > \leftarrow get\_input\_record(tid)$
    $k \leftarrow parse\_k(value)$
    $n \leftarrow parse\_n(value)$
    $centroids \leftarrow parse\_centroids(value)$
    $points \leftarrow parse\_points(value)$
    $new\_centroids \leftarrow [0, \ldots, 0]$
    $iter \leftarrow 1$
    while $(new\_centroids \neq centroids) \wedge (iter \leq 100)$ do
        for each $point$ in $points$ do
            $cluster\_id \leftarrow get\_closest\_cluster(k, centroids, point)$
            assign $cluster\_id$ to $point$
        end for
        sort $points$ based on cluster assignment
        for each cluster $j$ from 1 to $k$ do
            $new\_centroids[j] \leftarrow compute\_new\_centroids(j, points)$
        end for
        $iter \leftarrow iter + 1$
    end while
    emit $< key, new\_centroids >$
end function

# Map using "lazy processing" (runs on CPU)
function LazyProcessing($r$, $h$)
    while $r > 0$ do
        $cnt \leftarrow 0$
        while $cnt < h$ do
            buffer current record
            $cnt \leftarrow cnt + 1$
        end while
        transfer buffered records from CPU to GPU
        ProcessOnGPU($h$)
        transfer the results from GPU to CPU
        emit the results using the CPU
        $r \leftarrow r - h$
    end while
end function
```

code except for replacing Hadoop API calls with our GPU API. In the following section, we describe the second step consisting of techniques for efficient data-parallel execution.

3.3 Dynamic Mapping

After applying the *lazy processing*, tasks are dynamically mapped on the CPU or GPU. There are two approaches to perform this mapping. One approach is to overlap CPU and GPU execution, while the other approach is to select only the CPU or the GPU for processing. Moreover, there are multiple ways to overlap CPU and GPU execution. Shirahata et al. [84] overlap CPU and GPU tasks on one cluster node based on a linear optimization scheduling technique. MapCG [57] supports overlapped CPU and GPU execution but argues that it is less efficient compared to single-unit execution since different applications have different architectural requirements. HadoopCL [48] overlaps data transfers with data processing. Data transfers are handled by the CPU, while data processing is done on the GPU using asynchronous methods.

We have design, implemented and evaluated both alternatives for dynamic mapping of data-parallel processing on heterogeneous systems with GPU. First, we present a *one-time* workload profiling approach for selecting the most suitable processing unit between the CPU and GPU, as listed in Algorithm 3.4. The workload profiling is performed for the first h records on both processing units.

Algorithm 3.4 Non-overlapping task execution: one-time approach

```
profile first $h$ records on both CPU and GPU
if $T_{GPU} < T_{CPU}$ then
    map remaining records on GPU using lazy processing
else
    map remaining records on CPU
end if
```

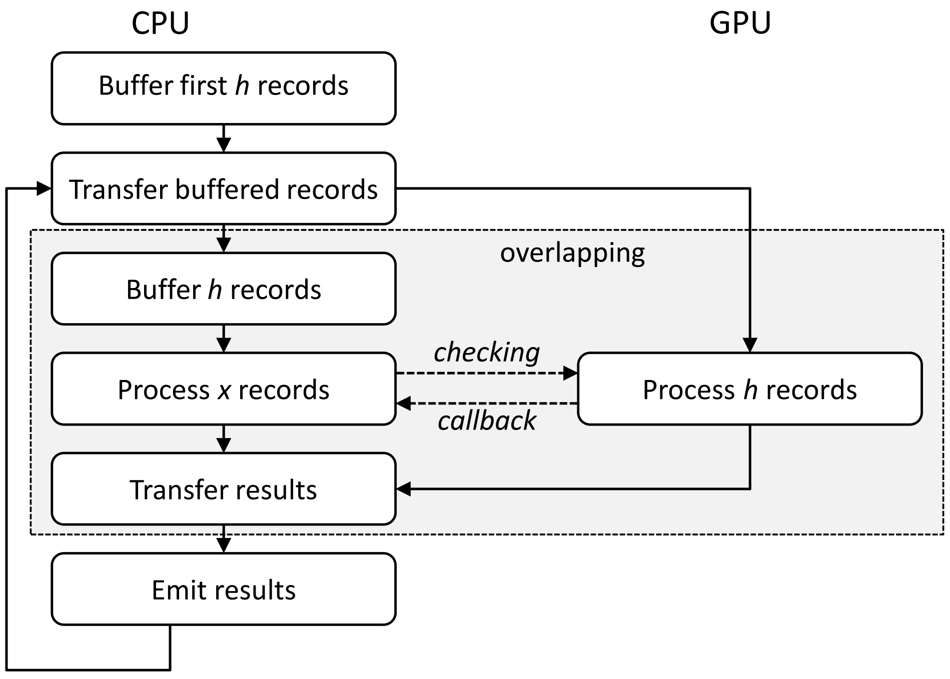


Figure 3.6: Overlapping execution: checking and callback

Based on this profiling, if the execution time on GPU is smaller compared to CPU, the remaining tasks are executed on GPU using *lazy processing*. Otherwise, the remaining tasks are executed on CPU.

Secondly, the processing of h records on GPU can be overlapped with the processing of a variable number of records, x, on CPU and with the buffering of next h records for GPU, as depicted in Figure 3.6. Emitting the results cannot be overlapped because only the CPU can handle this step that requires system I/O access. We design and implement two techniques to perform this overlapping:

- *checking* technique where the CPU checks for GPU processing completion after one or multiple records are processed by the CPU
- *callback* technique is based on asynchronous events. The CPU registers a callback through which the GPU signals the completion of its processing.

Counter to intuition, we observe that the *one-time* approach achieves better time-energy performance compared to overlapping approaches. Intuitively, partitioning the work and overlapping CPU and GPU processing should lead to more work done per unit time. But the CPU needs to handle storage and networking access in addition to task processing. Thus, overlapping may not be the best option when there is high performance imbalance between the CPU and GPU. In Chapter 4, Section 4.3.3, we present the experimental results showing that *one-time* performs better in most of the cases. A formal analysis using our execution time model is presented in Chapter 5, Section 5.4.2.

3.4 Implementation of MoSS

To evaluate our techniques, we implemented them using the widely-used open source Hadoop framework and Nvidia CUDA for GPU. We name our framework **MoSS**, **M**apReduce **o**n Heterogeneou**S** **S**ystems. To run generated native code on Java-based Hadoop framework, we use the Hadoop Pipes mechanism. The execution flow of MapReduce applications in MoSS is depicted in Figure 3.7. Given a MapReduce application written in C/C++, the developer adds Map task implementation in CUDA by replacing the API calls with MoSS counterparts. The application is then compiled with the CUDA compiler, *nvcc*, and linked with our MoSS library. The resulting binary is executed on Hadoop using Pipes mechanism.

MoSS provides an API to facilitate application porting for GPU execution. Listing 3.5 introduces a subset of MoSS API. Listed functions are frequently used in MapReduce CUDA kernels. For example, `gpuInit()` is mandatory called to initialize GPU data structures that are further used by other MoSS functions. The `gpuIdleThread()` is used to detect if a CUDA thread is outside worker threads range, in which case the kernel should finish execution using `return`. In addition to the listed API, we provide functions for string manipulation and conversion between numeric types and strings, as detailed in Appendix C.

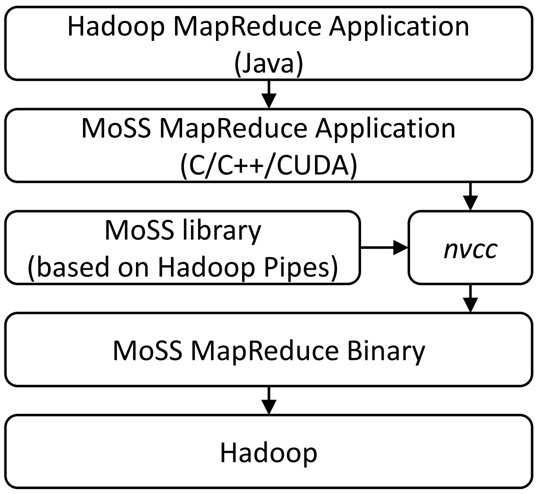


Figure 3.7: Application execution flow in MoSS

Listing 3.5: Example of MoSS API

```
/* initializes Map/Reduce context data structures on GPU */
void gpuInit(TaskContextGPU* ctx);

/* returns the key from the input <key,val> pair */
void gpuGetKey(TaskContextGPU* ctx, char** key);

/* returns the value from the input <key,val> pair */
void gpuGetValue(TaskContextGPU* ctx, char** val);

/* outputs a <key,val> pair */
void gpuEmit(TaskContextGPU* ctx, char* key, char* val);

/* returns true if calling thread is out of worker threads
   range */
int gpuIdleThread(TaskContextGPU* ctx);
```

MoSS supports both systems with discrete GPUs and integrated GPUs by using both explicit data transfers and unified memory feature of recent CUDA releases [50]. On systems with discrete GPUs, MoSS keeps two data structures, one in CPU (host) memory and one in GPU (device) memory. These data structures contain input and output buffers for the <*key, value*> pairs and housekeeping data. The synchronization of these two structures is done explicitly using CUDA APIs for transferring the data. In contrast, on systems integrating the CPU and

GPU there is no need for two different data structures and data transfers. Even on systems with discrete GPUs, recent Nvidia CUDA and GPU drivers perform these transfers transparently using the unified memory feature. However, we show in Section 4.3.2 that these transparent transfers are not always efficient.

We implement the two described overlapping techniques using CUDA features for asynchronous kernel and memory transfers. Checking technique uses `cudaStreamQuery()` function to check for GPU processing completion, while callback technique uses CUDA callback feature through which the GPU notifies the CPU about processing completion. Since checking after each map adds high overhead, we perform this checking after several map function calls. This number is determined based on the ratio of CPU and GPU task times obtained after profiling the execution of first tasks.

Nonetheless, the usability of MoSS can be further improved. Writing the application in C/C++ and CUDA does not have the same flexibility as Java. Approaches enabling Java code to run on systems with GPU exist but have significant limitations. For example, JCuda [99] embeds CUDA code into Java programs, but the developer still needs to write the CUDA kernels. IBM's Liquid Metal project provides solutions for programming heterogeneous systems, such as automatically compiling JVM-compatible Lime [21] code into OpenCL for GPU [34]. But using this approach, MoSS users will still have to firstly rewrite their application using Lime. On the other hand, writing C/C++/CUDA native code has the advantage of performing better than Java for some compute-intensive MapReduce applications [72]. Based on our experience, adapting MapReduce Java applications to C/C++/CUDA requires minor modifications. We believe it is possible to design a tool that can automatically translate Hadoop Java applications into MoSS, but we have not investigated this idea in our thesis.

Since MoSS uses Hadoop Pipes, the input and output of `map()` and `reduce()`

functions are strings. Hence, the developer can use our provided API to convert these strings to appropriate data structures. Secondly, MoSS requires the developer to provide the size for input and output buffers because it is not possible to dynamically allocate GPU memory. However, MoSS provides profiling support to determine the size of these buffers. The developer just has to run the workload on CPU-only and get the maximum size of input and emitted keys and values.

3.5 Summary

In this chapter, we presented the approach of this thesis consisting of three parts, (i) techniques for efficient data-parallel processing on intra-node and intra-chip heterogeneous systems with GPUs, (ii) measurement-driven performance analysis and (iii) model-driven performance analysis. Next, we discussed our techniques for efficient data-parallel processing on intra-node and intra-chip heterogeneous systems with GPU and their implementation under our Hadoop-CUDA framework called *MoSS*. Firstly, our *lazy processing* technique enables the processing of multiple input records on a GPU, in contrast with *chunking* which divides a single record among GPU threads. Secondly, we propose three *dynamic mapping* techniques to further optimize data-parallel processing on heterogeneous systems. Thirdly, our MoSS framework provides an expressive API that allows developers to easily modify existing MapReduce applications in order to exploit heterogeneous systems with GPU while preserving application's logic. In the next chapter, we are comparing the *lazy processing* implemented in MoSS with a related work based on chunking and analyze our proposed *dynamic mapping* techniques. Moreover, we are evaluating the time and energy performance of MoSS in comparison with Hadoop on both self-hosted brawny and wimpy systems, as well as the time performance on available brawny cloud instances with GPU.

Chapter 4

Measurement-driven Performance Analysis

Motivated by the performance improvements of wimpy systems and GPUs, we investigate the time and energy performance of MapReduce on heterogeneous systems with GPUs. We first evaluate MoSS, our Hadoop-CUDA framework based on the *lazy processing* and *dynamic mapping* techniques presented in Chapter 3, and tune framework configuration.

Secondly, we analyze MoSS on clusters of intra-node heterogeneous systems with GPU. For this evaluation, we use six representative workloads that exhibit different system resource demands and cover a range of application domains such as simulations, scientific computing, financial computing, machine learning and data mining, as shown in Table 4.1. The time and energy performance of MoSS is evaluated on an *in-house wimpy* cluster consisting of Nvidia Jetson TK1 nodes with *integrated* GPUs. We are also evaluating MoSS time performance[1] on *cloud-based brawny* clusters consisting of Amazon EC2 instances with *discrete* GPUs.

Thirdly, we discuss a measurement-driven time-energy comparison across six

[1]Energy usage cannot be measured on current cloud setups.

in-house single-node configurations representing brawny and wimpy systems with both discrete (*intra-node*) and integrated (*intra-chip*) GPUs:

- brawny system with Intel Core i7 CPU[2] (**i7**)
- brawny system with Intel Core i7 CPU and discrete Nvidia Maxwell GPU (**i7+GPU**)
- wimpy system with ARM Cortex-A9 CPU (**Kayla**)
- wimpy system with ARM Cortex-A9 CPU and discrete Nvidia Maxwell GPU (**Kayla+GPU**)
- wimpy system with ARM Cortex-A15 CPU (**Jetson**)
- wimpy system with ARM Cortex-A15 CPU and integrated Nvidia Kepler GPU (**Jetson+GPU**)

For this comparison, we are using three of the six MapReduce workloads to cover compute-intensive, data-intensive and mixed system resources demands.

4.1 Applications

The workloads used in this evaluation are chosen to represent a range of data-parallel applications typically used in related work. These applications cover domains such as simulations, scientific computing, financial computing, machine learning and data mining. Pi estimation (**PI**) is a MapReduce application included in Hadoop examples. The estimation algorithm uses Monte Carlo simulation to approximate the value of constant π. We adapted Hadoop's version of PI by replacing the random number generation method from using Halton sequence to

[2]Although typical brawny server systems employ Intel Xeon processors, we use Intel i7 in our time-energy analysis since it exhibits better performance, as shown in Appendix A.

Table 4.1: Data-parallel applications

Workload	Data Size [GB]	Description
Pi Estimation (PI)	0.003 (S)	16 billion samples
	0.018 (B)	100 billion samples
	0.056 (M)	300 billion samples
	0.186 (L)	1000 billion samples
BlackScholes (BS)	0.8 (S)	12 million options
	4.0 (B)	60 million options
	8.0 (M)	120 million options
	24.2 (L)	360 million options
Kmeans (KM)	0.3 (S)	n=3,474,500; m=34; k=5
	3.9 (B)	n=41,694,000; m=34; k=5
	7.7 (M)	n=83,397,420; m=34; k=5
	19.3 (L)	n=208,493,550; m=34; k=5
Matrix Multiplication (MM)	0.9 (S)	n=500
	5.4 (B)	n=900
	7.5 (M)	n=1,000
	26.0 (L)	n=1,500
Similarity Score (SS)	0.9 (S)	n=500, m=250,000
	5.4 (B)	n=900; m=810,000
	7.5 (M)	n=1,000; m=1,000,000
	26.0 (L)	n=1,500; m=2,250,000
Grep (GR)	0.6 (S)	7,800,963 lines
	6.3 (B)	83,328,469 lines
	11.1 (M)	166,656,938 lines
	22.3 (L)	368,789,935 lines

using *xorshift* [73] method which is more efficiently executed on GPU. Map phase generates random 2-dimensional points (samples) and counts how many are inside and outside the unit circle. Reduce phase computes the value of π based on the ratio of inside and outside point counts generated by Map phase.

BlackScholes (**BS**) implements a financial model that derives the price of European-type options. It takes options characteristics, such as stock price, interest rate, expiration time, as input and computes the price of each option. In our work, we adapt the open-source PARSEC 3.0 [22] BlackScholes version. In the Map phase, BlackScholes equations are applied to each option to get its price and Reduce phase forwards these prices to the output. BS datasets are generated with PARSEC input generator.

Kmeans (**KM**) is a popular cluster analysis method used in data mining to group a set of n points, each having m features, into k clusters. This grouping is based on the Euclidian distance between each point and the centroids of each cluster. We adapt Kmeans implementation from Mars [52] and use its input generator. In this implementation, the Map phase determines the closest cluster for each point and emits pairs consisting of the point and its associated cluster id. Reduce phase computes cluster centroids based on the points associated with each cluster id, and outputs the updated centroids of the k clusters.

Matrix multiplication (**MM**) is a widely-used application that is part of many real world applications such as recommendation systems used by the majority of web platforms. In our implementation, we multiply two randomly generated square matrices A and B of size n. Each input line contains two vectors, namely, row i from A and column j from B. Map phase multiplies these two vectors to get the i, j element of the result matrix. Reduce phase outputs this result matrix.

Similarity score (**SS**), which is widely used in data mining and recommendation systems, uses cosine similarity to establish the correlation between two objects represented by n-dimensional vectors. Our implementation compares m pairs of vectors with n elements. Map function takes a pair of vectors and computes their similarity as the cosine of the angle between the two vectors. Reduce function outputs similarity values computed by Map phase.

Grep (**GR**) is a well-known MapReduce workload [33] which determines input file lines matching a regular expression. We adapt Hadoop's Java implementation of Grep by modifying Reduce phase to compute the number of lines matching the regular expression, which is equivalent to running `grep <regex> <input> | wc -l` Unix command. In our evaluation, Grep searches for the string "*the*" in the latest dump of Wikipedia articles.

In addition to covering multiple application domains, these workloads exhibit diverse system resource demands. PI and BS represent compute-intensive applications, their kernels being used in HPC [22]. KM exhibits mixed resource demands by having both a relatively complex kernel and large data transfer requirements. Intuitively, MM and SS should be compute-intensive. However, their MapReduce implementations are more data-intensive since data transfers dominate the execution time. Moreover, their kernels consisting of a loop doing multiply-add operations is less compute-intensive compared to PI, BS and KM. Similarly, GR is more data-intensive since it handles large input text files and its kernel does only string matching. We present empirical evidence supporting this classification in Section 4.4.

4.2 Systems

Our experimental setup consists of five in-house system configurations covering homogeneous and heterogeneous computing landscape, as shown in Table 4.2. For cloud computing, since there are no wimpy systems available, we select homogeneous and heterogeneous brawny systems with GPU.

For single node analysis, we are using three heterogeneous systems each with two different configurations. Firstly, we use a traditional brawny *intra-node* heterogeneous system for comparison with the wimpy systems. This **i7** brawny system is based on a 4-core Intel Core i7 processor and 16 GB of RAM. A 512 GB solid-state drive (SSD) is used to store all datasets and workloads, while Ubuntu 13.04 with Linux kernel 3.11.0 is installed on another SSD.

Table 4.2: Systems selection

Homogeneous		**Heterogeneous with discrete GPU**		**Heterogeneous with integarted GPU**	
Brawny	**Wimpy**	**Brawny**	**Wimpy**	**Brawny**	**Wimpy**
i7	Kayla	i7+GPU	Kayla+GPU	N/A	Jetson

Secondly, an *intra-node* heterogeneous wimpy system is represented by a **Kayla** node equipped with Nvidia Tegra 3 System-on-a-Chip (SoC) having four ARM Cortex-A9 cores and 2 GB of low-power DDR2 [30]. This system has a PCI Express x16 port that can accommodate a full-fledged *discrete* GPU. Moreover, it has a SATA interface which enables the connection of a high-capacity disk. We use the same 512 GB SSD to store the datasets and workloads. By default, Ubuntu 12.04 with Linux kernel 3.1.10 is installed on system's flash storage. On top of this OS, we install CUDA toolkit 6.5 and necessary Nvidia drivers.

Thirdly, with the increasing adoption of *integrated* CPU-GPU systems [75], we use an *intra-chip* heterogeneous wimpy system represented by Nvidia **Jetson** TK1 based on Tegra K1 SoC which integrates four ARM Cortex-A15 CPU cores, 192 Nvidia Kepler GPU cores and a shared 2 GB low-power memory. Beside the four fully-fledged Cortex-A15 cores, Tegra K1 incorporates a transparent low-power *companion* core which runs the OS at low system utilization. We connect the same 512 GB SSD to this platform, while the Ubuntu 14.04 OS with Linux kernel 3.10.40 is installed on a 16 GB eMMC. By default, Jetson TK1 comes with CUDA toolkit 6.0 and associated Nvidia drivers. However, we encountered kernel panics and errors when running Hadoop-CUDA on this default setup. Upgrading the drivers and CUDA toolkit to version 6.5 solved these issues.

We use Nvidia Maxwell architecture by hosting on both the wimpy A9 and brawny i7 systems a GTX 750 Ti video card consisting of 640 cores with CUDA compute capability 5.0, and 2 GB of GDDR5 memory. Compared to previous Kepler architecture, Maxwell is two times more energy-efficient [51], being the suitable choice for connecting to a low-power system.

For the cluster-level analysis of MoSS, we use wimpy and brawny systems. The wimpy systems are represented by the previously described Jetson TK1 kit, while the brawny systems are represented by **g2.2xlarge Amazon EC2** instances with

GPUs. These instances are equipped with Intel Xeon E5-2670 CPU and Nvidia GRID K520 GPU. This GPU of Kepler architecture has 1536 cores and 4 GB of memory. Each instance is configured with eight virtual CPU cores, 15 GB of RAM and 500 GB of SSD-based storage space. From the software perspective, the instances run Ubuntu OS with Linux kernel 3.13.0. On this OS, MoSS code is compiled with *gcc* 4.8.2 and *nvcc* from CUDA toolkit 6.5, while Hadoop framework runs on *jdk1.8.0_25*. More details about the systems and their characterization are presented in Appendix A.

Hadoop is configured to use four map slots and one reduce slot on i7 and Jetson. On Amazon EC2 instances, we configure Hadoop with eight slots for Map tasks such that on n cluster nodes there are $8n$ Map tasks running in parallel. For Reduce tasks, we configure a single slot per cluster node. On the wimpy Kayla system, workloads with large inputs encounter failures due to insufficient memory. To avoid such failures and to reduce Hadoop's memory usage, we set `io.sort.mb` to 40 and `io.sort.spill.percent` to 0.50, compared to default values of 100 and 0.80, respectively. The native binaries for CPU-only and CPU+GPU are compiled using the *gcc* available on Ubuntu OS and the *nvcc* from CUDA toolkit, and executed through Hadoop Pipes mechanism. The C/C++ code is compiled with *gcc* using maximum level of optimizations (`-O3`). On the wimpy systems, we optimize the code for Cortex-A9 using `-mcpu=cortex-a9 -mtune=cortex-a9`, and Cortex-A15 using `-mcpu=cortex-a15 -mtune=cortex-a15`. Moreover, we enable fast NEON floating-point instructions (`-mfpu=neon`) available on ARM processors.

Performance metrics, such as instructions per cycle (IPC), utilization and memory operations, are collected using *perf* from Linux tools for CPU and *nvprof* from CUDA toolkit for GPU. For each execution, *perf stat* is attached to Hadoop TaskTracker daemon to collect performance counters for spawned Map and Reduce tasks. In contrast, *nvprof* is executed through Hadoop Pipes as a wrapper over

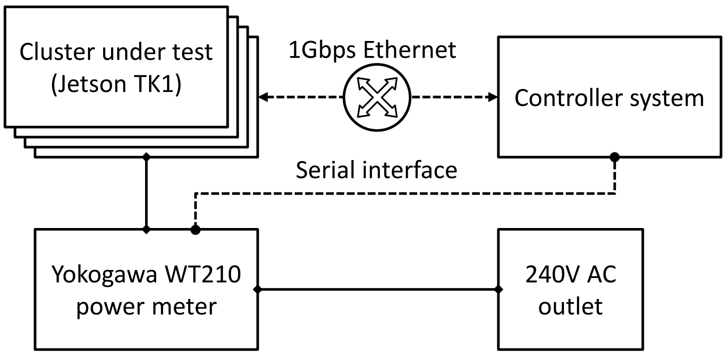


Figure 4.1: Experimental setup

CPU+GPU binaries. Energy and power are measured using a Yokogawa WT210 power monitor connected to the AC line of each system, as depicted in Figure 4.1. Hence, power and energy values include the inefficiencies of the power adapter, but since these values are reflected in the energy bill at the end of the month, we believe they are more meaningful than DC values. The logs are collected by a controller system sharing the same LAN as the systems under test.

Each workload is executed three times and average time and energy values are reported, while error bars on the plots indicate the standard deviation. We check workloads outputs for correctness across all input sizes and platform configurations. Interestingly, we observed that BS outputs are slightly different on the wimpy-only systems. This happens because ARM NEON floating point unit does not fully adhere to the IEEE 754 standard [43]. Thus, when the code is compiled with `-mfpu=vfpv3`, the results are identical with the references, but the execution time increases. Since we want the best performance and the precision loss is acceptable, we report the results obtained by faster NEON instructions.

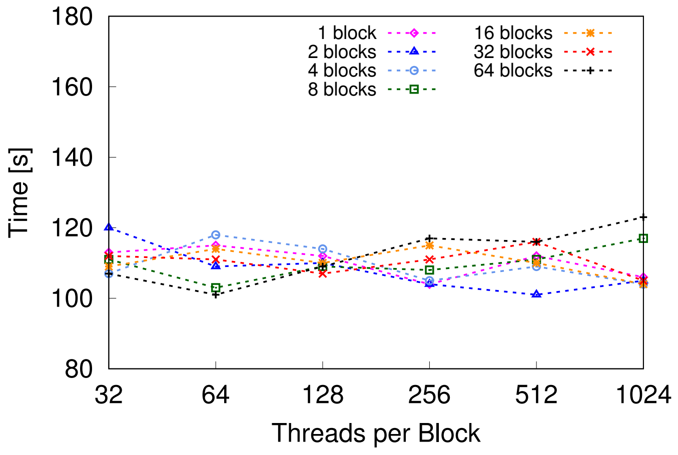
4.3 MoSS Setup and Evaluation

4.3.1 Determining GPU Thread Count

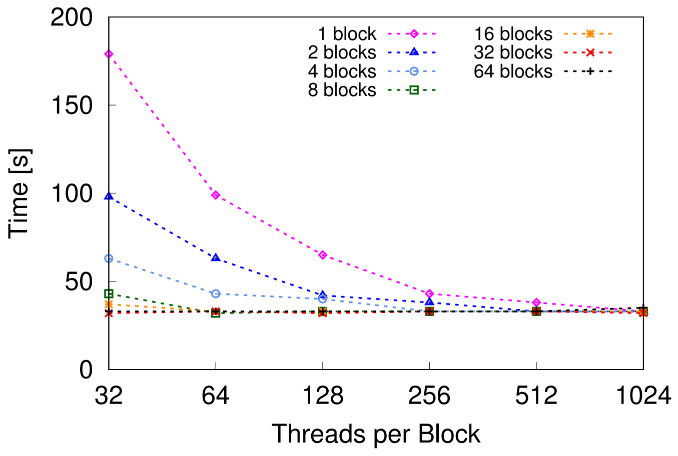
GPU thread count represents the parameter h used by the *lazy processing* technique to batch input records for GPU processing, as discussed in Section 3.2. In CUDA, this thread count consisting of number of thread blocks and threads per block plays an important role in efficient execution on GPUs. To set the value of h, we investigate the effect of different CUDA thread counts on GPU-only execution. The thread count directly affects the memory usage because each thread has input and output data buffers. Hence, we are constrained to select the smallest thread count that exhibits a good performance. This constraint is particularly stringent in wimpy systems as they have small memories of typically 1 or 2 GB. We determine the CUDA thread count by varying the number of blocks starting from one and the number of threads per block starting from 32 since CUDA threads are grouped in warps of 32 threads.

On Jetson, the effect of varying CUDA thread count is less visible, as shown in Figure 4.2a for KM workload. The execution time is within 10% of the average 110 seconds and it slightly degrades for high thread counts. This is explainable since Jetson's GPU has only 192 cores which cannot accommodate large numbers of CUDA threads. Hence, we select a single block with 256 CUDA threads as a good configuration to run MoSS on Jetson. The only exception is PI which has a highly compute-intensive kernel that achieves significantly better performance when executed on two thread blocks of 512 threads per block.

On the other hand, the effect of varying the number of CUDA threads is more visible on Amazon instances, as depicted in Figure 4.2b for KM workload. The Kepler GPU used by Amazon instances has 1536 cores and it is underutilized on small



(a) On Jetson TK1



(b) On Amazon EC2

Figure 4.2: Effect of CUDA threads for KM.S

thread counts, leading to high execution time. But with the increasing number of threads per block, the execution time across different block counts converges. The other workloads exhibit similar behavior as KM. Using this convergence effect, we can select the same thread count value across different workloads to achieve efficient MoSS execution. Thus, we select 8 blocks with 1024 threads per block since it exhibits good performance for all workloads at relatively low memory footprint on Amazon g2.2xlarge instances.

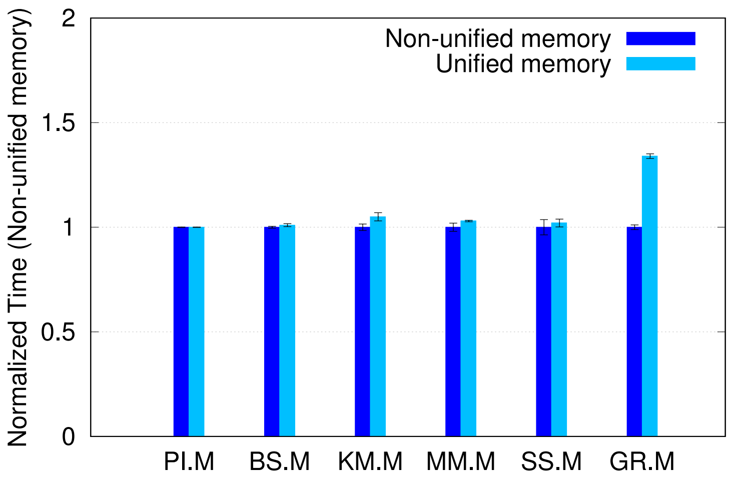


Figure 4.3: Effect of unified memory

4.3.2 Effect of Unified Memory

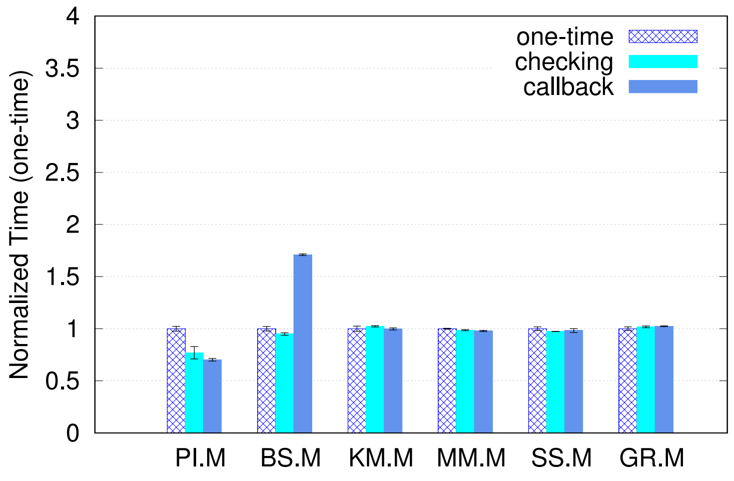
To improve the programmability of Nvidia GPUs, recent versions of CUDA support unified memory [50]. For platforms with integrated GPUs, such as Jetson, unified memory is the natural choice because the physical memory is shared between the CPU and GPU. In contrast, for systems with discrete GPUs, such as g2.2xlarge instances, it is unclear if unified memory performs better than explicit transfers. Since MoSS supports both unified memory and explicit transfers, we compare them on g2.2xlarge cloud instances across all workloads and present the results in Figure 4.3.

For workloads requiring less amount of data transfers, such as PI and BS, unified memory achieves the same performance as explicit transfers. In contrast, for data-intensive workloads such GR, unified memory exhibits a slowdown of 34%. Although unified memory is supposed to maximize data access speed, we observe that this is not true for large datasets. While we do not have access to CUDA internals to further investigate the cause, we reckon this slowdown is due to memory management and transfer serialization in CUDA driver. In conclusion, we set explicit data transfers in MoSS for systems with discrete GPUs.

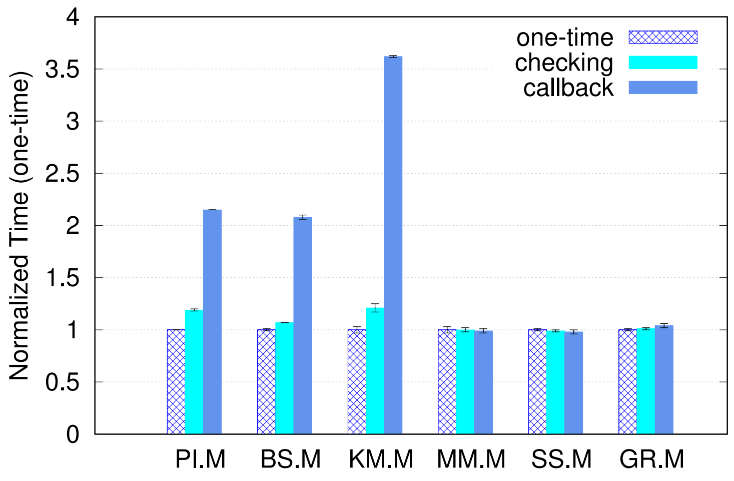
4.3.3 Comparison of Dynamic Mapping Techniques

Intuitively, overlapping task execution on both the CPU and GPU has the potential to improve the overall performance. However, we show that overlapping the CPU and GPU has no benefit for MapReduce workloads. By profiling workload execution, we find that GR exhibits at least 10 times faster record processing on the CPU, while PI, BS and KM input records are 2 to 7 times faster processed by the GPU and the ratio of computation to communication for the GPU is at least 2. The only workload which exhibits faster record processing on the GPU but high data transfer times is SS. But in practice, the overall execution of SS is not significantly improved by overlapping, as shown in Figure 4.4.

By experimentally comparing the three dynamic mapping techniques described in Section 3.3, namely, *one-time*, *checking* and *callback*, we observe that the *one-time* approach achieves better results. Our observation is in correlation with other work [57], which indicates that overlapping the CPU and GPU has the potential to improve MapReduce execution with less than 10%. Figure 4.4 shows the results of dynamic mapping techniques comparison by normalizing the execution times with those of the one-time approach. Surprisingly, the callback technique is much slower than the one-time approach for compute-intensive workloads such as BS and KM. By analyzing the execution logs, we found that CUDA inexplicably delays kernel execution when using the callback mechanism. Because of this delay, the majority of records are processed by the CPU, hence, the lower performance. While for compute-intensive workloads this delay results in much lower performance, for data-intensive workloads there is no degradation because the entire processing is done on CPU. To further test our hypothesis that CUDA callback is inefficient, we have implemented a simple matrix multiplication kernel which is first executed on the GPU and then on the CPU. We added a callback to signal the end of GPU



(a) On Jetson TK1



(b) On Amazon EC2

Figure 4.4: Dynamic mapping techniques comparison

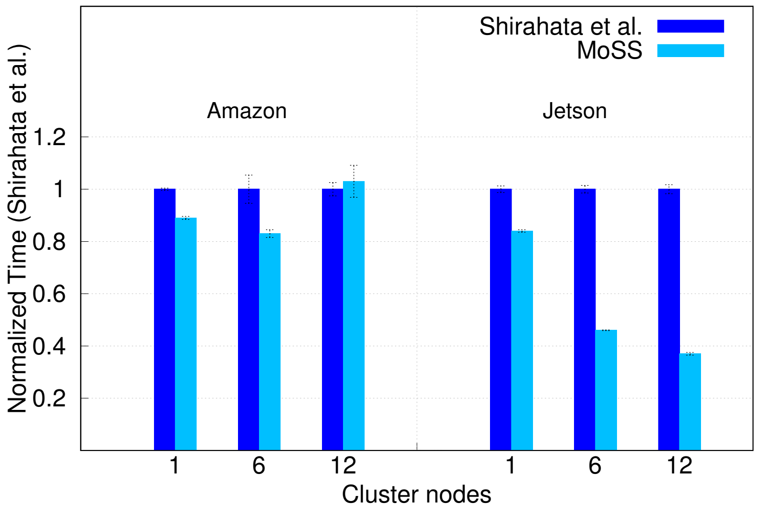
processing and found that it is triggered during or after CPU processing ends. Ideally, the callback should be triggered exactly at the end of GPU execution.

The checking technique is around 10% slower than the one-time for compute-intensive workloads. This slowdown is attributed to the overhead of checking the GPU to determine when it finishes the processing. Decreasing the checking frequency does not achieve better results because more records are processed by the CPU which exhibits slower processing compared to the GPU for these compute-intensive workloads. For workloads that exhibit better performance on CPU, such as GR, the overlapping has little effect since one-time decides to execute almost always on the CPU. In conclusion, MoSS is by default configured to use the one-time approach, and we present the results using this technique in the next subsections.

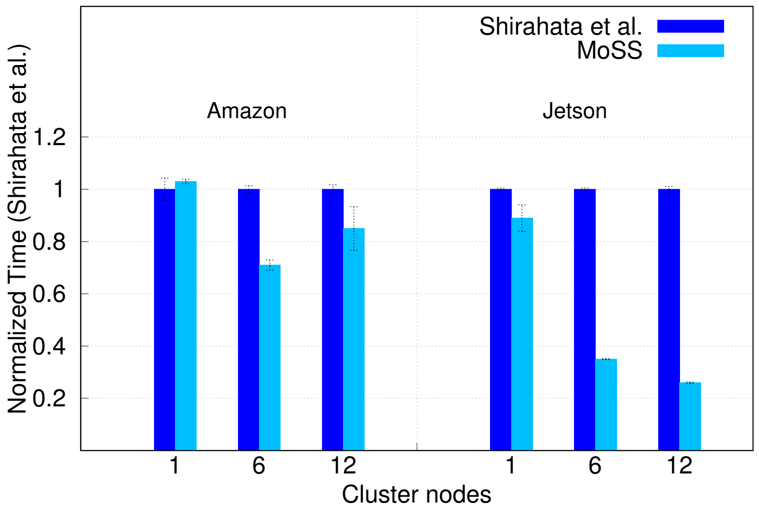
4.3.4 Comparison with Chunking

In this section, the performance of MoSS is compared with the work of Shirahata et al. [84]. We compare MoSS only with Hadoop-based projects to account for framework overheads. In contrast, other related works exhibit higher execution time performance but trade-off scalability and fault tolerance. Among Hadoop-based frameworks presented in Chapter 2, we could compare with Shirahata et al. [84] for which the source code is available. Since this work is based on Hadoop 0.20.1, we have ported MoSS to this version of Hadoop in order to perform a fair comparison.

We have implemented the same Kmeans clustering algorithm [84]. In this Kmeans version, all the work is performed in the Map phase, as opposed to the algorithm presented in Section 4.1. Each input record consists of n 2-dimensional points that are to be grouped into k clusters. Each map function performs this grouping and outputs k centroids. The Reduce phase only outputs the centroids. For the comparison, we generate two datasets of $n = 32$ points which are grouped into $k = 5$ clusters, with the input generator used in [84]. First dataset contains 10 million records occupying 6.9 GB of disk space, while the second dataset has 30 million records with a total size of 20.6 GB. We perform the comparison on clusters of one, six and twelve nodes of both Jetson TK1 boards and Amazon g2.2xlarge instances and present the results in Figure 4.5. We use the same hardware and software setup presented in Section A, except for Hadoop version 0.20.1 instead of version 1.2.1.



(a) Using M datasets



(b) Using L datasets

Figure 4.5: Comparison with chunking

On average, MoSS performs 54% better than Shirahata et al. [84]. On clusters of 12 Jetson TK1 nodes, MoSS achieves a maximum speedup of 3.8 and energy savings of almost two-thirds. On Amazon instances, MoSS achieves a maximum speedup of 1.4 on six nodes. As expected, [84] performs worse on the wimpy systems compared to the brawny systems for which it was designed. On the wimpy systems, we observe a delay in execution. By investigating the log files, we found that the acceleration factor used in splitting the tasks between CPU and

GPU [84] is frequently computed as zero in the initial phase. This zero factor results in assigning all the tasks to the GPU, thus, overwhelming the low-power GPU of Jetson TK1. In contrast, MoSS ensures a balanced execution by profiling the execution on CPU and GPU at the beginning of each task.

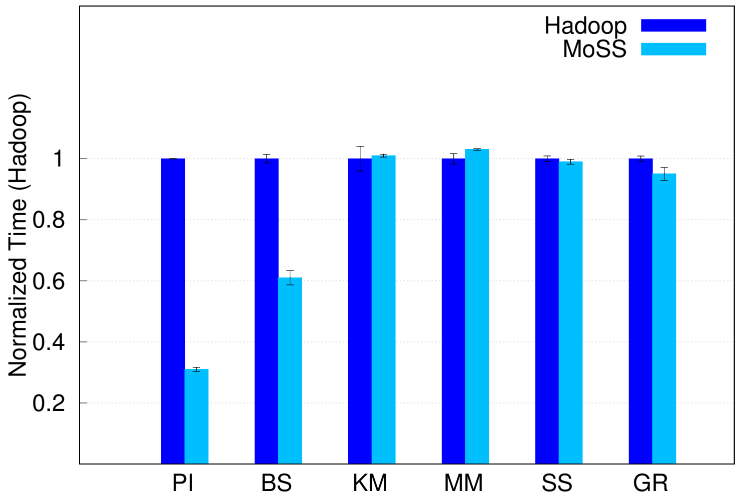
4.4 Homogeneous versus Heterogeneous Systems

In this section, we evaluate the time-energy performance of MoSS in comparison with Hadoop on clusters of both wimpy Jetson TK1 and brawny Amazon EC2 g2.2xlarge nodes.

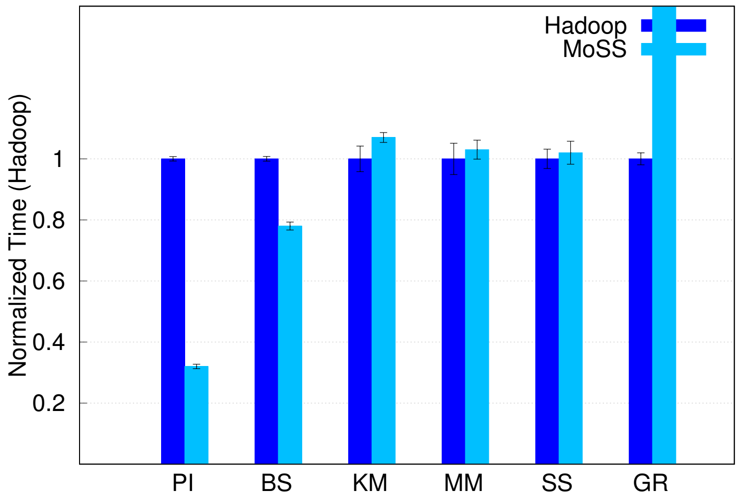
4.4.1 Analysis on Nvidia Jetson TK1

We present the execution time of MoSS normalized to Hadoop across Jetson clusters using both M and L datasets in Figure 4.6. For compute-intensive workloads, such as PI, BS and KM, MoSS improves the execution time on 12 nodes by factors of 3.1, 1.1 and 1.1, respectively. For workloads with larger input records, such as SS and GR, MoSS achieves the same execution time as Hadoop by deciding to process the workloads only on CPU. Log files for single node execution show that MoSS does not use the GPU for SS and GR, while for PI and BS the GPU processes 99% of the input records. Surprisingly, the GPU processes only 3% of the input records for KM.

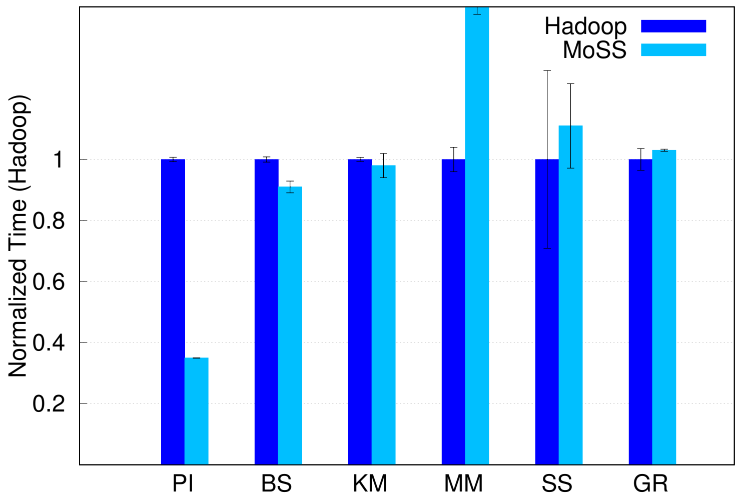
We further analyze the unexpected behavior of KM and SS. The profiling results of KM execution on both Hadoop and MoSS show that KM utilizes more than 95% of the memory, reads and writes around 8 GB of storage data, and sends and receives 2 GB of data on the network, on average per cluster node. Compared to the other workloads, KM is the most resource-intensive and pushes wimpy nodes to their limits. Map phase alone exhibits a speedup of up to 40%



(a) 1 node, M dataset

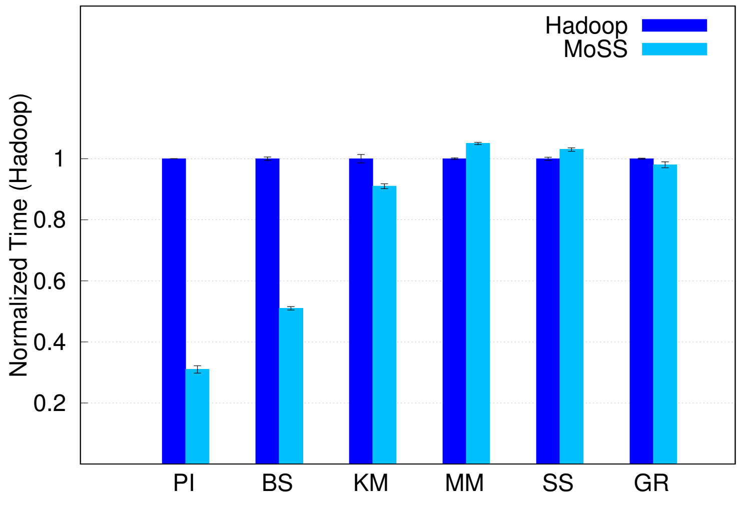


(b) 6 nodes, M dataset

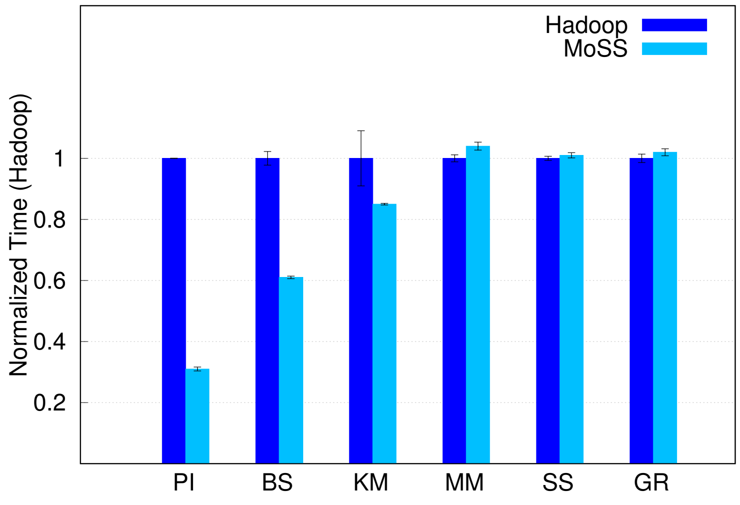


(c) 12 nodes, M dataset

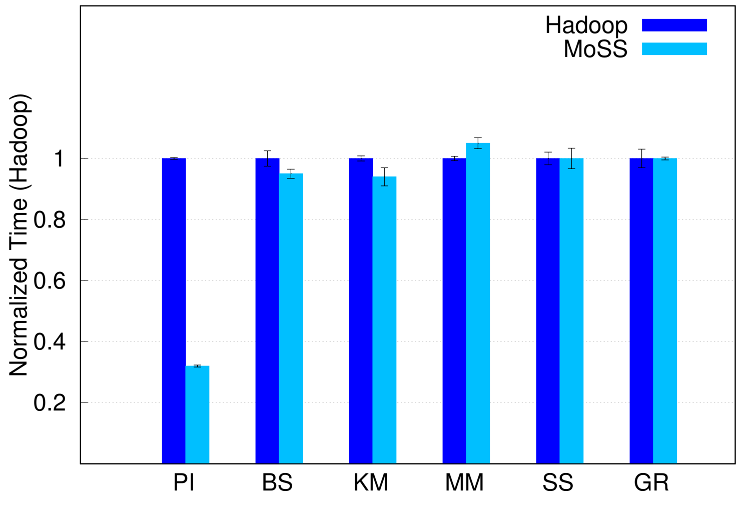
Figure 4.6: Normalized execution time on Jetson TK1 clusters



(d) 1 node, L dataset



(e) 6 nodes, L dataset



(f) 12 nodes, L dataset

Figure 4.6: Normalized execution time on Jetson TK1 clusters

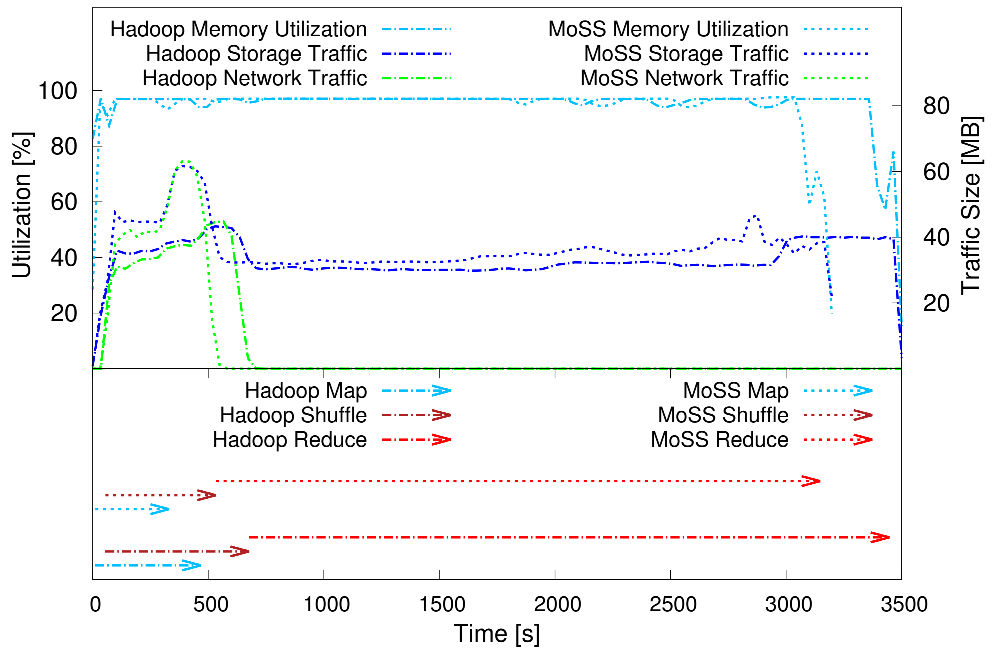
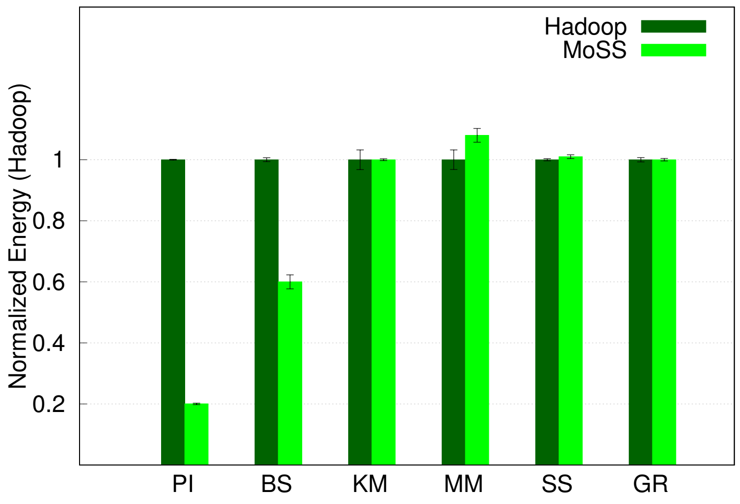


Figure 4.7: KM.L execution on a Jetson TK1 node

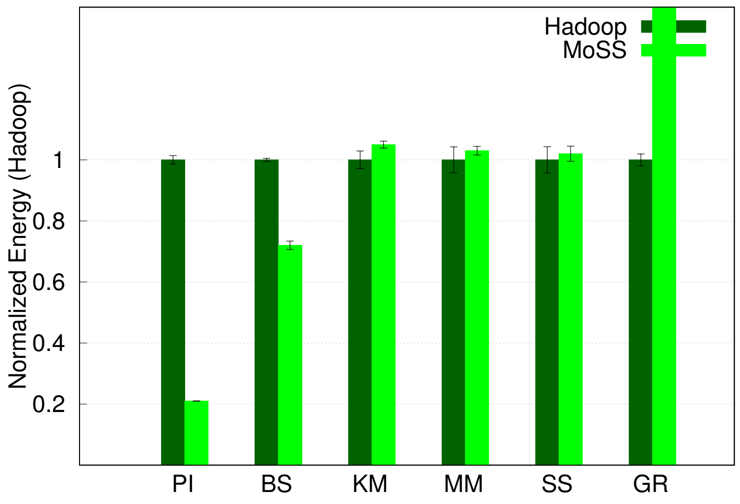
on six nodes when employing the GPU. But KM has heavy Shuffle and Reduce phases, as shown in Figure 4.7, and thus, the overall speedup is less than 10%. For SS, the profiling of GPU-only execution shows that it achieves a high *warp execution efficiency* of 67% which is close to the 74% achieved by PI. However, global memory usage is two times less efficient compared to that of PI, as indicated by the *gld efficiency* metric. These values show that similarity score computation can be accelerated by the GPU, but memory operations such as loading input records nullify this acceleration. Lastly, we present the reasons for low runtime performance of GR on GPU. The processing time of GR records across GPU threads is uneven because CUDA threads take longer time to finish the processing of records that do not contain the regular expression. In contrast, for records containing the expression at the beginning, the threads finish the processing very fast but have to wait for the slowest thread in the warp. Our explanation is reinforced by *warp execution efficiency* value of 50% for GR, the lowest among the six workloads.

Before showing the energy usage of MoSS on Jetson clusters, we present the power profile of a single Jetson TK1 system. This power profile is determined by executing micro-benchmarks that exercise each sub-system, such as the CPU, GPU and hard-disk, to its maximum capacity. The idle power consumed by Jetson TK1 running only the Ubuntu OS but excluding the hard-disk is 3.2 W. This power includes the 0.5 W consumed by the fan that is installed on the board by default. When executing compute-intensive applications on both the CPU and GPU, Jetson consumes up to 13.5 W. Apart from the idle power, the CPU contributes to this total power with more than 6.5 W, while the GPU uses around 3 W. Adding the 3 TB hard-disk almost doubles the idle power and increases the power consumption of the fully-utilized system to around 17 W. All these values represent AC power and include the inefficiencies of the power adapter. Nevertheless, this power is reflected in the energy bill at the end of the month and, thus, we believe it is more meaningful than DC power.

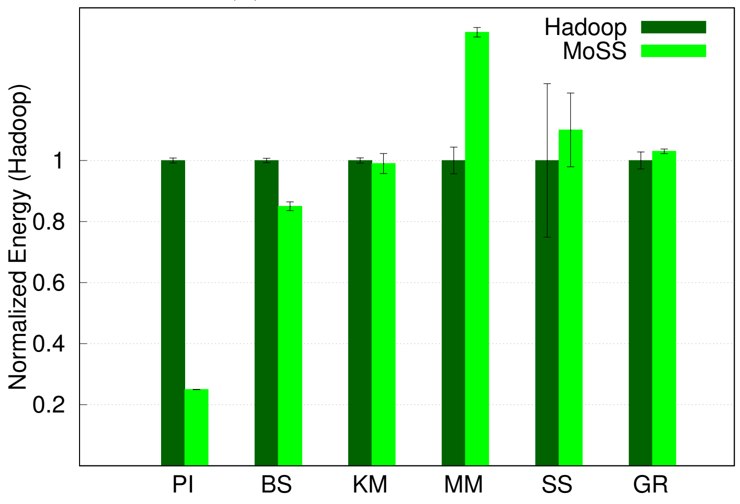
The energy usage of MoSS workloads, plotted in Figure 4.8, follows the execution time trend. However, across cluster sizes, the energy usage increases because more nodes are added and the speedup of Hadoop is sub-linear. For example, the speedup of BS on 12 nodes compared to a single node is 2.5. The energy usage of BS on a single node is almost 100kJ. Considering that the average power remains constant, the energy on 12 nodes should be the sum of energies for each node divided by the speedup, which results in 480kJ. But the average power of each node on the 12-node cluster is smaller than single-node average power because each node has lesser work to perform. Hence, the actual energy of BS on the 12-node cluster is 320kJ. Figure 4.8 shows that MoSS incurs lesser energy for compute-intensive workloads and that it saves around 80%, 16% and 5% of the energy consumed by Hadoop for PI, BS and KM, respectively. For the other two workloads, the energy consumption of MoSS is at most 1% higher.



(a) 1 node, M dataset

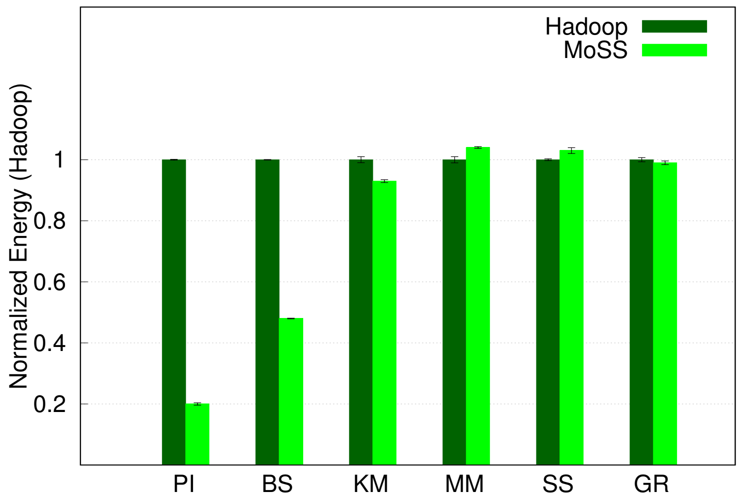


(b) 6 nodes, M dataset

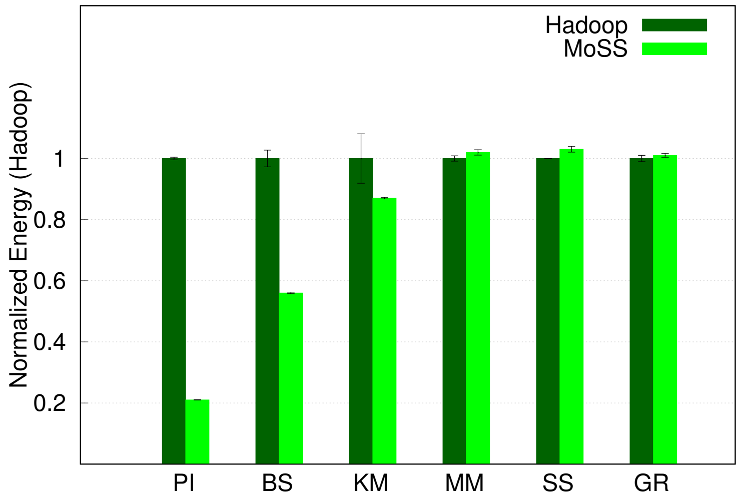


(c) 12 nodes, M dataset

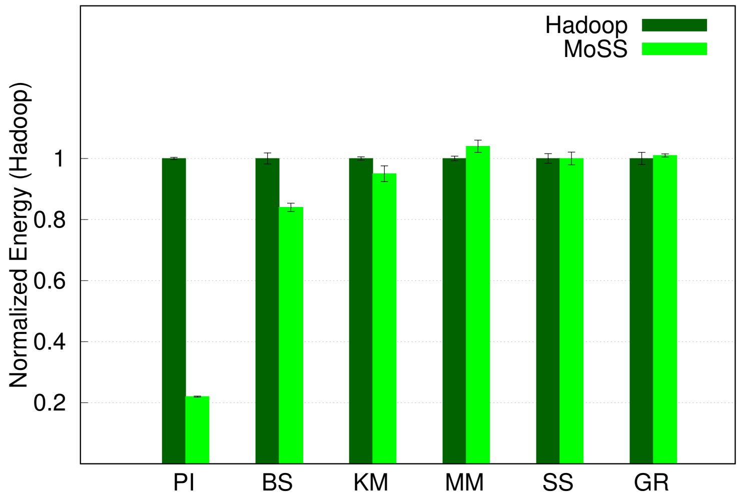
Figure 4.8: Normalized energy usage of Jetson TK1 clusters



(d) 1 node, L dataset

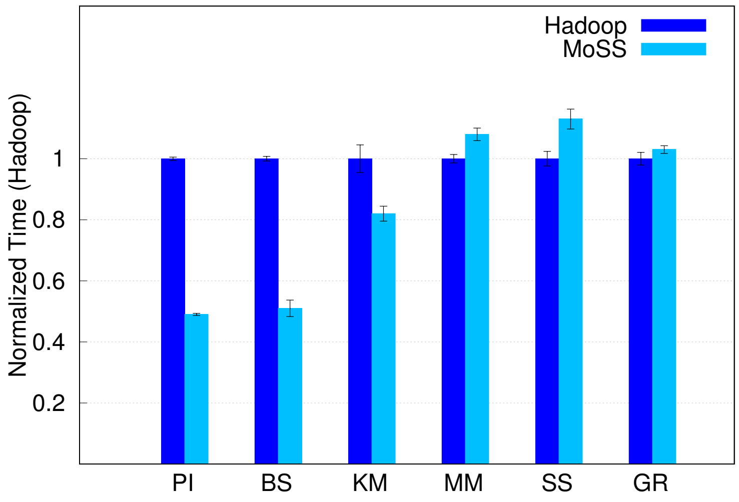


(e) 6 nodes, L dataset

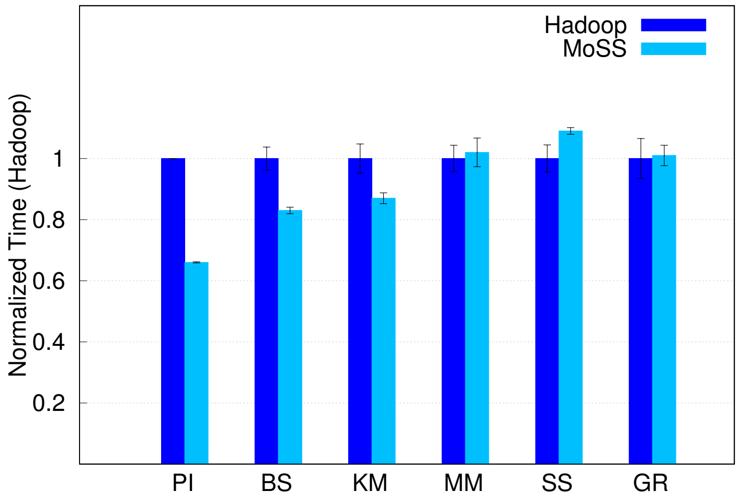


(f) 12 nodes, L dataset

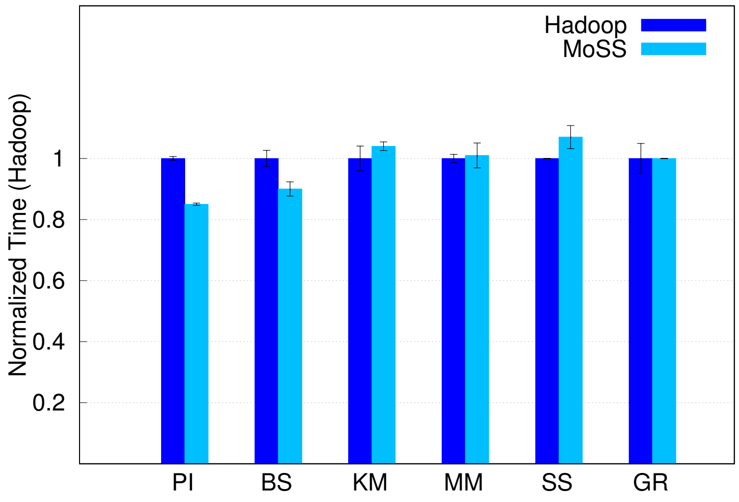
Figure 4.8: Normalized energy usage of Jetson TK1 clusters



(a) 1 node, M dataset

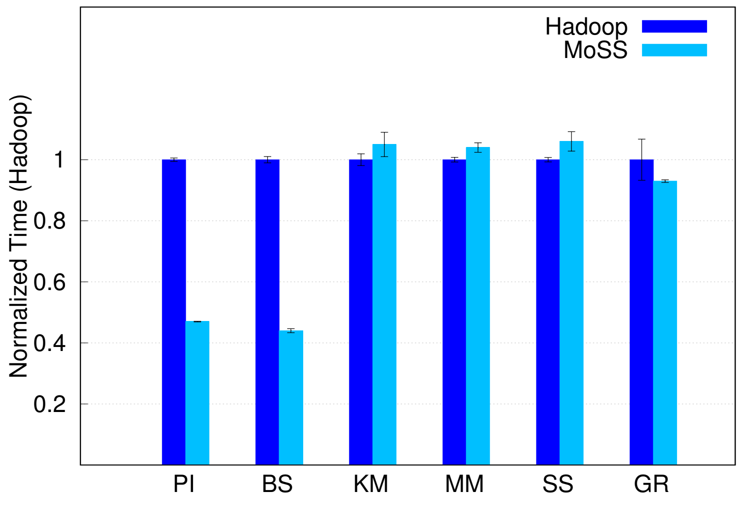


(b) 6 nodes, M dataset

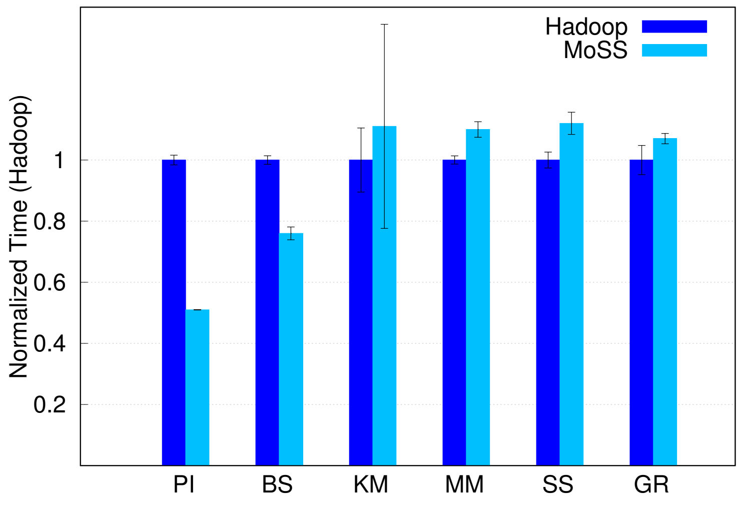


(c) 12 nodes, M dataset

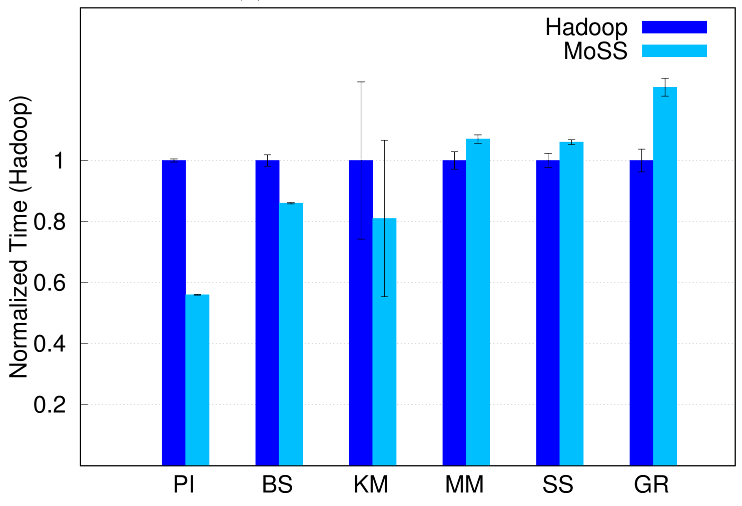
Figure 4.9: Normalized execution time on Amazon EC2 clusters



(d) 1 node, L dataset



(e) 6 nodes, L dataset



(f) 12 nodes, L dataset

Figure 4.9: Normalized execution time on Amazon EC2 clusters

4.4.2 Analysis on Amazon EC2

With cloud computing becoming ubiquitous, we evaluate MoSS on cloud-based Amazon EC2 instances with GPUs. Since power consumption of cloud instances cannot be measured, we present only the execution time performance. Execution time results across all six MoSS workloads with M and L datasets executing on one, six and twelve nodes are plotted in Figure 4.9. Similar to Jetson TK1, the execution time improvement delivered by MoSS depends on workload type. Compute-intensive workloads such as PI and BS are suitable for GPU processing. While PI exhibits speedups of around 2, BS exhibits maximum speedups of 2.3 and 1.2 on one and twelve nodes, respectively. For KM, heavy Shuffle and Reduce phases affect its performance. Moreover, on six and twelve nodes the execution times of KM vary significantly. We attribute this to the unpredictable I/O operations in the cloud and to the large data transfers incurred by KM. The profiling of storage and network activity for KM execution on Amazon instances shows 254 GB and 39 GB of traffic, respectively, at cluster level. Being the highest among all six workloads, these values are large considering the 19 GB input of KM.L and illustrate the impact of I/O operations on MapReduce execution. As a side note, the Map phase of KM is 25% faster on MoSS compared to Hadoop because MoSS uses the GPU to speedup the processing of around 98% of KM's input records. But the overall performance is influenced by the Shuffle and Reduce slowdown and the high variance across runs due to I/O operations in the cloud. Similar to Jetson TK1, for SS and GR the GPU-only execution does not improve the execution and, thus, MoSS uses only the CPU. Nevertheless, SS computational part is suitable for GPU processing, as indicated by the 99% *sm_efficiency* which is higher even than the values for PI and KM, 81% and 85%, respectively. But data transfer time between the main memory and the GPU's global memory dominates total

GPU processing time and leads to better overall performance of SS on the CPU. However, we anticipate improved MoSS execution of data-intensive applications on future GPU architectures such as Nvidia Pascal. These GPUs will have at least five times higher transfer rates enabled by NVLink [41].

4.5 Brawny versus Wimpy Systems

In this section, we present a time-energy performance analysis of MapReduce on intra-node and intra-chip heterogeneous systems with GPU, including bottleneck analysis and time-energy equivalence between brawny and wimpy systems.

4.5.1 Time-Energy Performance Analysis at Single-Node Level

Firstly, we present the time-energy performance of MapReduce workloads on the six platform configurations in Figure 4.10 using log scale. The standard deviation among multiple runs is very small, as shown by the error bars. For compute-intensive BS, the GPU significantly improves the execution time only on the brawny system. The speedup of 2.3 leads to 45% energy savings, although the average power of the i7+GPU is slightly higher compared to the i7 CPU-only. On the wimpy Kayla system, the speedup is less than 1.1, while on Jetson the GPU degrades the execution time by almost 10%. This is a surprising result since BS is a compute-intensive workload, suitable for GPU processing. By further analyzing the behavior of BS on Jetson, we discover that the compiler optimizations lead to a speedup of 3.3 compared to the non-optimized binary. As BS consists of a loop in which the option price is computed, it is suitable for loop optimizations, such as loop unrolling, that can significantly improve the execution time. However, for the other two workloads, these compiler optimizations lead to no execution time

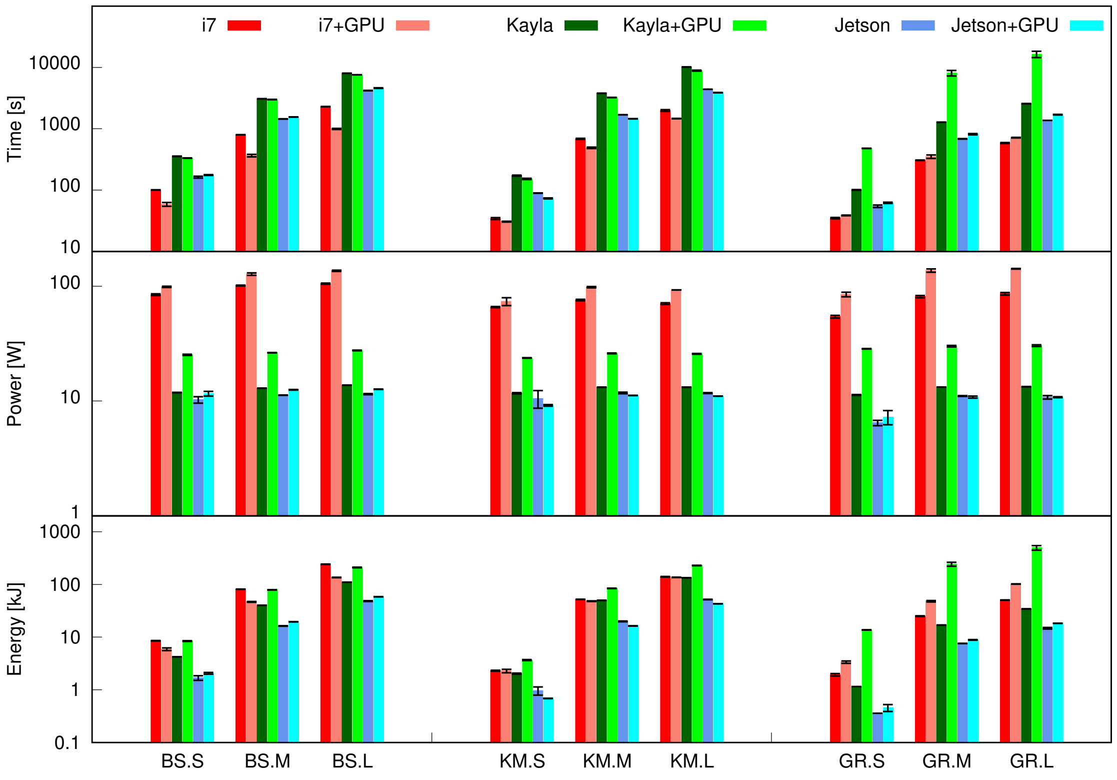
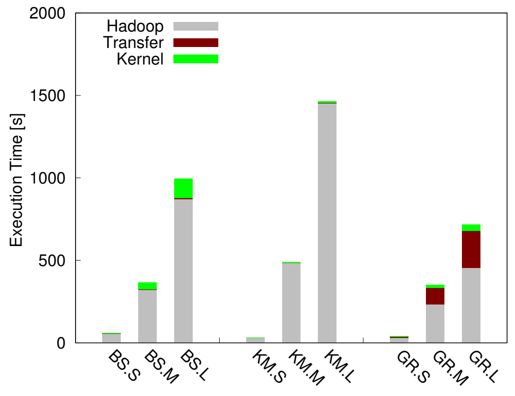
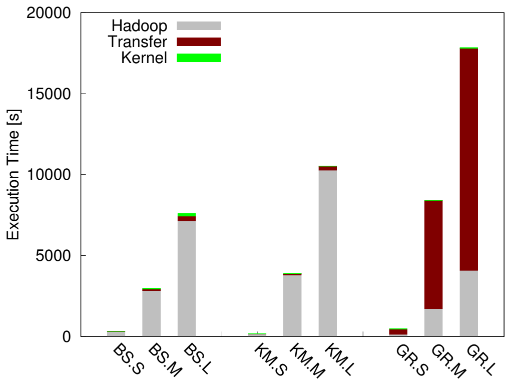


Figure 4.10: MapReduce time-energy performance

improvements. For KM, both wimpy systems with GPU exhibit speedups close to 1.2. While the speedup on Jetson+GPU leads to energy savings of around 20%, on Kayla+GPU the energy is 80% higher due to the much higher power consumption of the discrete GPU. On the brawny system with GPU, the time improvements are cancelled by the higher power consumption of the system with GPU. Thus, the i7+GPU exhibits energy savings of only 8% for KM. For GR, the usage of GPU always results in worse execution time compared to the CPU-only execution. This, corroborated with the higher power consumption of GPU, leads to energy usages that can be even 14 times higher in case of the Kayla+GPU system, compared to the Kayla-only. This is because GR is less compute-intensive and host-device transfers cannot be overlapped by the fast <*key, value*> processing. Conversely, for BS and KM there is more processing to be done for each <*key, value*> pair and the transfer time is amortized. Based on our analysis of Hadoop-CUDA logs,



(a) i7+GPU



(b) Kayla+GPU

Figure 4.11: Execution time breakdown on systems with discrete GPUs

we compute the proportion of time spent in Hadoop, data transfers to and from GPU and CUDA kernel, as shown in Figure 4.11. For GR, the total time spent in host-device transfers is up to 30% and 85% of total execution time on i7+GPU and Kayla+GPU respectively. In contrast, for BS and KM, this time is around 1% and 4%, respectively. The prohibitively large host-device transfer times make workloads such as GR unsuitable for execution on heterogeneous systems with GPUs.

For an in-depth analysis, we profile the entire execution of MapReduce workloads at the CPU, GPU, memory and storage levels. For CPU, we collect hardware counter values such as instructions, cycles, stall cycles, page faults using *perf* Linux tool attached to the TaskTracker daemon of Hadoop. We use *nvprof* to collect GPU execution metrics such as *warp execution efficiency*, and *dstat* tool available in Linux to get memory and storage usage.

Figure 4.12 shows that the memory utilization of MapReduce is not only proportional to the input size, but is also influenced by the amount of intermediate data generated by Map phase. For example, KM Map generates a large amount of intermediate <*key, value*> pairs and, hence, it uses around 90% of the memory on all systems when run with M and L datasets. Moreover, this is reflected in the amount of data transferred to and from the storage since Hadoop spills some intermediate data on the disk. In case of KM.L, the input size is around 19 GB but the total amount of data moved to and from the storage is more than 100 GB. In contrast, the Map phase of GR generates small <*key, value*> pairs only for matching input records, hence, GR has lower memory and storage utilization. This high memory utilization exposes the limitations of the wimpy systems where Hadoop is constrained by the small memory size. For the same workload, Hadoop uses more than 90% of the 16 GB available on our brawny system, while on the wimpy systems it can use a maximum of 2 GB. Thus, during MapReduce execution on the wimpy systems, the storage is used by the OS virtual memory and by Hadoop spill mechanism. Consequently, there are more page faults and storage transferred bytes for these systems compared to the brawny one, as depicted in Figure 4.12. This fact is more visible on the Jetson+GPU system where the same small memory is shared between CPU and GPU. For example, KM.L on Jetson+GPU exhibits 1.75 times more page faults and transfers 33 GB more data to the storage compared to i7+GPU.

Next, we analyze the IPC as a metric for CPU efficiency. Consistent with our workload description, BS and GR have the highest and the lowest IPC, respectively. Moreover, CPU+GPU has lower IPC compared to CPU-only execution because the most compute-intensive part of the workload is offloaded to GPU while the CPU executes Hadoop housekeeping. But the brawny system is the exception since (i) GR has higher IPC compared to KM and (ii) KM has higher IPC on CPU+GPU than on CPU-only. Firstly, both KM and GR have high memory utilization on a system with high clock frequency, such as i7. At higher clock frequencies, the speed gap between CPU and memory is larger and more CPU cycles are wasted waiting for memory requests to be serviced [81]. These wasted cycles are reported by *perf* as *stall cycles*. For example, BS.L, KM.L and GR.L exhibit

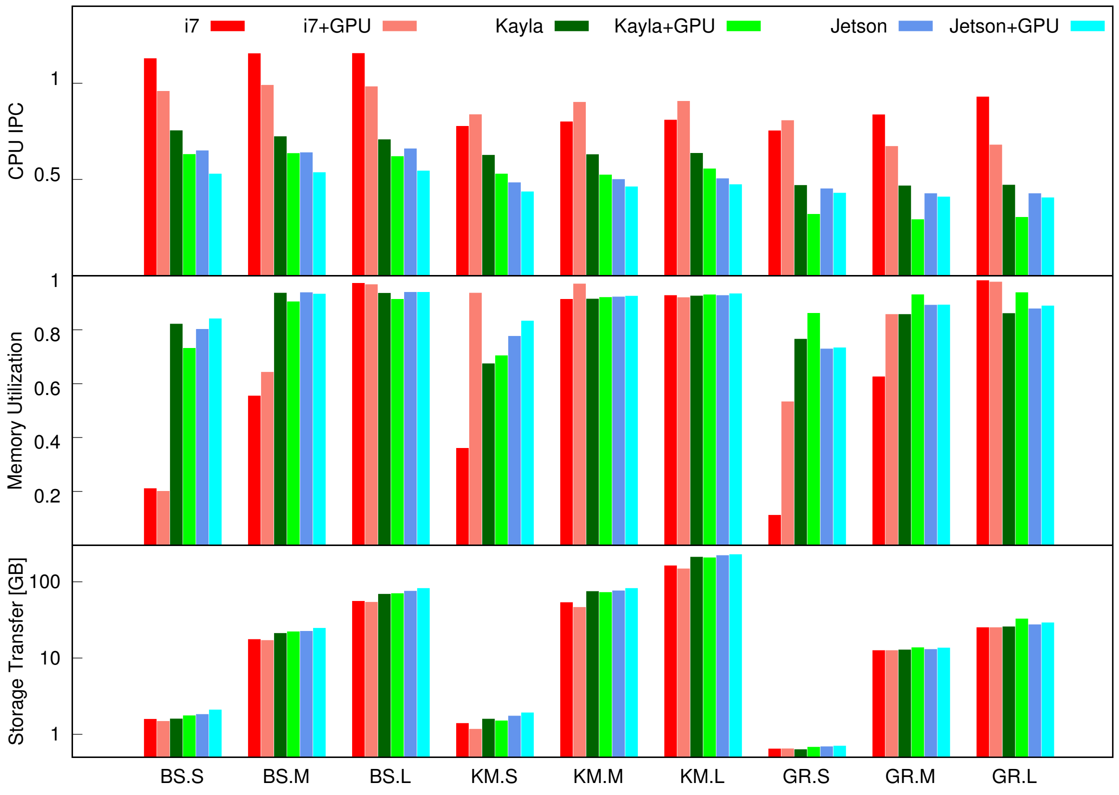


Figure 4.12: MapReduce performance at CPU, memory and storage levels

27%, 43% and 39% stall cycles, respectively, as reported to total cycles. These values show a strong correlation with the IPC. Secondly, KM has higher IPC on CPU+GPU than on CPU-only because the workload exhibiting stall cycles is offloaded to the GPU. Among systems, i7 exhibits the highest IPC since it has a deeper pipeline and bigger issue width. Surprisingly, Cortex-A9 CPU has higher IPC than Cortex-A15 CPU while performing poorer in terms of execution time. We analyze this result based on the fact that the execution time is determined by the total number of executed instructions divided by IPC, clock frequency and utilization,

$$T = \frac{Instructions}{IPC \cdot f \cdot U} \tag{4.1}$$

Kayla and Jetson *perf* logs show that the number of instructions and the utilization are similar. But the clock frequency of Cortex-A15 is almost two times

Table 4.3: GPU profiling

Workload	**Warp execution efficiency** [%]		
	i7+GPU	Kayla+GPU	Jetson+GPU
BS	97	97	77
KM	99	98	61
GR	26	26	17

higher, as indicated in Table A.1, hence, there are more stall cycles due to memory requests. Our measurements indeed show that Cortex-A15 executes more cycles than Cortex-A9 and, thus, has a lower IPC. However, this lower IPC is overcome by the increased frequency, thus, leading to a better execution time.

Lastly, we analyze the execution of the three workloads on GPU using the *warp execution efficiency* metric exposed by *nvprof*. This metric represents the average active threads per warp divided by the maximum number of threads per warp, which is 32 for both GPUs used in our evaluation. CUDA threads in a warp are inactive if they wait for other threads to execute a divergent path or they finished the execution. As expected, GR has the highest proportion of such inactive threads which decrease the efficiency to 17-26%, as shown in Table 4.3. In contrast, BS and KM achieve 61-99% warp execution efficiency.

We conclude that the small memory of wimpy systems hinders MapReduce execution since Hadoop has to use the storage to store and load intermediate data.

4.5.2 Bottleneck Analysis

We use the results from our time-energy analysis, system profiling and system characterization to discuss software and hardware improvements for more efficient MapReduce execution. Firstly, we analyze the scenarios in which the wimpy systems with powerful discrete GPUs, such as Kayla+GPU, become more energy-efficient. This is achievable by either (i) improving execution time or (ii) reducing

GPU power, because energy is the product of execution time and average power,

$$E = T \cdot P \tag{4.2}$$

Our measurements show an average power usage ratio of two between Kayla+GPU and Kayla-only, hence, the execution time of Hadoop-CUDA should exhibit a speedup of at least two. This could be obtained by highly compute-intensive workloads and by an improved Hadoop framework. On the other hand, given the maximum speedup of 1.2 shown by KM, Kayla+GPU becomes more energy-efficient by reducing GPU power by 80%. This power reduction could be achieved by a wimpy platform with integrated GPU, such as Jetson TK1. Indeed, our measurements show that the GK20A GPU on Jetson TK1 consumes around 3 W when active, compared to around 25 W drawn by Maxwell GPU on the Kayla system. Since the speedup on Jetson+GPU is also around 1.2, this configuration exhibits energy savings, as opposed to Kayla+GPU.

Secondly, we discuss the scenarios in which less compute-intensive workloads, such as GR, can benefit from GPU processing. On one hand, GR exhibits a large number of divergent control flow paths which results in high number of inactive CUDA threads. For an input record that does not contain the searched string, a CUDA thread takes longer time to finish the processing. In contrast, for a record containing the string at the beginning, the CUDA thread finishes the processing very fast but must wait for the slowest thread in the warp. Since this is a workload characteristic, one solution is to improve control flow handling on GPUs. On the other hand, GR exhibits little computational work since it consists of scanning the input record, but requires large host-device data transfers. For example, on the brawny system, GR.L spends 30% of the time in data transfers. However, improving transfer bandwidth by three times results in 2% faster execution on

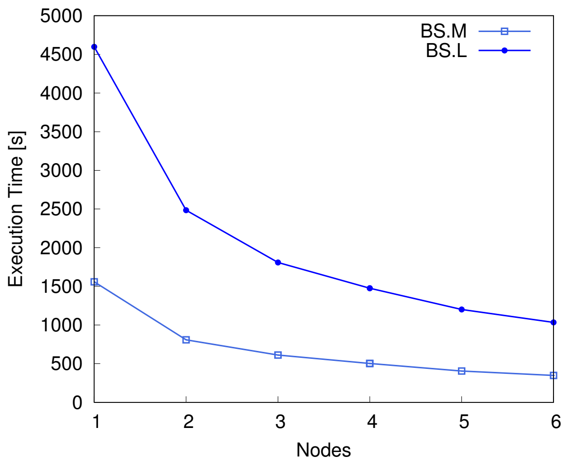
Table 4.4: Performance equivalence ratio

Workload	**Systems**	**Equivalence ratio**	**Savings [%]**	
			Time	**Energy**
BS.M	i7+GPU : Jetson+GPU	1 : 6	5	46
BS.L	i7+GPU : Jetson+GPU	1 : 6	-4	46
KM.M	i7+GPU : Jetson+GPU	1 : 3	-12	67
KM.L	i7+GPU : Jetson+GPU	1 : 3	0	68

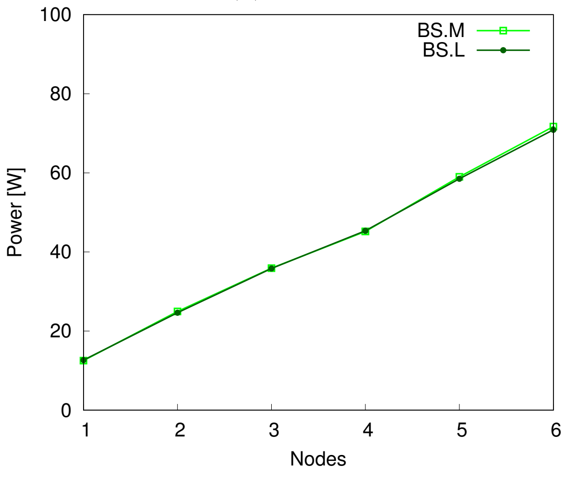
CPU+GPU compared to CPU-only. This improvement is feasible as Nvidia already announced NVLink, a faster and more energy-efficient CPU-GPU path that can achieve a bandwidth of up to 200 GB/s [41].

4.5.3 Time-Energy Performance Equivalence

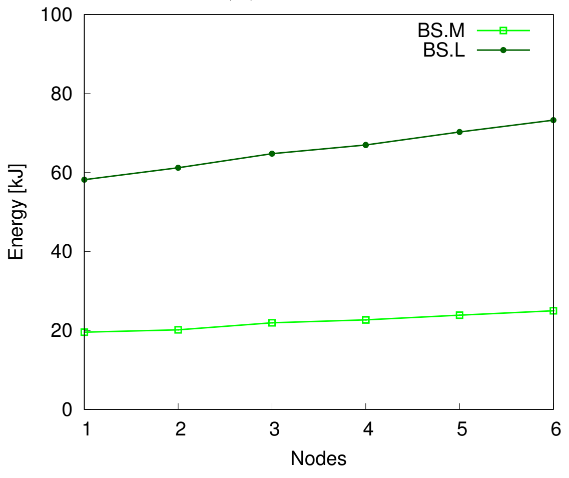
Analyzing the time-energy results across different systems, we observe that Kayla configurations exhibit the highest execution times leading to energies that are similar or higher than those of brawny configurations. On the other hand, even if these brawny configurations exhibit the best execution time for all workloads, *intra-chip* heterogeneous wimpy system always consumes less energy. This result opens the alternative of using multiple Jetson+GPU nodes to perform the work of a single i7+GPU system with potentially less energy. We further analyze this opportunity for compute-intensive workloads with medium and large inputs on heterogeneous configurations. Small inputs are not representative for this analysis due to their small execution time dominated by Hadoop overheads. We measure the execution time and energy for clusters of up to four Jetson+GPU nodes and observe that for KM, three wimpy nodes achieve similar execution times as one i7+GPU, while saving almost 70% energy, as shown in Table 4.4. However, for BS more than four nodes are needed and, using the measured values, we estimate the time and energy using regression analysis.



(a) Time



(b) Power



(c) Energy

Figure 4.13: BS performance on clusters of Jetson TK1

The execution time on n nodes,

$$T(n) = \frac{T(1)}{S(n)} \tag{4.3}$$

is determined by the speedup which should ideally be linear, $S(n) = n$. But in reality, the sequential fraction and the overheads of parallel and distributed execution lead to a sub-linear speedup. For BS, this sub-linear speedup determines an execution time that fits a power function, as shown in Figure 4.13a. Using the estimated execution time, we observe that six wimpy boards achieve the performance of one brawny system. Because energy is the product of execution time and average power, as in Equation 4.2, on n nodes it becomes

$$E(n) = T(n) \cdot P(n) = \frac{T(1)}{S(n)} \cdot n \cdot P(1) = E(1) \cdot \frac{n}{S(n)} \tag{4.4}$$

assuming that the average power grows linearly with the number of nodes. This assumption is true for our workload as shown in Figure 4.13b. Since the speedup is sub-linear, the energy usage slowly increases with the number of nodes, as shown in Figure 4.13c. Even with this small increase, six wimpy nodes save 46% of the energy used by a single brawny system. These results advocate the usage of wimpy systems with integrated GPUs for compute-intensive data analytics.

4.6 Summary

In this section, we have analyzed the time-energy performance of MapReduce on heterogeneous systems with GPUs. With the performance improvements of wimpy systems used in mobile devices, we investigate their performance on processing data analytics compared to brawny server systems. For the cluster-level analysis, we evaluate MoSS, our Hadoop-CUDA framework based on *lazy process-*

ing and *dynamic mapping* techniques. We used six representative workloads and two cluster systems with diverse performance capabilities. First, we evaluate the time-energy performance of MoSS on our in-house low-power wimpy cluster based on Nvidia Jetson TK1 nodes integrating quad-core ARM Cortex-A15 CPUs and 192-core Nvidia Kepler GPUs on the same chip. Second, we evaluate the execution time performance of MoSS on a high-performance brawny cluster based on Amazon EC2 instances equipped with discrete 1536-core GPUs. Compared to Hadoop, MoSS improves the execution time of compute-intensive workloads by factors of up to 3.1 and 2.3 on wimpy and brawny nodes clusters, respectively. Along with the improvement in execution time, MoSS saves up to 80% of the energy used by the wimpy cluster. Moreover, the execution time of MoSS is almost always within 5% of the best Hadoop CPU-only execution time for data-intensive workloads.

For the single-node analysis, we have selected three systems representing both *intra-node* and *intra-chip* heterogeneity, (i) an Intel i7 system hosting a discrete 640-core Nvidia GPU of Maxwell generation, representing *intra-node* heterogeneous brawny systems, (ii) a quad-core ARM Cortex-A9 with the same Maxwell GPU representing *intra-node* heterogeneous wimpy systems, and (iii) a quad-core ARM Cortex-A15 integrated with 192 Nvidia Kepler GPU cores representing *intra-chip* heterogeneous wimpy systems. We evaluate the time and energy performance of these systems using three MapReduce applications with diverse resource demands.

The single-node analysis shows that, for compute-intensive workloads such as BlackScholes, the brawny heterogeneous system achieves speedups of up to 2.3 and reduces the energy usage by almost half compared to the brawny homogeneous system. As expected, for applications such as Grep where data transfers dominate the execution time, heterogeneous systems exhibit worse time-energy performance compared to homogeneous systems. For example, the heterogeneous wimpy Kayla with discrete GPU consumes 14 times the energy of the homogeneous Kayla system due to very low host-device transfer bandwidth and high power overhead of the discrete GPU. Moreover, the lower performance of wimpy systems on data analytics is in part due to the small main memory size. While brawny systems have large memories to accommodate Hadoop's intermediate data, our profiling shows that on both wimpy systems data is spilled to disk leading to 80% more storage transfers compared to the brawny system.

Among heterogeneous systems, the wimpy with discrete GPU exhibits the worst time-energy performance. But the wimpy with integrated GPU uses the lowest energy across all workloads due to more energy-efficient CPU and GPU, and better balanced system resources. To account for the execution time difference, we establish an equivalence ratio between a single brawny heterogeneous node and multiple wimpy heterogeneous nodes. Based on this equivalence, the wimpy nodes not only achieve similar execution times compared to a single brawny node, but also exhibit energy savings of up to two-thirds. This result advocates the potential usage of wimpy systems with integrated GPUs for Big Data analytics.

Chapter 5

Model-driven Performance Analysis

In this chapter, we present our model-driven time-energy performance analysis for scale-out workloads and clusters. While a measurement-based performance analysis approach, such as the one we have presented in Chapter 4, provides highly accurate results and in-depth insights, it also has limited scalability and applicability. For example, measuring the performance of scale-out workloads is time consuming, measuring the performance of scale-out clusters is tedious, and measuring the performance of newly designed systems may be impossible as long as there is no physical implementation. Moreover, data-parallel application developers and users may want to know what is the optimal configuration for running their application before buying a dedicated cluster or compute time in the cloud. We address these issues by designing measurement-driven analytic models for execution time and energy usage of data-parallel MapReduce execution on both homogeneous systems running Hadoop and heterogeneous systems running MoSS. We design a hybrid approach based on both baseline measurements and analytic equations to increase accuracy and ease of use, respectively. Using these models, we analyze (i) our techniques for efficient execution presented in Chapter 3 in a formal manner, (ii) hypothetical system configurations to help designers and developers improve the performance of hardware and software and (iii) scale-out workloads and clusters to compare homogeneous with heterogeneous and brawny with wimpy systems.

5.1 Execution Time Model

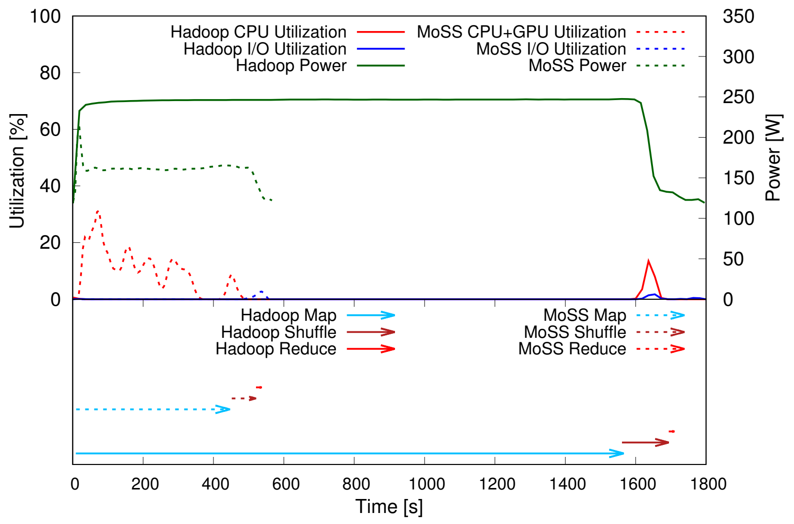
With the notations in Table 5.1, we first present the execution time model for MapReduce execution on both homogeneous systems using Hadoop and heterogeneous systems using MoSS. More details about these systems are presented in Section 4.2 and in Appendix A. As shown in Figure 5.1 for the six applications described in Section 4.1, MapReduce execution consists of three phases, namely Map, Shuffle and Reduce. Shuffle phase starts after the first wave of Map tasks and it is partially overlapped with Map phase. Reduce phase starts after the end of Shuffle phase, as shown in Figure 5.1 and Figure 5.2. Hence, total execution time is the sum of Map time, T_M, non-overlapped Shuffle time, T_S, and Reduce time, T_R,

$$T = T_M + T_S + T_R \tag{5.1}$$

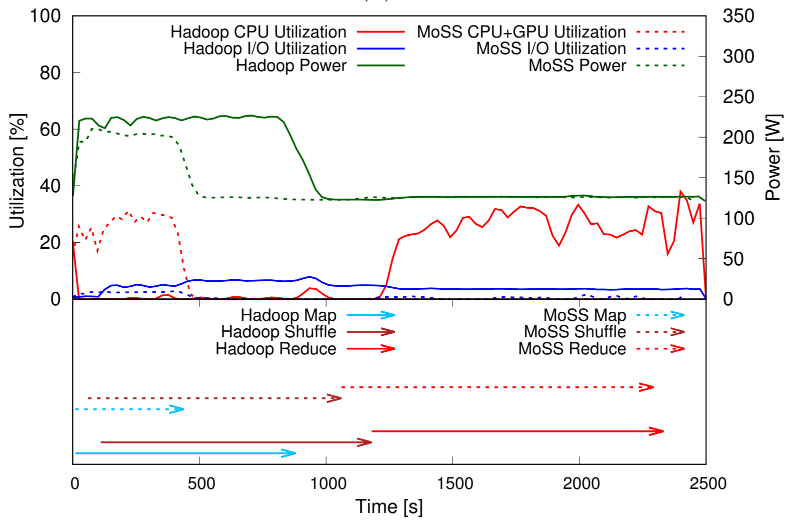
Given a MapReduce application and an input of size S containing R records, this input is divided into chunks of size S_K such that each Map task instance processes one chunk. The number of Map task instances is proportional to the input size,

$$N_{MT} = \left\lceil \frac{S}{S_K} \right\rceil \tag{5.2}$$

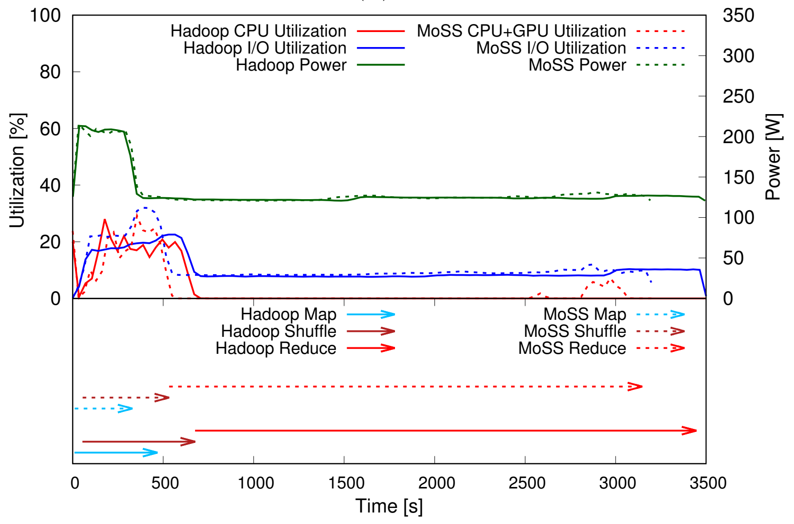
For example, the number of Map tasks for Grep execution depicted in detail in Figure 5.2 is 179, the result of dividing 11.18 GB of input size to 64 MB, the default



(a) PI

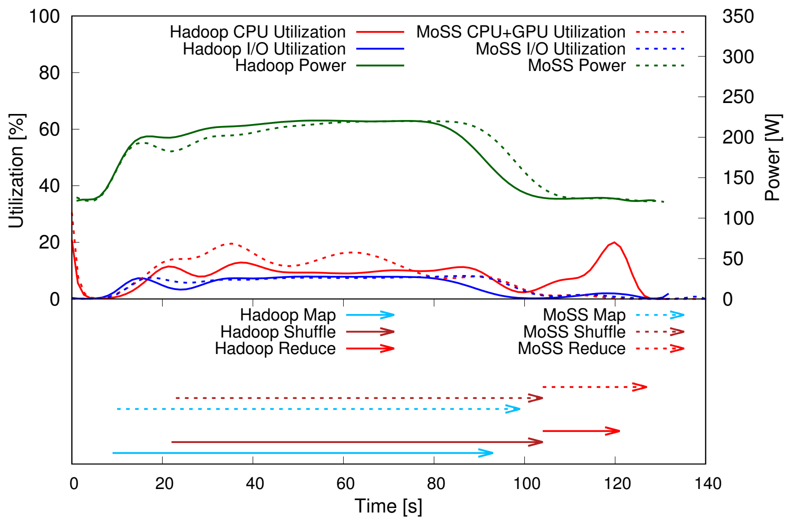


(b) BS

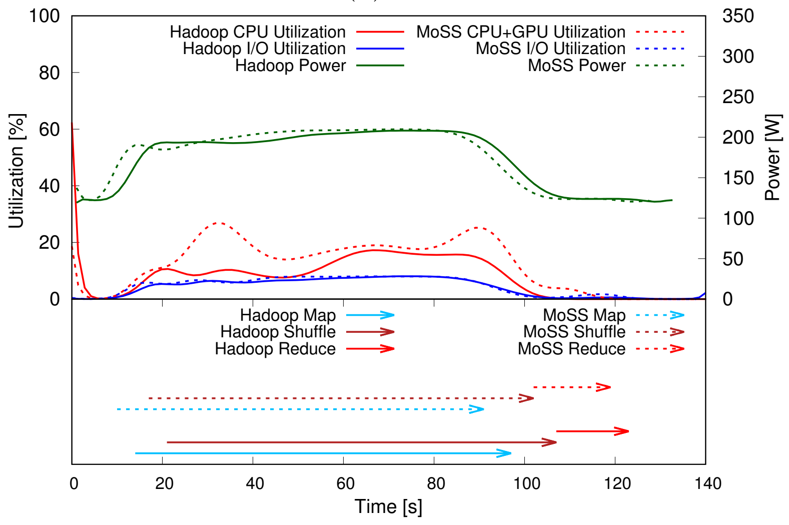


(c) KM

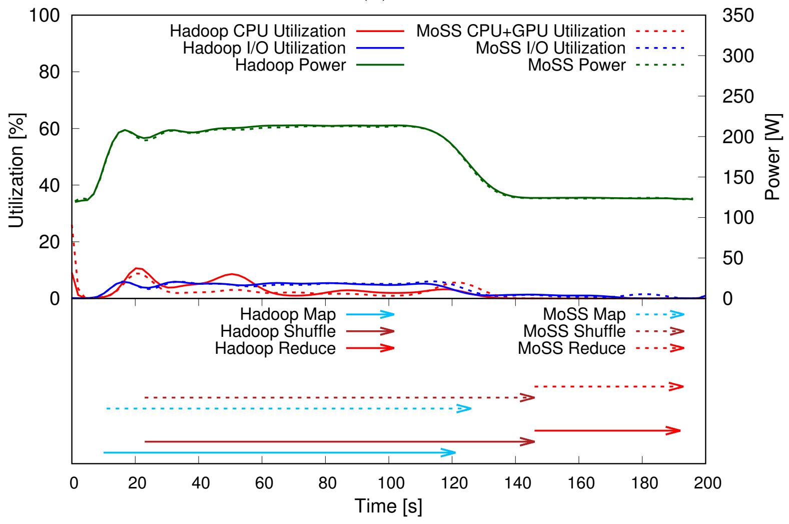
Figure 5.1: Hadoop and MoSS execution on 12-node Jetson TK1 cluster



(d) MM



(e) SS



(f) GR

Figure 5.1: Hadoop and MoSS execution profile on 12-node Jetson TK1 cluster

Table 5.1: Notations

Symbol	**Description**
	Application
S	input size
R	number of input records[1]
S_M	Map output size
R_M	Map output records
	Framework
S_K	input split size
ς_M	Map slots per node
ς_R	Reduce slots per node
$N_{\mathcal{P}T}$	number of tasks for phase $\mathcal{P}$[2]
r	number of input records per task
	System
n	number of cluster nodes
c	CPU cores per cluster node
g	GPU cores per cluster node
d	generic notation for CPU/GPU threads
h	GPU threads per cluster node
	Baseline run
S_B	input size of baseline run
$S_{M,B}$	size of Map output for baseline run
$R_{M,B}$	Map phase output records for baseline run
$T_{Mr}^{\mathcal{U}}$	execution time of one Map record on execution unit $\mathcal{U}$[3]
$T_{Rr}^{\mathcal{U}}$	execution time of one Reduce record on execution unit $\mathcal{U}$
$P_{\mathcal{P}}$	average power usage during phase $\mathcal{P}$
B_{HDFS}	HDFS I/O bandwidth
$\sigma(n)$	scaling as function of node count
	Model
T	execution time
$T_{\mathcal{P}}$	execution time of phase $\mathcal{P}$
$T_{\mathcal{P}}^{\mathcal{U}}(R,t)$	execution time of phase $\mathcal{P}$ on execution unit $\mathcal{U}$ using t threads
$T_{\mathcal{P}T}^{\mathcal{U}}(r,t)$	execution time of a task on execution unit $\mathcal{U}$ using t threads
$T_{\mathcal{P}r}^{\mathcal{U}}$	processing time of one record during phase $\mathcal{P}$ on $\mathcal{U}$
$T_{I/O}^{\mathcal{U}}$	time to read/write records for processing on $\mathcal{U}$
$K_{\mathcal{P}}$	proportionality factors for phase $\mathcal{P}$
E	energy

[1] We use the terms <*key,value*> and *record* interchangeably.
[2] Phase $\mathcal{P}$ can be Map(M), Shuffle(S), or Reduce(R).
[3] Execution unit $\mathcal{U}$ can be CPU or GPU.

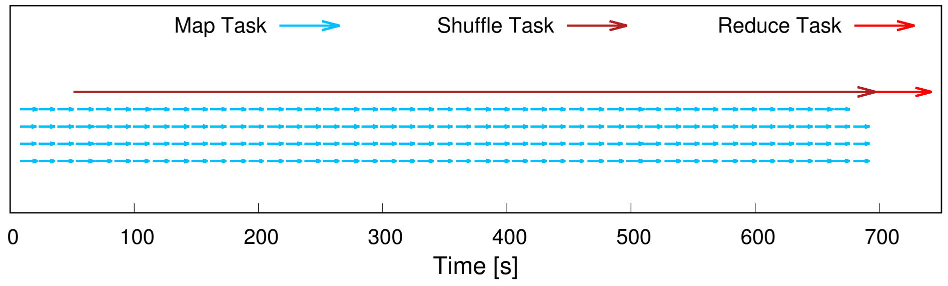


Figure 5.2: Detailed MapReduce execution of Grep on one cluster node

split size in Hadoop. These Map tasks are executed in waves on a number of slots, ς_M, specified in Hadoop's *mapred-site.xml* configuration file. A best practice is to set this number of slots equal to the number of CPU cores on a cluster node [72]. In our example depicted in Figure 5.2, Jetson TK1 has four cores, thus, Map tasks are executed in waves of four tasks. Each Map task processes r records sequentially by applying user-defined `map()` function, and produces none, one or multiple <*key, value*> pairs. These pairs are sorted based on their keys to form lists of pairs with the same key. Each Reduce task processes one of these lists of pairs by applying user-defined `reduce()` function, and produces the final output <*key, value*> pairs.

Assumption 1. *Typical MapReduce workloads have input records of small size and this record size is roughly constant across different input sizes.*

Further description of MapReduce execution is based on Assumption 1 which is in correlation with works analyzing real MapReduce traces from Facebook and Yahoo, among others [29,63]. For these workloads, each Map task processes roughly the same amount of records,

$$r \simeq \frac{R}{N_{MT}} \simeq \frac{R \cdot S_K}{S} \tag{5.3}$$

and takes roughly the same amount of time to finish, as shown in Figure 5.2 for Grep. Hence, Map phase time distributed on a cluster with n nodes, where each node allocates ς_M Map task slots and uses execution unit $\mathcal{U}$ with d threads is

$$T_M^{\mathcal{U}}(R,d) = \left\lceil \frac{N_{MT}}{n \cdot \varsigma_M} \right\rceil \cdot T_{MT}^{\mathcal{U}}(r,d) \tag{5.4}$$

In Hadoop running only on the CPU, Map task time is

$$T_{MT}^{CPU}(r,d) = r \cdot T_{Mr}^{CPU} \tag{5.5}$$

since all records are processed sequentially by the task using a single CPU thread. When running MoSS on a GPU with d threads, Map task time is

$$T_{MT}^{GPU}(r,d) = \left\lceil \frac{r}{d} \right\rceil \cdot T_{Mr}^{GPU} \tag{5.6}$$

since d records are processed at a time using the *lazy processing* technique. While T_{Mr}^{GPU} may vary with the number of threads used, d, once this thread count is set, we assume the processing time per record remains constant and is not affected by the GPU's internal architecture. Moreover, this processing time per record remains constant for a given application across different inputs with the same record size. Hence, we make the following assumption for which we present empirical evidence in Section 5.3.

Assumption 2. *Processing time for one record during Map phase, T_{Mr}, is constant across different input sizes.*

In summary, Map phase time is proportional to the number of input records and the time to process one record and scales with the number of threads per task, d, and with a scaling factor $\sigma(n)$ specific to a cluster with n nodes,

$$T_M = K_M \cdot \frac{1}{\sigma(n)} \cdot \frac{1}{d} \cdot R \cdot T_{Mr} \tag{5.7}$$

where $d = 1$ CPU thread for homogeneous Hadoop execution, and $d = h$ GPU threads for heterogeneous MoSS execution.

Map phase produces R_M intermediate <*key,value*> pairs, with a total size of S_M bytes. These pairs are shuffled and sorted based on their keys to form lists of pairs having the same key. Each Reduce task processes one of these lists by applying user-defined `reduce()` function to produce the final <*key,value*> pairs. R_M and S_M are application-dependent and proportional with input size S. For the applications evaluated in this thesis, R_M and S_M grow linearly with S, as shown in Section 5.3.

Assumption 3. *Map phase output size, S_M, and number of output records, R_M, grow linearly with input size, S.*

To determine the values of S_M and R_M for a given input size, we perform baseline runs on a smaller input, S_B, and use linear regression,

$$R_M(S) = \frac{S}{S_B} \cdot R_{M,B} \tag{5.8}$$

$$S_M(S) = \frac{S}{S_B} \cdot S_{M,B} \tag{5.9}$$

Shuffle phase partially overlaps with Map phase, as shown in Figure 5.1. The non-overlapped part of Shuffle phase is due to the transfer of the <*key,value*> pairs produced by the last wave of Map tasks. We estimate the output size of this last wave of Map phase as being proportional with the input split size, S_K, reported to input size, S. Since there are ς_M Map tasks producing this <*key,value*> pairs, and considering HDFS bandwidth per node, B_{HDFS}, the non-overlapped Shuffle time is

$$T_S = K_S \cdot \varsigma_M \cdot \frac{S_K}{S} \cdot S_M(S) \cdot \frac{1}{B_{HDFS}} \tag{5.10}$$

Using Equation 5.9, we obtain

$$T_S = K_S \cdot \varsigma_M \cdot \frac{S_K}{S} \cdot \frac{S}{S_B} \cdot S_{M,B} \cdot \frac{1}{B_{HDFS}} \tag{5.11}$$

Reduce phase time is determined, among others, by application and input characteristics, and it is proportional with the output of Map phase that we suppose to grow linearly with the input size. Similar to Map phase, we assume the time to process one record during Reduce phase to be constant across problem size.

Assumption 4. *Processing time for one record during Reduce phase, T_{Rr}, is constant across different input sizes.*

This assumption holds for the applications evaluated in this paper, as shown in Section 5.3. In practice, the user must adapt the model to the characteristics of a specific MapReduce application. Reduce phase time is

$$T_R = K_R \cdot \frac{1}{\sigma(n)} \cdot R_M(S) \cdot T_{Rr} \tag{5.12}$$

Using Equation 5.8, we obtain

$$T_R = K_R \cdot \frac{1}{\sigma(n)} \cdot \frac{S}{S_B} \cdot R_{M,B} \cdot T_{Rr} \tag{5.13}$$

5.2 Energy Model

The energy model is derived based on the observation that the power utilization is proportional to system utilization and that during Map phase, this utilization is usually more than 90%, while during Shuffle and Reduce phase it is low, typically less than 10%, as shown in Figure 5.1. This observation is based on analyzing a large volume of MapReduce executions on different cluster systems. Thus, we model the total energy as being the sum of energies during each phase, multiplied by the number of cluster nodes,

$$E = n \cdot (T_M \cdot P_M + T_S \cdot P_S + T_R \cdot P_R) \tag{5.14}$$

To increase the accuracy, we measure during baseline runs the average power values for MapReduce execution phases, P_M, P_S and P_R. In case the user does not have access to power measurements but she knows systems specifications for idle and peak power, she can approximate the power as function of utilization for Map, Shuffle and Reduce phases.

5.3 Validation

In this section, we validate the predicted time and energy values against measured values. We start by discussing model parameterization using baseline runs, and show empirical evidence for some of the assumptions made in the previous sections.

Our hybrid time-energy analytic models use measured parameters to increase accuracy. These parameters, listed in Table 5.1 under Baseline run, are measured during applications execution using the small input size B described in Section 4.1. We summarize parameter values for both Hadoop and MoSS on three system configurations in Table 5.2. These configurations consist of Jetson TK1 using four Map slots, i7 using eight Map slots and Amazon using eight Map slots. All these systems allocate one slot for Reduce tasks. Power values are not available for cloud instances because Amazon does not provide power and energy measurements.

All baseline parameters can be measured on single-node setups, except for HDFS bandwidth, B_{HDFS}, and for the scaling function $\sigma(n)$, which need to be measured on clusters of multiple nodes. We show in the Appendix B, Section B.2, that HDFS performance at single-node and cluster level are different due to the

Table 5.2: Models parameters

App.	**System**	$S_{M,B}$ [MB]	$R_{M,B}$	T_{Mr}^{Hadoop} [μs]	T_{Mr}^{MoSS} [μs]	T_{Rr}^{Hadoop} [μs]	T_{Rr}^{MoSS} [μs]	P_{M}^{Hadoop} [W]	P_{M}^{MoSS} [W]	$P_{S/R}^{Hadoop}$ [W]	$P_{S/R}^{MoSS}$ [W]	$\sigma(n)$
	Jetson			1852.0	527.0	7.0	6.0	20.1	14.1	12.6	12.7	0.86n+0.25
PI	i7	16.2	1,000,000	336.0	343.0	6.0	6.0	105.9	122.2	55.9	61.8	0.69n+0.49
	Amazon			393.0	196.5	6.0	7.0	-	-	-	-	0.69n+0.49
	Jetson			27.4	13.0	5.4	5.3	18.2	17.5	17.6	16.3	0.33n+0.72
BS	i7	923.3	60,000,000	5.2	1.9	0.8	0.8	103.7	135.7	102.2	121.7	0.24n+1.06
	Amazon			6.3	2.1	1.4	1.4	-	-	-	-	0.24n+1.06
	Jetson			24.1	26.4	2.6	2.7	16.2	16.5	16.0	15.5	0.35n+1.20
KM	i7	4,119.4	41,694,000	4.3	2.3	0.9	1.5	99.6	103.3	80.0	112.2	0.04n+0.80
	Amazon			5.2	3.2	1.0	1.0	-	-	-	-	0.04n+0.80
	Jetson			234.6	235.8	7.4	8.6	15.8	16.0	16.8	17.0	0.58n+0.73
MM	i7	13.1	810,000	101.2	116.0	7.4	8.6	73.4	85.1	88.6	110.6	0.49n+0.84
	Amazon			108.6	129.6	8.6	7.4	-	-	-	-	0.49n+0.84
	Jetson			234.6	244.4	7.4	8.6	16.0	16.0	16.8	16.7	0.58n+0.73
SS	i7	10.6	810,000	102.5	109.9	7.4	8.6	75.0	97.9	89.4	110.6	0.49n+0.84
	Amazon			111.1	129.6	7.4	8.6	-	-	-	-	0.49n+0.84
	Jetson			4.2	4.2	1.6	1.1	16.2	16.5	17.1	17.1	0.58n+0.30
GR	i7	76.4	10,019,282	1.2	1.3	0.6	0.7	92.0	95.1	95.5	115.6	0.49n+0.84
	Amazon			1.4	1.4	0.6	0.6	-	-	-	-	0.49n+0.84

effects of networking and replication. Similarly, we show in Chapter 4 and Appendix B that Hadoop does not scale with the identity function, and, thus, we approximate scaling factors using linear equations

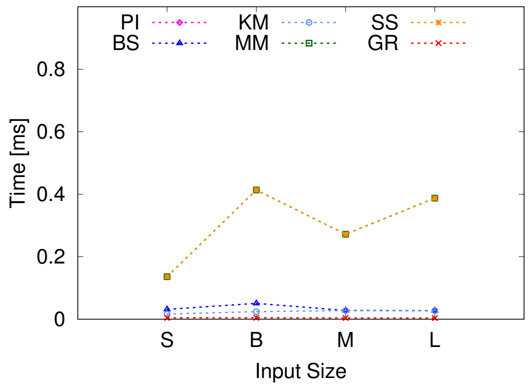
$$\sigma(n) = a \cdot n + b \tag{5.15}$$

and list them in Table 5.2. We assign Amazon's scaling factor to i7 since we have a single node of this type for baseline runs and since these two systems have similar specifications and exhibit similar performance, as shown by the baseline runs values.

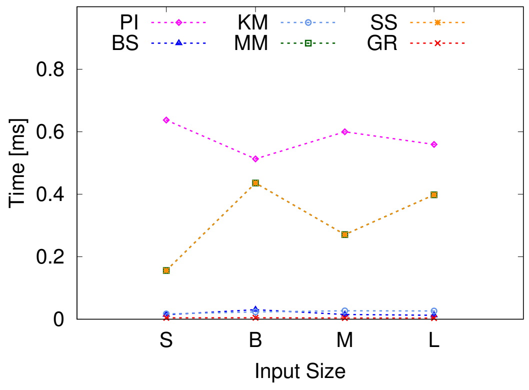
Next, we validate three assumptions made during model description using empirical evidence.

Assumption 2. *Processing time for one record during Map phase, T_{Mr}, is constant across different input sizes.*

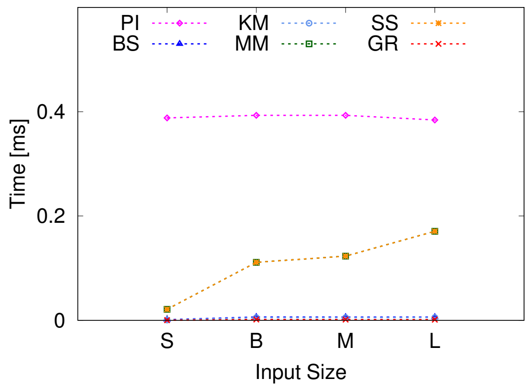
To validate this assumption, we measure T_{Mr} for all applications across all four input sizes and plot the results in Figure 5.3. We observe a higher variation among results for MM and SS which can be explained by the fact that record size is slightly different across input size for these two workloads, as shown in Table 4.1.



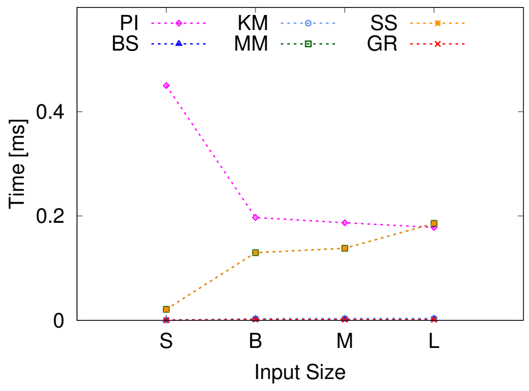
(a) Hadoop, Jetson TK1



(b) MoSS, Jetson TK1



(c) Hadoop, Amazon EC2



(d) MoSS, Amazon EC2

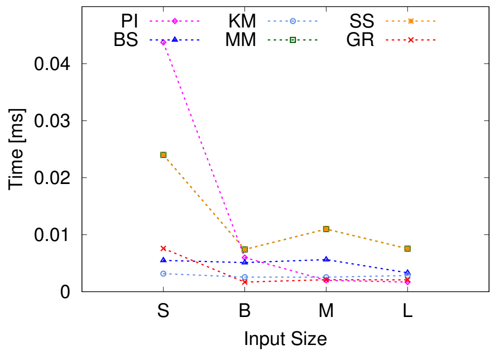
Figure 5.3: Map record processing time

In addition, the results plotted in Figure 5.3 support our decision to use size B for baseline runs since model parameters have values similar to those of input sizes M and L.

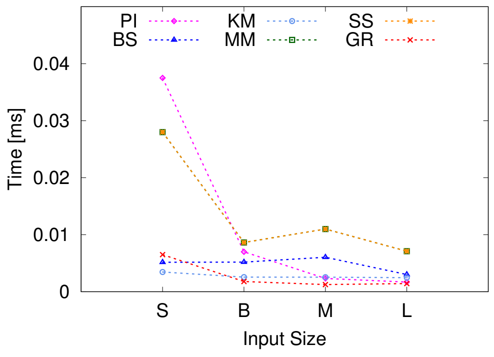
Assumption 3. *Map phase output size, S_M, and number of output records, R_M, grow linearly with input size, S.*

To validate this assumption used by Equations 5.8 and 5.9, we have computed Pearson correlation coefficients (PCC) between S and S_M, and S and R_M respectively, for all workloads. All computed coefficients are greater than 0.99, suggesting a very strong correlation between the linearity of the input size and the Map output size and number of records.

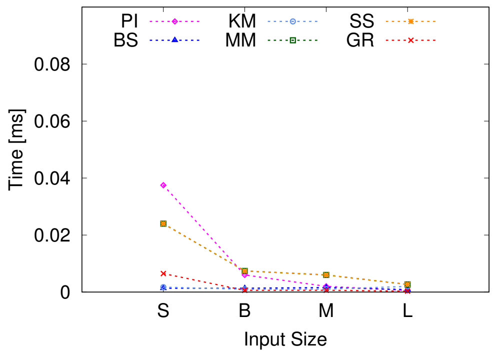
Assumption 4. *Processing time for one record during Reduce phase, T_{Rr}, is con-*



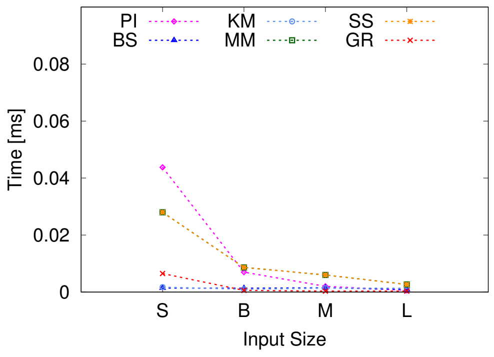
(a) Hadoop, Jetson TK1



(b) MoSS, Jetson TK1



(c) Hadoop, Amazon EC2



(d) MoSS, Amazon EC2

Figure 5.4: Reduce record processing time

stant across different input sizes.

To validate this assumption, we measure T_{Rr} for all applications across all four input sizes and plot the results in Figure 5.4. Counter to intuition, we observe higher values for T_{Rr} on input size S, while for the other sizes the values are similar. This could be explained by the Sort phase which is included in the Reduce phase of Hadoop. For small sizes, sort cannot be properly overlapped with reduce function calls, hence, the overall efficiency of Reduce phase is lower. Similar to the profiling of T_{Mr}, the results for T_{Rr} show that size B is more suitable for baseline runs to get input parameters with values similar to input sizes M and L.

Lastly, we present average time model error on a total of 264 configurations and average energy model error on a total of 192 configurations. This number of configurations is determined when using six applications, two input sizes for

Table 5.3: Models error

	Jetson (2/1 slots)		Jetson (4/1 slots)		i7 (4/1 slots)		i7 (8/1 slots)		Amazon (8/1 slots)
	Time [%]	Energy [%]	Time [%]	Energy [%]	Time [%]	Energy [%]	Time [%]	Energy [%]	Time [%]
PI	14.0	11.8	7.7	9.8	2.5	8.7	1.2	1.1	14.9
BS	6.6	6.6	16.0	20.5	4.0	4.9	3.6	1.7	19.7
KM	19.7	21.5	18.9	17.8	5.1	12.3	7.1	9.4	19.9
MM	10.3	13.0	19.2	16.9	13.4	19.1	9.4	22.4	13.6
SS	10.8	13.9	15.4	11.0	13.8	15.4	8.8	23.3	12.9
GR	8.4	10.2	19.7	15.6	19.5	15.8	7.3	21.9	18.3

each application, three cluster size for Jetson and Amazon, consisting of one, six and twelve nodes, one cluster size for single-node i7, and two framework, namely Hadoop and MoSS. Moreover, we configure two and four slots for Map tasks on Jetson, while on i7 we configure four and eight Map tasks slots. All systems allocate one slot for Reduce tasks. Since we cannot measure the energy in the cloud, there are 192 configurations for energy validation, in contrast with 264 for time validation.

We compute model error percentage as the difference between measured and predicted value, reported to the measured value. Table 5.3 shows the percentage error for each application on three cluster configurations. Amazon EC2 is not reporting the energy usage, thus, we show only execution time validation. The highest average time error per workload is 19.9% for KM on Amazon, while the highest error for predicted energy is 23.3% for SS on i7. We observe that higher errors are encountered by applications with significant I/O requirements, such as KM, SS and GR. These applications also exhibit higher variation among different runs, as shown in Section 4.4. In summary, the overall average errors across all configurations for execution time and energy usage are 13.3% and 14.1%, respectively. These values are comparable with those reported by related works, as presented in Chapter 2, Section 2.2. However, the accuracy of our models could be improved by (i) modeling Map and Reduce processing time per record in detail by including CPU, GPU, memory and I/O parameters, (ii) measuring these system parameters using hardware counters, (iii) using advanced mathematical modeling techniques, such as the LWLR technique used by HP+ [64]. In contrast to other works [54, 55, 93, 94], we avoid the high overhead due to in-depth profiling. Moreover, our models are easy to apply since they do not require additional Hadoop or MoSS code for profiling.

5.4 Formal Model-driven Analysis

5.4.1 Formal Analysis of Lazy Processing

In this section, we use our models to formally show that (i) lazy processing is better than chunking for inputs with low record size skew, and that (ii) selecting a thread count value that achieves minimum execution time on small inputs, also achieves minimum execution time on larger inputs with similar record size.

Statement 1. *Given a MapReduce application and an input with low record size skew, lazy processing is faster than chunking.*

Proof. Let each Map task process r records of relatively the same size, as stated in Assumption 1, on h GPU threads. Without losing generality, we suppose r is a multiple of h. Lazy processing takes h records at a time, transfers them to the GPU, processes them on the GPU, collects and outputs the results on the CPU. By developing Equation 5.6, task time for lazy processing[1] is

$$T_{MT}^{LP}(r,h) = \frac{r}{h} \cdot (h \cdot T_{I/Or} + T_{Mr}) \tag{5.16}$$

where data transfer and result outputting for each record is depicted by $T_{I/Or}$. In contrast, chunking partitions each record, sends it for processing on h treads, col-

[1]We use LP and CK superscripts to denote lazy processing and chunking, respectively.

lects and computes the final result before outputting. Thus, the time for chunking technique is

$$T_{MT}^{CK}(r,h) = r \cdot (T_{setup} + T_{I/Or} + \frac{T_{Mr}}{h}) \tag{5.17}$$

where partitioning and final result computing are denoted by T_{setup}. We can safely assume that transfer and output times, $T_{I/O}$, are the same for both techniques because they are working with the same amount of data. Moreover, we can assume that the time for processing one record on h threads is $\frac{T_{Mr}}{h}$ for chunking. However, we cannot neglect the effect of partitioning and result collection. For example, when Grep application processes one input line on multiple threads, the developer needs to pay attention to word boundary when splitting the input and must perform result reduction before outputting the final result. Thus, task time for chunking is generally higher than task time for lazy processing due to setup time for input chunking and output collection. □

Statement 2. *Given a MapReduce application and two inputs of the same type, selecting a thread count value that achieves minimum execution time on a small input also achieves minimum execution time on larger inputs.*

Proof. Let S_1 be the small input size and S_2 the larger input size, with corresponding number of records R_1 and R_2, respectively. Since the inputs are of the same type, record processing time is the same across input sizes, as stated in Assumption 2 and validated in Section 5.3.

Our hypothesis states that there is no h' to achieve better execution with the small input,

$$\nexists h', h' \neq h : T_M^{GPU}(R_1, h') < T_M^{GPU}(R_1, h) \tag{5.18}$$

However, suppose there is a h'' that achieves better execution with the large input,

$$\exists h'', h'' \neq h : T_M^{GPU}(R_2, h'') < T_M^{GPU}(R_2, h) \tag{5.19}$$

Using Equation 5.4 and Equation 5.6, we obtain that

$$\begin{aligned} T_M^{GPU}(R_2,h'') < T_M^{GPU}(R_2,h) \Rightarrow \\ \left\lceil \frac{r_2}{h''} \right\rceil \cdot T_{Mr}^{GPU} < \left\lceil \frac{r_2}{h} \right\rceil \cdot T_{Mr}^{GPU} \Rightarrow \\ \left\lceil \frac{r_1}{h''} \right\rceil \cdot T_{Mr}^{GPU} < \left\lceil \frac{r_1}{h} \right\rceil \cdot T_{Mr}^{GPU} \Rightarrow \\ T_M^{GPU}(R_1,h'') < T_M^{GPU}(R_1,h) \end{aligned} \tag{5.20}$$

which contradicts our hypothesis in Equation 5.18. □

In practice, we observe that not only the same value of h achieves minimum execution time across workload sizes, but also across different workloads, as shown in Section 4.3.1.

5.4.2 Formal Analysis of Dynamic Techniques

We formally analyze the cases where overlapping does not achieve better performance compared to selecting the best execution unit between the CPU and GPU. When overlapping CPU and GPU processing the execution time is

$$\begin{aligned} T_{MT}^{Overlapping}(r,h) = p \cdot T_{I/O}^{CPU} + (r-p) \cdot T_{I/O}^{CPU+GPU} \\ +max(p \cdot T_{Mr}^{CPU}, \left\lceil \frac{r-p}{h} \right\rceil \cdot T_{Mr}^{GPU}) \end{aligned} \tag{5.21}$$

where p is the total number of records processed on the CPU. We analyze the cases where overlapping time is equal or higher than the best time of CPU- and GPU-only execution,

$$T_{MT}^{Overlapping}(r,h) \geq min(T_{MT}^{CPU}(r,1), T_{MT}^{GPU}(r,h)) \tag{5.22}$$

For simplicity and without affecting the analysis, we suppose that r is a multiple of h. There are two cases to analyze:

1. Record processing achieves better performance on the CPU compared to the GPU

$$T_{Mr}^{CPU} \leq T_{Mr}^{GPU} \; and \; T_{MT}^{CPU} \leq T_{MT}^{GPU} \tag{5.23}$$

There are two sub-cases for evaluating the maximum in Equation 5.21:

(a) The processing of p records on the CPU takes longer that the processing of $r - p$ records on the GPU, hence Equation 5.21 becomes

$$\begin{aligned} T_{MT}^{Overlapping}(r, h) = p \cdot (T_{I/O}^{CPU} + T_{Mr}^{CPU}) \\ +(r - p) \cdot T_{I/O}^{CPU+GPU} \end{aligned} \tag{5.24}$$

Applying the inequality in Equation 5.22 results in

$$T_{I/O}^{CPU+GPU} - T_{I/O}^{CPU} \geq T_{Mr}^{CPU} \tag{5.25}$$

a highly-encountered case in real world where transfer times between main memory and GPU memory are higher than processing times [71].

(b) The processing of $r - p$ records on the GPU takes longer that the processing of p records on the CPU,

$$p \cdot T_{Mr}^{CPU} \leq \left\lceil \frac{r - p}{h} \right\rceil \cdot T_{Mr}^{GPU} \tag{5.26}$$

and Equation 5.21 becomes

$$\begin{aligned} T_{MT}^{Overlapping}(r, h) \leq p \cdot T_{I/O}^{CPU} \\ +(r - p) \cdot (T_{I/O}^{CPU+GPU} + \frac{1}{h} \cdot T_{Mr}^{GPU}) \end{aligned} \tag{5.27}$$

Applying the inequality in Equation 5.22 results in

$$\begin{aligned} \left\lceil \frac{r-p}{h} \right\rceil \cdot T_{Mr}^{GPU} + \\ (r-p)(T_{I/O}^{CPU+GPU} - T_{I/O}^{CPU}) \geq r \cdot T_{Mr}^{CPU} \end{aligned} \tag{5.28}$$

which means that the processing of all r records on the CPU should be much faster than processing $r-p$ records on the GPU.

2. Record processing achieves better performance on the GPU compared to the CPU

$$T_{Mr}^{CPU} \geq T_{Mr}^{GPU} \ and \ T_{MT}^{CPU} \geq T_{MT}^{GPU} \tag{5.29}$$

There are two sub-cases for evaluating the maximum in Equation 5.21:

(a) The processing of p records on the CPU takes longer that the processing of $r-p$ records on the GPU,

$$p \cdot T_{Mr}^{CPU} \geq \left\lceil \frac{r-p}{h} \right\rceil \cdot T_{Mr}^{GPU} \tag{5.30}$$

Applying the inequality in Equation 5.22 results in

$$\begin{aligned} p \cdot T_{Mr}^{CPU} - \left\lceil \frac{r}{h} \right\rceil \cdot T_{Mr}^{GPU} \geq \\ p \cdot (T_{I/O}^{CPU+GPU} - T_{I/O}^{CPU}) \geq 0 \end{aligned} \tag{5.31}$$

which implies that the processing time of one record is much larger on the CPU than the GPU.

(b) The processing of $r-p$ records on the GPU takes longer that the processing of p records on the CPU. Applying the inequality in Equation 5.22 results in

$$T_{Mr}^{GPU} \geq T_{I/O}^{CPU+GPU} - T_{I/O}^{CPU} \tag{5.32}$$

an ideal case for GPU execution where the computation takes longer than the communication represented by data transfers.

In summary, the overlapping exhibits worse performance compared to selecting the best execution unit between the CPU and GPU when there is a large imbalance between system resources. This happens either when CPU processing is much faster than GPU processing, or when GPU processing is much faster than CPU processing but the computation time on GPU is higher than data transfer and loading times.

5.5 System Profile Analysis

In this section, we are analyzing the effect of system profile on the time-energy performance of Hadoop and MoSS on Jetson TK1 system. Firstly, to analyze the influence of storage system, we are comparing the hard-disk (HDD) used in our evaluation of MoSS with a solid-state drive (SSD). A stand-alone characterization using *dd* Linux tool shows that the SSD has 83% and 56% higher read and write throughput compared to the HDD, respectively. But from HDFS perspective, SSD read throughput is four times higher than HDD throughput while write throughput is similar. On the other hand, HDFS throughput is at least four times lower compared to raw throughput, showing the poor performance of HDFS layer which is not able to fully utilize storage capabilities. From power and energy perspective, the SSD is more efficient since it decreases the idle power of Jetson TK1 from 6.5 W to 3.8 W.

Analyzing the effect of replacing the storage on MapReduce workloads, we observe that the execution time remains almost the same, except for KM which shows an improvement of 10% when using the SSD. As we explained in Section 4.4.1, KM has mixed system resource demands, thus, using a faster storage improves its overall performance. For the other workloads, I/O operations are mostly overlapped with CPU/GPU processing. On the other hand, these results show that Hadoop is not able to exploit the advantage of faster storage devices.

From energy perspective, the system with SSD saves between 18% and 33% of the energy used by the system with HDD. This is almost exclusively due to the lower power profile of the SSD, rather that improvements in execution time. Because the system with SSD is more energy-efficient, we are using it for the remaining of this section.

Secondly, we are analyzing the hypothetical case of increasing system's performance such that Map and Reduce record processing time are improved by a factor β. However, increasing system's performance leads to higher power usage. Hence, we determine the maximum increase in processing power such that the overall energy usage of the new system is at most the same as the energy usage of the initial system. We assume that idle power, $P_I = 3.8\ W$, remains the same and that processing power is always higher than idle power. The processing power of the initial Jetson system is computed as the average across all workloads, $P_P = 10.4\ W$. Given a fixed time, T, the system does useful work for a time T_P and stays idle for a time T_I such that $T_P + T_I = T$. The improved system does the processing faster, $T_P^{'} = \frac{T_P}{\beta}$, and remains idle for the rest of the time, $T_I^{'} = 1 - \frac{T_P}{\beta}$. We want the energy usage of the improved system to be at most equal to the energy usage of the initial system, $E^{'} \leq E$, during the time period T. Solving this inequality, we obtain

$$P_P^{'}(\beta) \leq \beta \cdot P_P + (1 - \beta) \cdot P_I \tag{5.33}$$

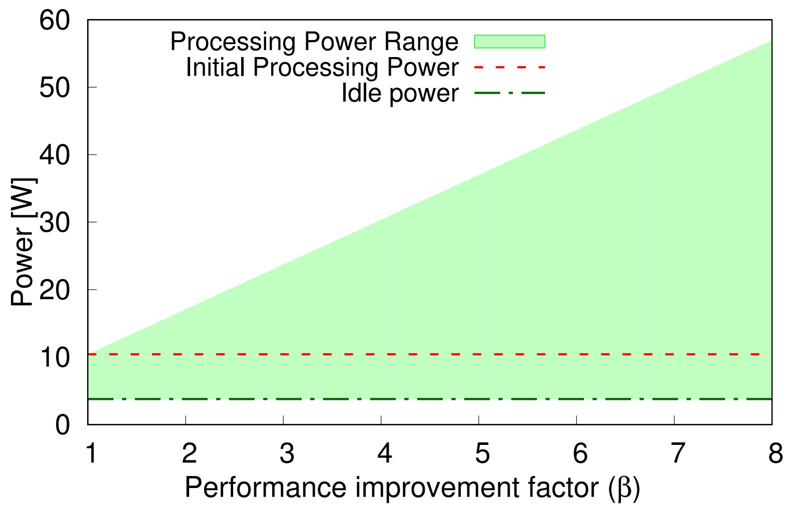


Figure 5.5: System power profile

We plot this inequality in Figure 5.5 where the highlighted area represents the possible values for the processing power of the improved system. For example, if the new system is eight times faster, it can use up to 57 W, almost six times more than the initial system, and still consume the same energy. Through this type of analysis, our models can help the designing of future energy-efficient systems able to process data much faster.

5.6 Model-driven Analysis of Scale-out Workloads and Systems

In this section, we compare heterogeneous with homogeneous and wimpy with brawny systems to determine time-energy-efficient configurations for executing scale-out data-parallel workloads on scale-out clusters. For this comparison, we apply our time-energy models on input sizes in the order of terabytes and clusters with more than 100 nodes.

5.6.1 Homogeneous versus Heterogeneous

First, we provide an answer to the research question: is a heterogeneous cluster consisting of nodes equipped with GPUs more energy-efficient than a homogeneous cluster with CPU-only nodes? Let the homogeneous cluster run Hadoop, while the heterogeneous cluster runs MoSS to exploit GPU acceleration. Given a workload mix of compute- and data-intensive applications that run alternatively and not at the same time, let α denote the ratio of compute-intensive workload, ranging from zero to one. Specifically, this workload mix runs for times of T and T' on the homogeneous and heterogeneous clusters, respectively. On the homogeneous cluster, compute- and data-intensive workloads run for total times T_c and T_d, respectively, such that

$$T = T_c + T_d = \alpha \cdot T + (1 - \alpha) \cdot T \tag{5.34}$$

where

$$\alpha = \frac{T_c}{T} \tag{5.35}$$

Based on our measurement analysis presented in Chapter 4, data-intensive applications achieve the same time performance on homogeneous and heterogeneous clusters with GPU, such that $T_d = T'_d$. In contrast, compute-intensive applications achieve a speedup S due to GPU acceleration. Thus, workload mix execution time on the heterogeneous cluster becomes

$$T' = \alpha \cdot \frac{T}{S} + (1 - \alpha) \cdot T \tag{5.36}$$

We define the time saving achieved by the heterogeneous cluster as

$$\frac{T - T'}{T} = \alpha - \frac{\alpha}{S} \tag{5.37}$$

For energy, we have to consider different average power usage, P_c and P_c' for homogeneous nodes, P_d and P_d' for heterogeneous nodes running compute- and data-intensive applications, respectively. Since energy is the product of execution time and average power usage, for the homogeneous cluster we have

$$E = \alpha \cdot T \cdot P_c + (1 - \alpha) \cdot T \cdot P_d \tag{5.38}$$

while for the heterogeneous cluster the energy usage is

$$E' = \alpha \cdot \frac{T}{S} \cdot P_c' + (1 - \alpha) \cdot T \cdot P_d' \tag{5.39}$$

Hence, energy saving is defined as

$$\frac{E - E'}{E} = 1 - \frac{E'}{E} = 1 - \frac{\frac{\alpha}{S} \cdot P_c' + (1 - \alpha) \cdot P_d'}{\alpha \cdot P_c + (1 - \alpha) \cdot P_d} \tag{5.40}$$

For this analysis, we select BS and GR as representative for compute- and data-intensive applications, respectively, and clusters of 100 nodes for both brawny and wimpy systems. Using model equations, we derive the execution time and energy savings as function of compute-intensive workload ratio, α. These savings are derived as the difference between the time or energy values on homogeneous and heterogeneous systems, reported to the values on homogeneous systems. Thus, positive values represent time and energy savings, while negative values show that heterogeneous systems are less efficient than homogeneous systems.

Time and energy savings for both brawny and wimpy systems are plotted in

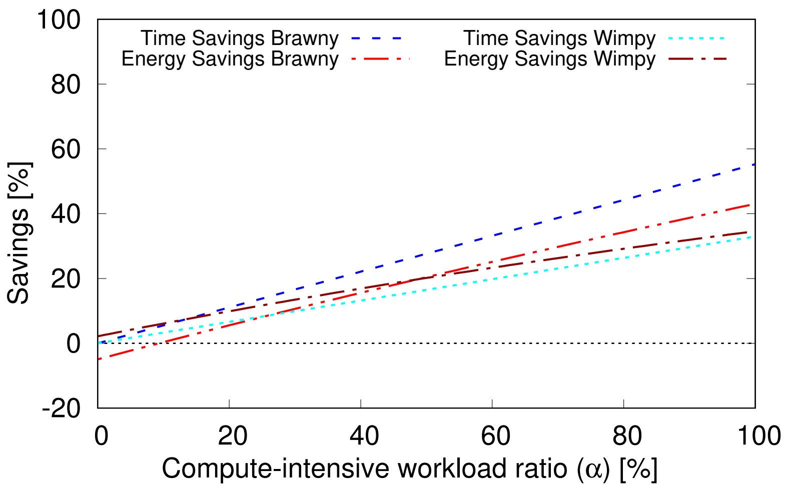
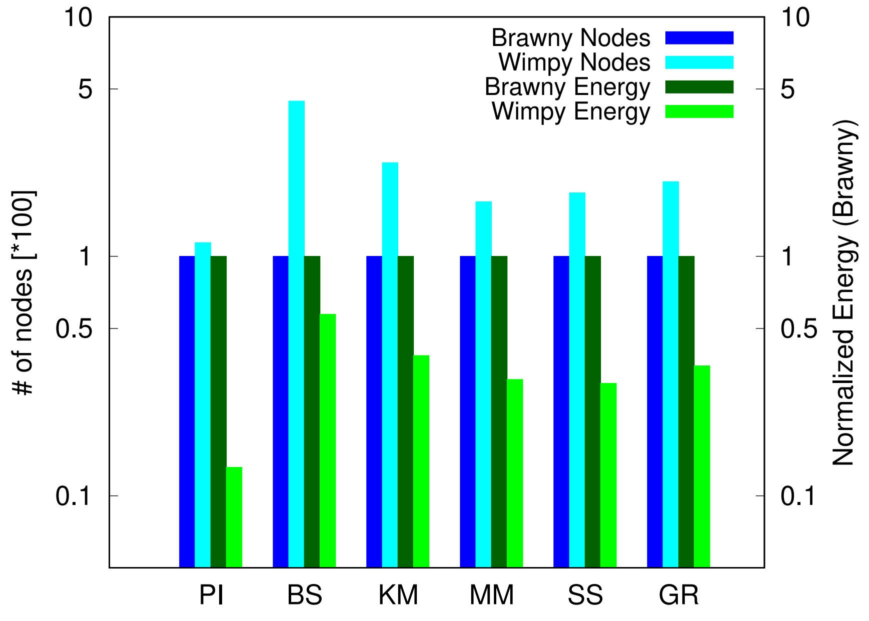


Figure 5.6: Time and energy savings on heterogeneous clusters

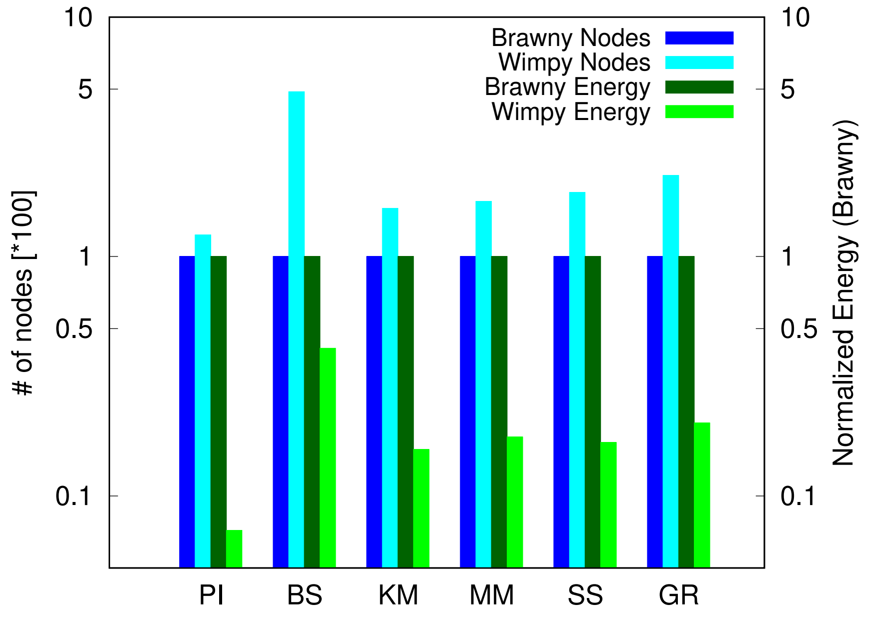
Figure 5.6. Energy savings are up to 43% and 35% on brawny and wimpy heterogeneous clusters, respectively, when running only compute-intensive workloads. These results are similar to the measurements presented in Section 4.4 where MoSS running BS on Jetson TK1 clusters achieves energy savings between 20% and 40%. For a balanced mix of 50% compute-intensive and 50% data-intensive workload, the energy savings for both brawny and wimpy clusters have the same value, 20%. The only area with energy loss is for the brawny cluster when the compute-intensive workload represents less than 9.3% of the total load. This energy loss is due to the higher idle power of the brawny system with GPU while this accelerator is not used. In contrast, the GPU of wimpy Jetson TK1 is much more energy-efficient and, thus, it does not affect the idle energy. In summary, we advocate for heterogeneous clusters consisting of nodes with CPU and GPU for faster and energy-efficient data-parallel processing.

5.6.2 Brawny versus Wimpy

In this last section, we provide an answer to the research question: are wimpy nodes more energy-efficient for data-parallel processing compared to traditional



(a) Nodes with HDD



(b) Nodes with SSD

Figure 5.7: Model-based brawny-wimpy equivalence

brawny nodes? To answer this question, we are using the time-energy models and the observation that an equivalence ratio can be established between a single brawny node and multiple wimpy nodes such that they achieve the same execution time, as shown by our measurement-driven analysis in Section 4.5.3. We are using the i7 and Jetson TK1 with HDD and SSD as representatives for the brawny and wimpy systems, respectively, and a cluster with 100 brawny nodes as baseline. Using the models, we determine how many wimpy nodes are needed in the wimpy cluster to achieve the same execution time as the brawny cluster for each application. We then compare the energy usage of the two types of clusters. The results for Jetson nodes with both HDD and SSD are plotted in Figure 5.7 using log scale to highlight energy savings. The plot shows that the equivalence ratio between brawny and wimpy nodes ranges between 1:1 for PI and 1:5 for BS, while wimpy nodes save up to 93% of the energy for PI on Jetson TK1 with SSD.

These model-based results for clusters with hundreds of nodes are similar but more optimistic compared to the single-node measurements presented in Section 4.5.3. For BS, measured values lead to an equivalence ratio of 1:6 while model-based values lead to equivalence ratio of 100:445, or 1:5 using ceiling, and 100:488, or 1:5, for nodes with HDD and SSD, respectively. For KM, measured values lead to an equivalence ratio of 1:3, while model-based results expose equivalence ratios of 1:3 and 1:2 for nodes with HDD and SSD, respectively. Even if we consider installation space and cost of acquisition, it is still possible to replace one brawny node with up to five or six wimpy nodes to achieve energy-efficient data-parallel processing. Hence, our analysis advocates for heterogeneous wimpy nodes to achieve efficient data-parallel processing.

5.7 Summary

In this thesis chapter, we have introduced our measurement-driven analytic models to determine the execution time and energy usage of data-parallel MapReduce execution using Hadoop on homogeneous systems with CPUs and using MoSS on heterogeneous systems with CPUs and GPUs. To the best of our knowledge, we are the first to design an energy usage model for MapReduce. Our models use baseline runs to measure key parameters in order to increase the accuracy. For example, the energy model uses measured values of average power during Map, Shuffle and Reduce phases of MapReduce execution. We have validated our models on up to 264 configurations consisting of six applications, two frameworks represented by Hadoop and MoSS, two input sizes, three cluster sizes and three types of systems covering both brawny and wimpy nodes. This validation shows an average model error below 15% for both time and energy.

Using our models, we have first proved in a formal way that lazy processing is faster than chunking and we have analyzed the cases where overlapping is less suitable compared to selecting the fastest execution unit between the CPU and GPU. This formal analysis strengthens the description of our techniques for efficient data-parallel processing on heterogeneous systems.

Secondly, we have analyzed real and hypothetical system configurations to improve the time-energy performance of data-parallel processing. We show that replacing the traditional HDD with the faster SSD does not significantly affect the execution time but can save up to 33% of the energy.

Thirdly, we have compared heterogeneous with homogeneous systems and wimpy with brawny systems when processing large inputs on clusters with hundreds of nodes. In line with our measurement-driven analysis, we show that heterogeneous systems always save time and energy when the workload mix contains at least 10% of compute-intensive data-parallel workload. Next, we show that multiple wimpy nodes achieve the same time performance as one brawny node while saving up to 93% of the energy. Hence, we advocate for heterogeneous wimpy nodes to achieve efficient data-parallel processing.

Chapter 6

Conclusions

We conclude by summarizing our current work and briefly discussing future research directions.

6.1 Summary

In the last few years, we have witnessed the explosion of Big Data, the advent of data-parallel processing, and the proliferation of heterogeneous systems that combine multiple processing units with different performance-to-power ratio (PPR). In this context, our objective is to efficiently execute batch data-parallel applications on heterogeneous systems, with a focus on intra-node and intra-chip heterogeneous systems with GPU. To achieve this objective, we propose an approach consisting of three parts, (i) techniques for efficient execution of data-parallel applications on intra-node and intra-chip heterogeneous systems with accelerators, such as GPUs [70, 71], (ii) measurement-driven [71, 72] and (iii) model-driven time-energy performance analysis of data-parallel applications on heterogeneous systems. We briefly summarize the contributions of this thesis, as previously depicted in the context of related work in Figure 2.1.

Techniques for Efficient Data-parallel Processing on Heterogeneous Systems with GPU

We have firstly presented our techniques for efficient data-parallel processing on intra-node and intra-chip heterogeneous systems with GPU and their implementation under MoSS, a Hadoop-CUDA framework that we have developed. MoSS allows developers to easily modify existing MapReduce applications by providing an expresive GPU API and by preserving application's logic structure. Among the presented techniques, *lazy processing* enables the processing of multiple input records at a time on a GPU, in contrast with *chunking* which divides a single record among GPU threads. Compared to chunking [84], our lazy processing is on average 54% faster. To increase the efficiency of data-parallel processing, we propose *dynamic mapping* techniques. Counter to intuition, we show that a *one-time* profiling approach that selects the best processing unit achieves better performance compared to overlapping the execution on both the CPU and GPU.

Measurement-driven Time-Energy Performance Analysis

We have evaluated our techniques using six representative workloads and two cluster systems with diverse performance capabilities. First, we evaluate the time-energy performance on our in-house low-power wimpy cluster based on Nvidia Jetson TK1 nodes integrating quad-core ARM Cortex-A15 CPUs and 192-core Nvidia Kepler GPUs on the same chip. Second, we evaluate the execution time performance on a high-performance brawny cluster based on Amazon EC2 instances equipped with discrete GPUs with 1536 cores. Compared to Hadoop [14], MoSS improves the execution time of compute-intensive workloads by factors of up to 3.1 and 2.3 on wimpy and brawny nodes clusters, respectively. Along with the improvement in execution time, MoSS saves up to 80% of the energy used by the wimpy cluster. Moreover, the execution time of MoSS is almost always within

5% of the best Hadoop CPU-only execution time for data-intensive workloads.

For our in-depth performance analysis of heterogeneous systems with GPU, we have selected three configurations representing both *intra-node* and *intra-chip* heterogeneity, (i) an Intel i7 system hosting a discrete 640-core Nvidia GPU of Maxwell generation, representing *intra-node* heterogeneous brawny systems, (ii) a quad-core ARM Cortex-A9 with the same Maxwell GPU representing *intra-node* heterogeneous wimpy systems, and (iii) a quad-core ARM Cortex-A15 integrated with 192 Nvidia Kepler GPU cores representing *intra-chip* heterogeneous wimpy systems. We evaluate the time and energy performance of these systems using three MapReduce applications with diverse resource demands.

We have shown that for compute-intensive workloads such as BlackScholes, the brawny heterogeneous system achieves speedups of up to 2.3 and reduces the energy usage by almost half compared to the brawny homogeneous system. As expected, for applications such as Grep where data transfers dominate the execution time, heterogeneous systems exhibit worse time-energy performance compared to homogeneous systems. For example, the heterogeneous wimpy Kayla with discrete GPU consumes 14 times the energy of the homogeneous Kayla system due to very low host-device transfer bandwidth and high power overhead of the discrete GPU. Moreover, the lower performance of wimpy systems on data analytics is in part due to the small main memory size. While brawny systems have large memories to accommodate Hadoop's intermediate data, our profiling shows that on the wimpy systems data is spilled to disk leading to 80% more storage transfers compared to the brawny system. Among heterogeneous systems, the wimpy with discrete GPU exhibits the worst time-energy performance. But the wimpy with integrated GPU uses the lowest energy across all workloads due to more energy-efficient CPU and GPU, and better balanced system resources. To account for the execution time difference, we establish an equivalence ratio between a single brawny heterogeneous node and multiple wimpy heterogeneous nodes. Based on this equivalence, the wimpy nodes not only achieve similar execution times compared to a single brawny node, but also exhibit energy savings of up to two-thirds. This result advocates the potential usage of wimpy systems with integrated GPUs for Big Data analytics.

Model-driven Time-Energy Performance Analysis

Based on our extensive performance measurements, we have developed analytic models to determine the execution time and energy usage of data-parallel MapReduce execution using Hadoop on homogeneous systems with CPUs and using MoSS on heterogeneous systems with CPUs and GPUs. To the best of our knowledge, we are the first to design an energy usage model for MapReduce. Our models use baseline runs to measure key parameters in order to increase the accuracy. For example, the energy model uses measured values of average power during Map, Shuffle and Reduce phases of MapReduce execution. We have validated our models on up to 264 configurations consisting of six applications, two frameworks represented by Hadoop and MoSS, two input sizes, three cluster sizes and three types of systems covering both brawny and wimpy nodes. This validation shows an average model error less than 15% for both time and energy.

We have compared heterogeneous with homogeneous systems and wimpy with brawny systems when processing large inputs on clusters with hundreds of nodes. In line with our measurement-driven analysis, we show that heterogeneous systems always save time and energy when the workload mix contains at least 10% of compute-intensive data-parallel workload. Next, we show that multiple wimpy nodes achieve the same time performance as one brawny node while saving up to 90% of the energy. Hence, we advocate for the use of heterogeneous wimpy nodes to achieve efficient data-parallel processing.

6.2 Future Work

In this section, we acknowledge the limitations of this thesis and discuss future research directions. From programming perspective, our approach is limited by the usage of MapReduce which exposes only two types of operators, namely Map and Reduce. In this work, we have focused only on Map operator since (i) we target massive data-parallel execution suitable for GPUs and (ii) we want to keep existing application structure when writing GPU code in MoSS. Nonetheless, it would be useful to design and analyze an approach for efficient Reduce execution on heterogeneous systems with GPU. From systems perspective, we have focused on Nvidia CUDA GPUs, but it is worthy to apply and analyze our techniques on wimpy systems with wimpy GPUs such as PowerVR or Mali GPUs supporting OpenCL programming model. Lastly, the accuracy of our time-energy models should be improved by using in-depth measurements exposing hardware performance metrics and advanced mathematical modeling tools.

With the announcement of Google Cloud Dataflow [5, 45] and its open source implementation under Apache Beam [11,12], data-parallel processing enters a new era of unified batch, micro-batch and stream processing. But Cloud Dataflow and its runtime engines are still designed for homogeneous brawny clusters. Hence, there is need for techniques and performance analysis of Cloud Dataflow execution on heterogeneous wimpy clusters. We anticipate that our current approach will further improve the time-energy performance of Cloud Dataflow applications on heterogeneous systems. For example, *ParDo* operator in Cloud Dataflow which performs fine-grain record transformation similar to *Map* operator in MapReduce could use *lazy processing* to exploit the GPU in order to save energy. On the other hand, Cloud Dataflow exposes more data operators compared to MapReduce. These data operators with different runtime resource demands could use suitable heterogeneous processing units to achieve better time-energy performance. For example, on the latest heterogeneous systems with many-core CPUs and GPUs, such as Nvidia's Jetson TX2 [10], the GPU could handle the *ParDo* operators, the big CPU cores could handle *Join* and *Flatten* operators, while the little cores could handle I/O operations.

While cloud has dominated computing landscape of the last decade, new applications requiring very low latency, high bandwidth and robust networking are pushing computing to the edge of the network. Hence, edge and fog computing [3] represent the next revolution for distributed, global-scale data processing. Using state-of-the-art heterogeneous systems, techniques for data-parallel processing on heterogeneous processing units and measurement-driven performance models, we plan to advance the research on edge and fog computing and achieve the next level of time-energy efficiency for massive-scale data-parallel processing.

References

[1] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, T. N. Vijaykumar, Tarazu: Optimizing MapReduce on Heterogeneous Clusters, *Proc. of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–74, 2012.

[3] Y. Ai, M. Peng, K. Zhang, Edge Cloud Computing Technologies for Internet of Things: A Primer, *Digital Communications and Networks*, Accepted Manuscript, 2017.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, MillWheel: Fault-tolerant Stream Processing at Internet Scale, *Proc. of VLDB Endowment*, 6(11):1033–1044, Aug. 2013.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing, *Proc. of VLDB Endowment*, 8(12):1792–1803, 2015.

[6] Amazon, Amazon EMR, http://www.webcitation.org/6jvOzd7Vh, 2016.

[7] AMD, What is Heterogeneous System Architecture (HSA)?, http://www.webcitation.org/6PgDYqFrY, 2012.

[8] AMD, AMD to Accelerate the ARM Server Ecosystem with the First ARM-based CPU and Development Platform from a Server Processor Vendor, http://www.webcitation.org/6PgFAdEFp, 2014.

[9] AMD, What is Heterogeneous Computing?, http://www.webcitation.org/6Pg0hzrCj, 2014.

[10] AnandTech, Nvidia Announces Jetson TX2: Parker Comes To Nvidias Embedded System Kit, http://www.webcitation.org/6qhLYHVCd, 2017.

[11] Apache, Beam Proposal, http://tinyurl.com/jkbpg6k, 2016.

[12] Apache, Apache Beam: An Advanced Unified Programming Model, https://tinyurl.com/yb75o5jo, 2017.

[13] Apache, Flink, https://tinyurl.com/zocntyw, 2017.

[14] Apache, Hadoop, http://tinyurl.com/5f4ojf, 2017.

[15] Apache, Spark, https://tinyurl.com/nm9wjs9, 2017.

[16] ARM, ARM Announces Support For EEMBC CoreMark Benchmark, http://www.webcitation.org/6RPwNECop, 2009.

[17] ARM, Dhrystone and MIPs performance of ARM processors, http://www.webcitation.org/6RPwC2TUb, 2010.

[18] ARM, *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*, ARM, 2012.

[19] ARM, big.LITTLE Processing, http://www.webcitation.org/6Phid5IWk, 2014.

[20] K. Arvind, R. S. Nikhil, Executing a Program on the MIT Tagged-token Dataflow Architecture, *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[21] J. Auerbach, D. F. Bacon, P. Cheng, R. Rabbah, Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures, *Proc. of 2010 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 89–108, 2010.

[22] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC Benchmark Suite: Characterization and Architectural Implications, *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[23] T. Bingmann, Parallel Memory Bandwidth Benchmark / Measurement, 2013.

[24] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, N. Weizenbaum, FlumeJava: Easy, Efficient Data-parallel Pipelines, *Proc. of 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375, 2010.

[25] K. Chen, J. Powers, S. Guo, F. Tian, CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds, *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1403–1412, 2014.

[26] L. Chen, G. Agrawal, Optimizing MapReduce for GPUs with Effective Shared Memory Usage, *Proc. of 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 199–210, 2012.

[27] L. Chen, X. Huo, G. Agrawal, Accelerating MapReduce on a Coupled CPU-GPU Architecture, *Proc. of 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 25:1–25:11, 2012.

[28] Y. Chen, S. Alspaugh, D. Borthakur, R. Katz, Energy Efficiency for Large-scale MapReduce Workloads with Significant Interactive Analysis, *Proc. of 7th ACM European Conference on Computer Systems*, pages 43–56, 2012.

[29] Y. Chen, S. Alspaugh, R. Katz, Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads, *VLDB Endowment*, 5(12):1802–1813, Aug. 2012.

[30] A. Cunningham, From Smartphone to Server Room: Nvidia's "Kayla" Shows the Future of Tegra, http://www.webcitation.org/6VcpwYsD5, 2013.

[31] M. Curtin, Write Once, Run Anywhere: Why It Matters, http://www.interhack.net/people/cmcurtin/rants/write-once-run-anywhere, 1998.

[32] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, The Scalable Heterogeneous Computing (SHOC) Benchmark Suite, *Proc. of the 3rd Workshop on General-Purpose*

Computation on Graphics Processing Units, pages 63–74, New York, NY, USA, 2010.

[33] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Proc. of 6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[34] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, S. J. Fink, Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers), *Proc. of 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2012.

[35] I. El-Helw, R. Hofman, H. E. Bal, Scaling MapReduce Vertically and Horizontally, *Proc. of 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 525–535, 2014.

[36] M. Elteir, H. Lin, W. c. Feng, T. Scogland, StreamMR: An Optimized MapReduce Framework for AMD GPUs, *Proc. of 17th IEEE International Conference on Parallel and Distributed Systems*, pages 364–371, 2011.

[37] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark Silicon and the End of Multicore Scaling, *Proc. of 38th Annual International Symposium on Computer Architecture*, pages 365–376, 2011.

[38] R. Farivar, A. Verma, E. Chan, R. Campbell, MITHRA: Multiple Data Independent Tasks on a Heterogeneous Resource Architecture, *Proc. of 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[39] E. Feller, L. Ramakrishnan, C. Morin, On the Performance and Energy Efficiency of Hadoop Deployment Models, *Proc. of 2013 IEEE International Conference on Big Data*, pages 131–136, 2013.

[40] M. J. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[41] D. Foley, NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data, http://www.webcitation.org/6XL0t4iVi, 2014.

[42] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

[43] GNU, GCC ARM Options, http://www.webcitation.org/6VazB3S7x, 2015.

[44] I. n. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, R. Bianchini, GreenHadoop: Leveraging Green Energy in Data-processing Frameworks, *Proc. of 7th ACM European Conference on Computer Systems*, pages 57–70, 2012.

[45] Google, Sneak Peek: Google Cloud Dataflow, a Cloud-native Data Processing Service, http://www.webcitation.org/6RELNVEr8, 2014.

[46] Google, Overview of Running a MapReduce Job, http://www.webcitation.org/6jvPnxjWV, 2016.

[47] D. Grewe, Z. Wang, M. F. P. O'Boyle, OpenCL Task Partitioning in the Presence of GPU Contention, 2014.

[48] M. Grossman, M. Breternitz, V. Sarkar, HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL, *Proc. of 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1918–1927, 2013.

[49] V. Gupta, K. Schwan, Brawny vs. Wimpy: Evaluation and Analysis of Modern Workloads on Heterogeneous Processors, *Proc. of 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 74–83, 2013.

[50] M. Harris, Unified Memory in CUDA 6, http://www.webcitation.org/6WrmVnwyE, 2013.

[51] M. Harris, 5 Things You Should Know About the New Maxwell GPU Architecture, http://www.webcitation.org/6VcZH7xTv, 2014.

[52] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, Mars: A MapReduce Framework on Graphics Processors, *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[53] W. He, H. Cui, B. Lu, J. Zhao, S. Li, G. Ruan, J. Xue, X. Feng, W. Yang, Y. Yan, Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters, *Proc. of 29th ACM on Intl. Conference on Supercomputing*, pages 143–153, 2015.

[54] H. Herodotou, F. Dong, S. Babu, No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics, *Proc. of 2nd ACM Symposium on Cloud Computing*, pages 18:1–18:14, 2011.

[55] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu, Starfish: A Self-tuning System for Big Data Analytics, *Proc. of 5th Biennial Conference on Innovative Data Systems Research*, pages 261–272, 2011.

[56] U. Hoelzle, L. A. Barroso, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan and Claypool Publishers, 1st edition, 2009.

[57] C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, MapCG: Writing Parallel Program Portable Between CPU and GPU, *Proc. of 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010.

[58] IBM, Datasheet - Hadoop-as-a-service, http://tinyurl.com/zb3gt6o, 2016.

[59] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed Data-parallel Programs from Sequential Building Blocks, *Proc. of 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[60] F. Ji, X. Ma, Using Shared Memory to Accelerate MapReduce on Graphics Processing Units, *Proc. of 25th IEEE International Parallel Distributed Processing Symposium*, pages 805–816, 2011.

[61] D. Jiang, B. C. Ooi, L. Shi, S. Wu, The Performance of MapReduce: An In-depth Study, *Proc. of VLDB Endowment*, 3(1-2):472–483, 2010.

[62] S. Kadirvel, J. A. B. Fortes, Grey-Box Approach for Performance Prediction in Map-Reduce Based Platforms, *Proc. of 21st International Conference on Computer Communications and Networks*, pages 1–9, 2012.

[63] S. Kavulya, J. Tan, R. Gandhi, P. Narasimhan, An Analysis of Traces from a Production MapReduce Cluster, *Proc. of 10th IEEE/ACM Intl. Conference on Cluster, Cloud and Grid Computing*, pages 94–103, 2010.

[64] M. Khan, Y. Jin, M. Li, Y. Xiang, C. Jiang, Hadoop Performance Modeling for Job Estimation and Resource Provisioning, *IEEE Transactions on Parallel and Distributed Systems*, 27(2):441–454, 2016.

[65] W. Lang, J. M. Patel, Energy Management for MapReduce Clusters, *Proc. of VLDB Endowment*, 3(1-2):129–139, Sept. 2010.

[66] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, B. Moon, Parallel Data Processing with MapReduce: A Survey, *SIGMOD Rec.*, 40(4):11–20, Jan. 2012.

[67] J. Leverich, C. Kozyrakis, On the Energy (in)Efficiency of Hadoop Clusters, *SIGOPS Operating Systems Review*, 44(1):61–65, Mar. 2010.

[68] M. D. Linderman, J. D. Collins, H. Wang, T. H. Meng, Merge: A Programming Model for Heterogeneous Multi-core Systems, *Proc. of 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296, 2008.

[69] H. Liu, A Measurement Study of Server Utilization in Public Clouds, *Proc. of 9th IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 435–442, 2011.

[70] D. Loghin, L. Ramapantulu, , Y. M. Teo, Efficient Time-Energy Execution of MapReduce on Heterogeneous Systems with GPU, Technical report, NUS, 2017.

[71] D. Loghin, L. Ramapantulu, O. Barbu, Y. M. Teo, A TimeEnergy Performance Analysis of MapReduce on Heterogeneous Systems with GPUs, *Performance Evaluation*, 91:255–269, 2015.

[72] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, Y. M. Teo, A Performance Study of Big Data on Small Nodes, *Proc. of VLDB Endowment*, 8(7):762–773, 2014.

[73] G. Marsaglia, Xorshift RNGs, *Journal of Statistical Software*, 8(1):1–6, 2003.

[74] Microsoft, HDInsight - A managed Apache Hadoop, Spark, R, HBase, and Storm cloud service made easy, http://www.webcitation.org/6jvPaAOIf, 2016.

[75] Nvidia, Nvidia Unveils First Mobile Supercomputer for Embedded Systems, http://www.webcitation.org/6VdkUISQn, 2014.

[76] Nvidia, CUDA, https://tinyurl.com/ye65wt2, 2017.

[77] K. Parrish, NVIDIA GPUs Can Outperform Google Brain, http://tinyurl.com/qykle22, 2014.

[78] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, A. Ramirez, The Low Power Architecture Approach Towards Exascale Computing, *Journal of Computational Science*, 4(6):439–443, 2013.

[79] L. Ramapantulu, D. Loghin, Y. M. Teo, An Approach for Energy Efficient Execution of Hybrid Parallel Programs, *Proc. of 29th IEEE International Parallel and Distributed Processing Symposium*, pages 1000–1009, 2015.

[80] L. Ramapantulu, D. Loghin, Y. M. Teo, On Energy Proportionality and Time-Energy Performance of Heterogeneous Clusters, *Proc. of 18th IEEE International Conference on Cluster Computing*, pages 221–230, Sept 2016.

[81] L. Ramapantulu, B. M. Tudor, D. Loghin, T. Vu, Y. M. Teo, Modeling the Energy Efficiency of Heterogeneous Clusters, *Proc. of 43rd Intl. Conference on Parallel Processing*, pages 321–330, 2014.

[82] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, *Proc. of 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[83] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, D. Fetterly, Dandelion: A Compiler and Runtime for Heterogeneous Systems, *Proc. of 24th ACM Symposium on Operating Systems Principles*, pages 49–68, 2013.

[84] K. Shirahata, H. Sato, S. Matsuoka, Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters, *Proc. of 2nd International Conference on Cloud Computing Technology and Science*, pages 733–740, 2010.

[85] A. Solana, Using ARM chips and Linux, Barcelona center dreams of being 'Airbus of supercomputing', http://www.webcitation.org/6t35IuUr5, 2015.

[86] J. A. Stuart, J. D. Owens, Multi-GPU MapReduce on GPU Clusters, *Proc. of 25th IEEE International Parallel and Distributed Processing Symposium*, pages 1068–1079, 2011.

[87] H. Sutter, J. Larus, Software and the Concurrency Revolution, *ACM Queue*, 3(7):54–62, 2005.

[88] F. Tian, K. Chen, Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds, *Proc. of 4th International Conference on Cloud Computing*, pages 155–162, 2011.

[89] TOP500, Piz Daint, http://www.webcitation.org/6PgPSsLA8, 2012.

[90] Top500.org, Highlights - June 2017, http://www.webcitation.org/6ssYM95Tu, 2017.

[91] B. M. Tudor, Y. M. Teo, On Understanding the Energy Consumption of ARM-based Multicore Servers, *Proc. of SIGMETRICS '13*, pages 267–278, 2013.

[92] L. van Doorn, Enabling Cloud Workloads Through Innovations in Silicon, http://www.webcitation.org/6t33R0NZg, 2017.

[93] A. Verma, L. Cherkasova, R. H. Campbell, ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments, *Proc. of 8th ACM International Conference on Autonomic Computing*, pages 235–244, 2011.

[94] A. Verma, L. Cherkasova, R. H. Campbell, Resource Provisioning Framework for MapReduce Jobs with Performance Goals, *Proc. of 12th International Middleware Conference*, pages 160–179, 2011.

[95] A. Verma, L. Cherkasovab, R. H. Campbellc, Profiling and Evaluating Hardware Choices for MapReduce Environments: An Application-aware Approach, *Performance Evaluation*, 79:328–344, 2014.

[96] R. P. Weicker, Dhrystone: A Synthetic Systems Programming Benchmark, *Communications of ACM*, 27(10):1013–1030, 1984.

[97] Wikipedia, Electricity Pricing, http://www.webcitation.org/6R9bgVRLG, 2013.

[98] D. Wong, M. Annavaram, KnightShift: Scaling the Energy Proportionality Wall Through Server-Level Heterogeneity, *Proc. of 45th International Symposium on Microarchitecture*, pages 119–130, 2012.

[99] Y. Yan, M. Grossman, V. Sarkar, JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, *Proc. of 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, 2009.

[100] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *Proc. of 9th USENIX Conference on Networked Systems Design and Implementation*, pages 2–2, 2012.

[101] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, *Proc. of 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[102] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized Streams: Fault-tolerant Streaming Computation at Scale, *Proc. of 24th ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.

[103] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica, Improving MapReduce Performance in Heterogeneous Environments, *Proc. of 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

[104] Z. Zhang, L. Cherkasova, B. T. Loo, Benchmarking Approach for Designing a Mapreduce Performance Model, *Proc. of 4th ACM/SPEC International Conference on Performance Engineering*, pages 253–258, 2013.

[105] Z. Zhang, L. Cherkasova, B. T. Loo, Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments, *Proc. of 6th International Conference on Cloud Computing*, pages 839–846, 2013.

Appendix A

Heterogeneous Systems Characterization

Throughout this thesis, we use three brawny server systems and three wimpy systems representing both homogeneous and heterogeneous systems, as depicted in Figure A.1. These systems are used as homogeneous systems by enabling a single processing unit, typically the CPU. When another processing unit, such as a GPU or a different type of CPU is added, the systems become heterogeneous. In

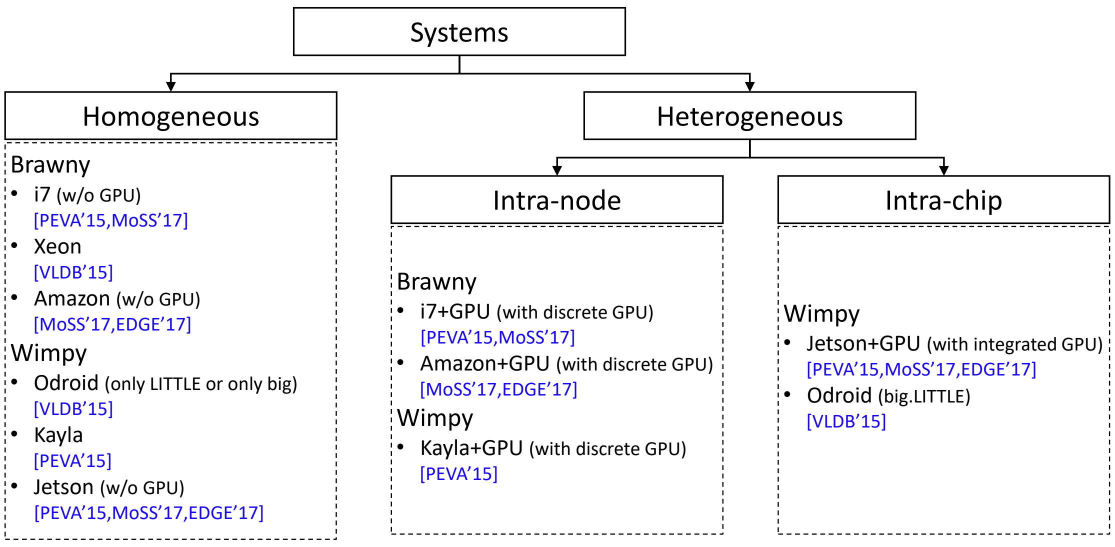


Figure A.1: Homogeneous and heterogeneous systems

this section, we describe these systems and perform an in-depth characterization at CPU, GPU, memory, storage and networking level.

A.1 Specifications

A.1.1 Brawny Systems

Brawny Homogeneous System with Intel Xeon (Xeon). As server-class homogeneous brawny systems, we use Supermicro 813M 1U based on two Intel Xeon E5-2603 CPUs with four cores each. This system has 8 GB DDR3 memory, 1 TB hard disk and 1 Gbit Ethernet network card. For a fair comparison with 4-core systems, we remove one of the two Xeon CPUs. The idle power of the 4-core Xeon node is 35 W, and its peak power is around 55 W, making it a lower power brawny system.

This Xeon server system runs Ubuntu 13.04 with Linux kernel 3.8.0 for x64 architecture. The C/C++ compiler available on this system is *gcc* 4.7.3, while Java code runs on Oracle's Java Virtual Machine (JVM) version 1.7.0_45.

Brawny Intra-node Heterogeneous System with Discrete GPU (i7 or i7+GPU). As traditional brawny intra-node heterogeneous system, we use a server system based on a quad-core Intel Core i7 processor hosting a discrete Nvidia GPU on the PCI Express (PCIe) slot. This system has 16 GB of RAM and a 512 GB solid-state drive (SSD) to store all datasets and workloads. We employ the Nvidia Maxwell architecture by hosting a GTX 750 Ti video card consisting of 640 cores with CUDA compute capability 5.0, and 2 GB of GDDR5 memory. Compared to previous Kepler architecture, Maxwell is two times more energy-efficient [51], being the suitable choice for achieving energy efficiency. For our comparison between heterogeneous and homogeneous systems, we add the

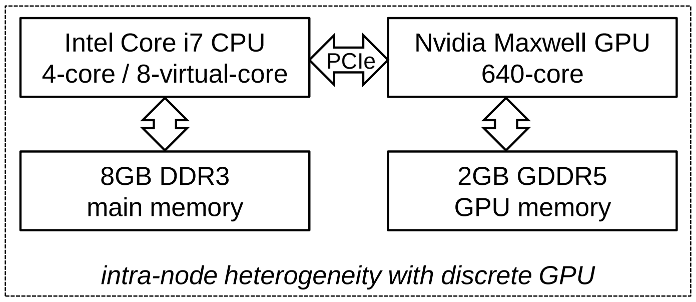


Figure A.2: Brawny system with Intel CPU and discrete Nvidia GPU

GPU in the **i7+GPU** system, as depicted in Figure A.2, and remove it in a homogeneous **i7**-only system.

This brawny system runs Ubuntu 13.04 with Linux kernel 3.11.0 installed on a separate SSD. We use *gcc* 4.8.1 as C/C++ compiler and *nvcc* from CUDA toolkit 6.5 as CUDA compiler. Java code runs on Oracle's Java Virtual Machine (JVM) version 1.8.0_25.

Brawny Intra-node Heterogeneous Cloud System (Amazon or Amazon+GPU). With the advances in cloud computing and with larger system configuration space offered by the cloud, we use cloud-based intra-node heterogeneous nodes with discrete GPUs. We employ up to 12 Amazon EC2 *g2.2xlarge* instances launched in the same region. These instances are equipped with Intel Xeon E5-2670 CPU and Nvidia GRID K520 GPU. This GPU of Kepler architecture has 1536 cores and 4 GB of memory. Each instance is configured with eight virtual CPU cores, 15 GB of RAM and 500 GB of SSD-based storage space.

From the software perspective, the cloud instances run Ubuntu OS with Linux kernel 3.13.0. We compile native code with *gcc* 4.8.2 and *nvcc* from CUDA toolkit 6.5, while Java code runs on Oracle's jdk1.8.0_25.

A.1.2 Wimpy Systems

Wimpy Intra-chip Heterogeneous System based on ARM big.LITTLE (Odroid). We use Odroid XU as many-core intra-node heterogeneous wimpy system, as depicted in Figure A.3. Odroid XU is a development board equipped with Samsung Exynos 5410 System on a Chip (SoC). This board is representative for high-end mobile phones. For example, Samsung Exynos 5410 is used in the international version of the Samsung Galaxy S4 phones. Other high end contemporary mobile devices employ SoCs with very similar performance characteristics, such as Qualcomm Snapdragon 80x and Nvidia Tegra 4. Specific to the Exynos 5410 SoC is that the CPU has two types of cores: ARM Cortex-A7 *little cores*, which consume a small amount of power and offer slow in-order execution, and ARM Cortex-A15 *big cores* which support faster out-of-order execution, but with a higher power consumption. This heterogeneous CPU architecture is termed *ARM big.LITTLE* [19]. The CPU has a total of eight cores, split in two groups[1] of cores: one group of four ARM Cortex-A7 little cores, and one group of four ARM Cortex-A15 big cores. Each core has a pair of dedicated L1 data and instruction caches, and each group of cores has an L2 unified cache.

Although the CPU has eight cores, Exynos 5410 allows either the four big cores, or the four little cores to be active at one moment. To save energy, when one group is active, the other one is powered down. Thus, a program cannot execute on both the big and the little cores at the same time. Instead, the operating system (OS) can alternate the execution between them. Switching between the two groups incurs a small performance price, as the L2 and L1 caches of the newly activated group must warm up.

[1]In the computer architecture literature, this group of cores is termed *cluster of cores*. However, due to potential confusion with cluster of nodes encountered in distributed computing, we shall use the term *group of cores*.

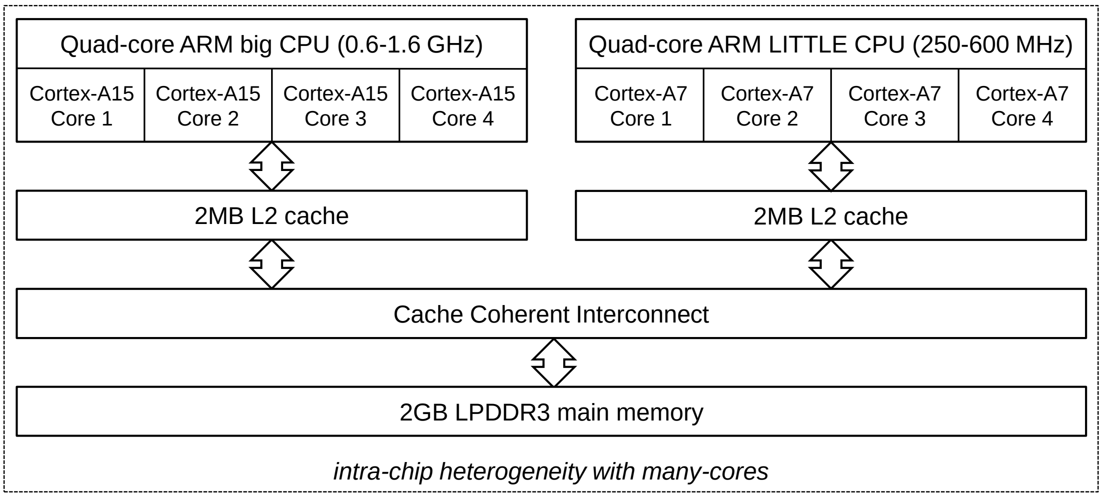


Figure A.3: Wimpy ARM big.LITTLE system

The core clock frequency of the little cores ranges from 250 to 600 MHz, and that of big cores ranges from 600 MHz to 1.60 GHz. Dynamic voltage and frequency scaling (DVFS) is employed to increase the core frequency in response to the increase in CPU utilization. On this ARM big.LITTLE architecture, the OS can be instructed to operate the cores in three configurations:

1. *little*: only use the ARM Cortex-A7 little cores, and their frequency is allowed to range from 250 to 600 MHz.

2. *big*: only use the ARM Cortex-A15 big cores, and their frequency is allowed to range from 600 to 1600 MHz.

3. *big.LITTLE*: when the OS is allowed to switch between the two types of cluster. The switching frequency is 600 MHz.

We have build a 4-node Odroid cluster, as shown in Figure A.4. Each Odroid XU node has 2 GB of low-power DDR3 memory, a 64 GB eMMC flash-storage and a 100 Mbit Ethernet card. However, to improve the network performance, we connect a Gbit Ethernet adapter on the USB 3.0 interface.

The ARM-based Odroid XU board runs Ubuntu 13.04 operating system with

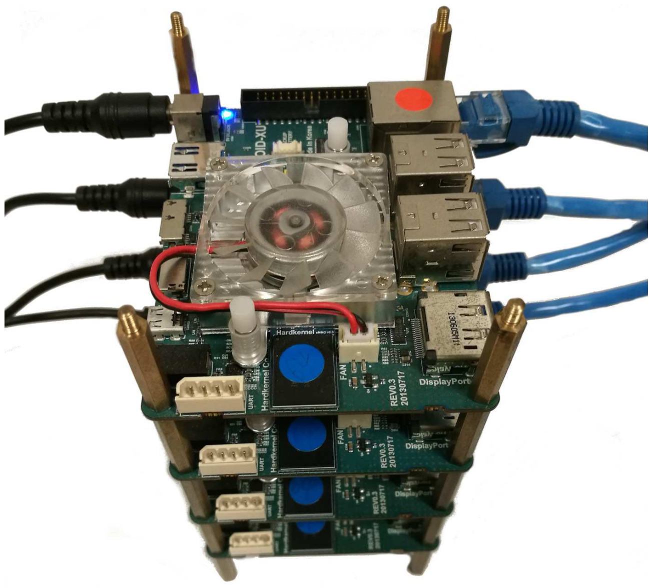


Figure A.4: Wimpy Odroid cluster

Linux kernel 3.4.67, which is the latest kernel version working on this platform. For compiling native C/C++ programs, we use *gcc* 4.7.3 arm-linux-gnueabihf.

Wimpy Intra-node Heterogeneous System with Discrete GPU (Kayla or Kayla+GPU). The *intra-node* heterogeneous wimpy systems with *discrete* GPUs are represented by a Kayla DevKit equipped with Nvidia Tegra 3 System-on-a-Chip (SoC) having four ARM Cortex-A9 cores and 2 GB of low-power DDR2. This system has a PCI Express x16 port that can accommodate a full-fledged discrete GPU. Moreover, it has a SATA interface which enables the connection of a high-capacity disk. We use a 512 GB SSD to store datasets and workloads. For the

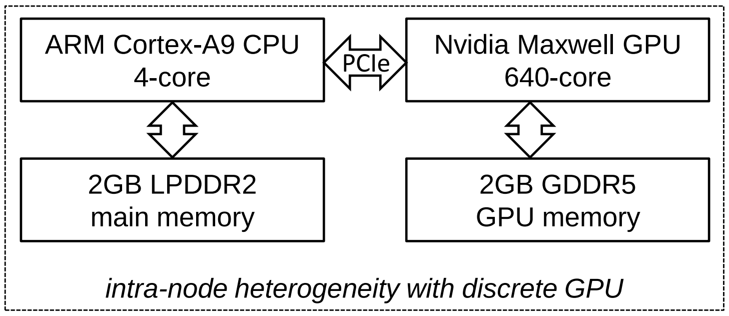


Figure A.5: Wimpy system with ARM CPU and discrete Nvidia GPU

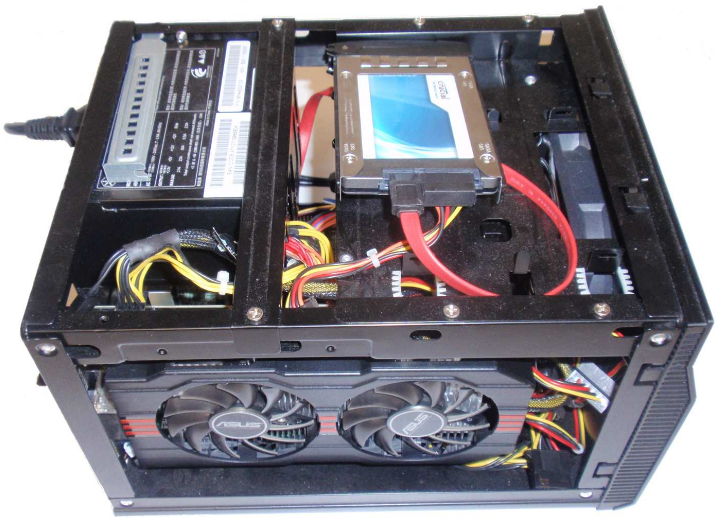


Figure A.6: Wimpy Kayla with GPU

comparison of heterogeneous and homogeneous systems, we add the GPU in the **Kayla+GPU** system, or remove it in a homogeneous **Kayla**-only system. The diagram of the heterogeneous system with discrete GPU is shown in Figure A.5 and the real system in shown Figure A.6.

By default, Ubuntu 12.04 with Linux kernel 3.1.10 is installed on system's flash storage. On top of this OS, we install CUDA toolkit 6.5 and necessary Nvidia drivers. To compile native code, we use *gcc* 4.6.3 and *nvcc* from CUDA toolkit. Java code runs on Oracle's Java Virtual Machine (JVM) version 1.8.0_06.

Wimpy Intra-chip Heterogeneous System with Integrated GPU (Jetson or Jetson+GPU). With the increasing adoption of *integrated* CPU-GPU systems [75], we use an *intra-chip* heterogeneous wimpy system represented by Jetson TK1 based on Nvidia Tegra K1 SoC which integrates four ARM Cortex-A15 CPU cores, 192 Nvidia Kepler GPU cores and a shared 2 GB low-power memory. This system's diagram is presented in Figure A.7, and a 6-node Jetson cluster is shown in Figure A.8. Beside the four fully-fledged Cortex-A15 cores, Tegra K1 incorporates a transparent low-power *companion* core which runs the OS at low system utilization. On this Jetson system, the OS is installed on a 16 GB eMMC.

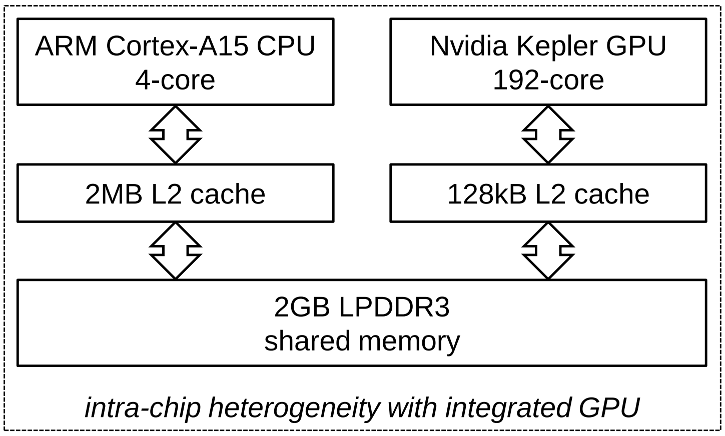


Figure A.7: Wimpy system with ARM CPU and integrated Nvidia GPU

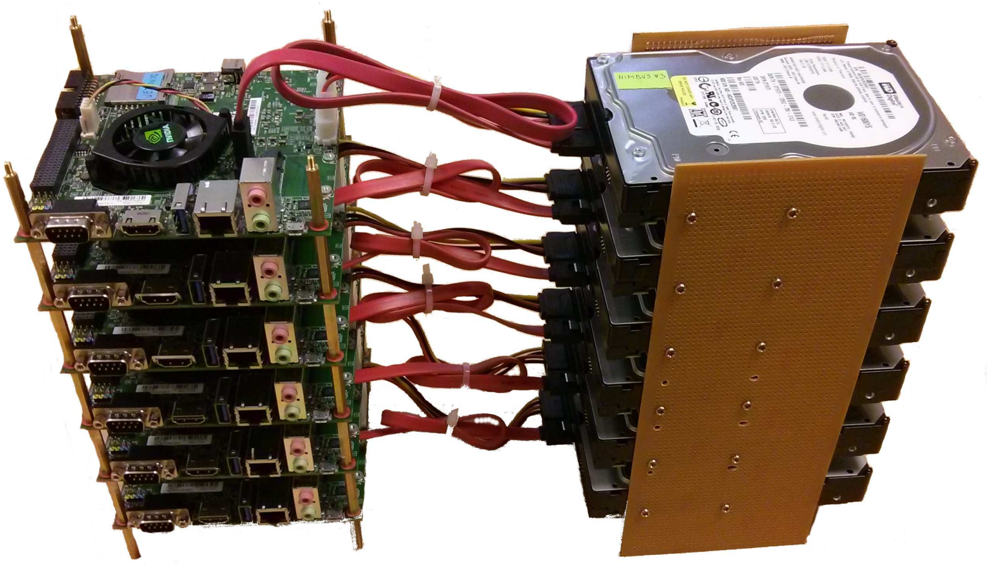


Figure A.8: Wimpy Jetson TK1 cluster

We connect a 512 GB SSD on the SATA port to store our datasets and workloads. For the comparison between heterogeneous and homogeneous systems, we use the system without GPU in **Jetson**-only configuration and the system with GPU in **Jetson+GPU** configuration.

Jetson runs Ubuntu 14.04 OS with Linux kernel 3.10.40 installed on the eMMC. To compile native code, we use *gcc* 4.8.2 and *nvcc*, while Java runs on Oracle's Java Virtual Machine (JVM) version 1.8.0_06. By default, Jetson TK1 comes with CUDA toolkit 6.0 and associated Nvidia drivers. However, we encountered kernel panics and errors when running Hadoop-CUDA on this default setup. Upgrading the drivers and CUDA toolkit to version 6.5 solved these issues.

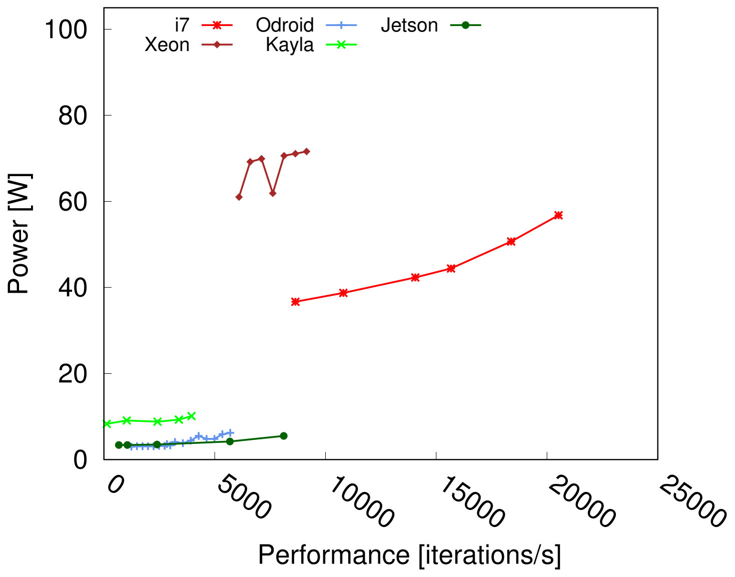
A.2 Characterization

To assess system performance and to evaluate the gap between wimpy and brawny systems, we perform a measurement-driven characterization of the six systems at the CPU, GPU, memory, storage and networking level. The results are summarized in Table A.1.

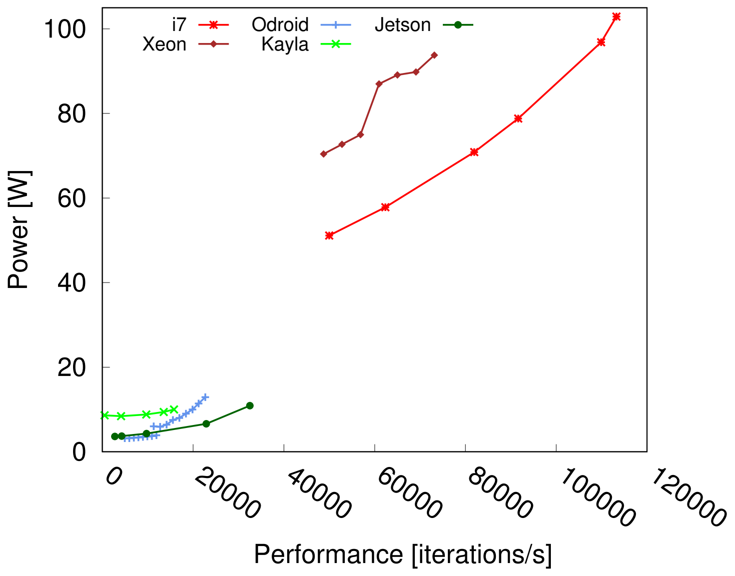
Table A.1: Systems characterization

		Brawny			Wimpy			
		homogeneous	with discrete GPU	with discrete GPU	many-core		with discrete GPU	with integrated GPU
		Xeon	**i7**	**Amazon**	**ARM big.LITTLE (Odroid)**		**Kayla**	**Jetson**
					LITTLE	big		
Specs	System	Supermicro 813M	Dell Optiplex	g2.2xlarge	Odroid XU		Kayla	Jetson TK1
	CPU	Intel Xeon E5-2603	Intel Core i7	Intel Xeon E5-2670	ARM Cortex-A7	ARM Cortex-A15	ARM Cortex-A9	ARM Cortex-A15
	ISA	x86-64	x86-64	x86-64	ARMv7l	ARMv7l	ARMv7l	ARMv7l
	Cores	4	4 (8 threads)	8 (virtual)	4	4	4	4
	Frequency [GHz]	1.20 - 1.80	1.60 - 3.40	2.60	0.25 - 0.60	0.60 - 1.60	0.05 - 1.40	0.05 - 2.32
	L1 D-Cache (per core) [kB]	32	32	32	32	32	32	32
	L2 Cache [MB]	1	1	2	2	2	1	2
	L3 Cache [MB]	10	8	20	-	-	-	-
	GPU Architecture	-	Nvidia Maxwell	Nvidia Kepler	-	-	Nvidia Maxwell	Nvidia Kepler
	Memory Type	DDR3	DDR3	-	LPDDR3	LPDDR3	LPDDR2	LPDDR3
	Memory Size [GB]	16	8	15	2	2	2	2
CPU	Dhrystone [MIPS]	10350	22161	17726	2192	4936	2526	7434
	CPU power [W]	15.0	39.5	-	0.5	3.4	2.3	3.0
	System power [W]	50.0	65.3	-	4.4	7.3	10.0	6.1
	Dhrystone PPR [MIPS/W]	690	561	-	4384	1452	1106	2470
	CoreMark [iterations/s]	9456	17237	15664	3025	5628	3952	8155
	CPU power [W]	15.6	25.9	-	0.3	2.5	2.4	2.7
	System power [W]	50.6	51.7	-	4.2	6.4	10.1	5.9
	CoreMark PPR [iterations/sW]	606	666	-	10082	2251	1658	3009
	Java [MIPS]	653	1365	1083	242	605	411	880
	CPU power [W]	16.5	37.0	-	0.3	3.4	2.6	3.5
	System power [W]	51.5	62.8	-	3.0	6.1	10.3	6.7
	Java PPR [MIPS/W]	40	37	-	807	178	159	251
	Idle system power [W]	35.0	25.8	-	3.1	3.1	7.7	3.2
GPU	Performance [GFLOPS]	-	1514	2157	-	-	1512	209
	Average system power [W]	-	105.2	-	-	-	44.0	6.1
	Idle system power [W]	-	40.6	-	-	-	19.2	3.2
Storage	Write throughput [MB/s]	165.0	198.0	123.0	32.6	39.2	89.7	161.0
	Read throughput [MB/s]	173.0	284.0	138.0	118.0	121.0	85.1	275.0
	Buffer read throughput [GB/s]	4.6	10.0	7.3	0.8	1.2	0.4	1.6
Network	TCP bandwidth [Mbits/s]	942	-	1080	199	308	-	921
	UDP bandwidth [Mbits/s]	811	-	808	295	420	-	804
	Ping latency [ms]	0.2	-	0.2	0.7	0.7	-	2.7

To assess CPU performance, we run three benchmarks, (i) the traditional Dhrystone benchmark [96], (ii) the emerging CoreMark benchmark which is increasingly used by hardware vendors, including ARM [16], and (iii) our in-house microbenchmark to asses Java performance. We first measure CPU MIPS native performance with traditional Dhrystone benchmark [96]. We are compiling the code with *gcc* using maximum level of optimization, `-O3`, and tuning the code for the target processor (e.g. for ARM big.LITTLE we use `-mcpu=cortex-a7 -mtune=cortex-a7` for little cores, and `-mcpu=cortex-a15 -mtune=cortex-a15` for big cores). We observe that Dhrystone MIPS are proportional with core clock frequency. However, in terms of Dhrystone MIPS per MHz, we obtain a surprising result on ARM big.LITTLE: little cores perform 21% better than big cores, as per MHz. This is unexpected because ARM reports that Cortex-A7 has lower Dhrystone MIPS per MHz than Cortex-A15, but using its internal *armcc* compiler [17]. We conclude that it is the *gcc* way of generating machine code that leads to these results. To check our results, we run newer CoreMark CPU benchmark which is being increasingly used by embedded market players, including ARM [16]. We use compiler optimization flags to match those employed in the reported performance results for an ARM Cortex-A15. More precisely, we activate NEON SIMD (`-mfpu=neon`), hardware floating point operations (`-mfloat-abi=hard`) and aggressive loop optimizations (`-faggressive-loop-optimizations`). We obtain a score of 3.52 per core per MHz, as opposed to the reported 4.68 for ARM Cortex-A15. We attribute this difference to different compiler and system setup. However, little cores are again more energy-efficient, obtaining more than half the score of big cores with only 0.3 W of power. The difference between ARM cores and Xeon cores is similar for both Dhrystone and CoreMark benchmarks. ARM Cortex-A9 core is around two times slower than Cortex-A15 and four times slower than Intel i7 processor. However, A15's clock frequency is 65% higher, while i7's clock frequency is more



(a) One core



(b) All cores

Figure A.9: Systems power-performance profile

than double compared to A9. Interestingly, A15 uses almost the same power per core as A9 to deliver twice the performance. This shows the significant improvements in energy efficiency of ARM processors. On the other hand, Intel processor uses five times more power compared to the wimpy processors. In terms of idle power, which is measured when the systems are running only the OS, Odroid XU and Jetson systems are the most efficient, while the Kayla-only and i7-only systems consumes around two and eight time more power, respectively.

We summarize CoreMark performance and power results of all systems in Figure A.9 when the benchmark is running on a single core and on all available cores. We observe a big gap of 30W in terms power consumed between wimpy and brawny systems. On the other hand, performance gap is much smaller and is mainly due to higher core count of x86-64 systems. This PPR plots demonstrate the power efficiency of wimpy systems based on ARM cores.

Since Big Data frameworks, such as Hadoop and Spark, run on top of Java Virtual Machine, we also benchmark Java execution. We develop a synthetic benchmark performing integer and floating point operations such that it stresses core's pipeline. In correlation with Dhrystone and CoreMark results, we observe that MIPS performance is proportional to core clock frequency. Interestingly, ARM Cortex-A15 cores on Tegra K1 deliver more MIPS that Xeon cores. On ARM big.LITTLE, the little Cortex-A7 cores obtain less than half the MIPS of Cortex-A15 cores. On the other hand, the big Cortex-A15 cores achieve just 7% fewer MIPS than Xeon cores, but using quarter the power.

In summary, ARM cores outperform by an order of magnitude Intel cores in terms of PPR, but at the cost of lower raw performance. For ARM cores, Cortex-A7 achieves the best PPR, but the lowest raw performance. ARM Cortex-A15 cores on Jetson TK1 achieve better PPR than ARM Cortex-A15 cores on Odroid XU. We attribute this to different hardware implementation and different core clock frequency.

With the GPU being a key device in heterogeneous systems, we measure peak performance in terms of FLOPS, data transfer bandwidth and latency between main (*host*) memory and GPU (*device*) memory. We use MaxFlops to get arithmetic performance and BusSpeedDownload for host-device bandwidth, both from level zero benchmarks of Scalable HeterOgeneous Computing (SHOC) [32] benchmarking suite. However, on Jetson TK1 integrated system there is no point in measuring host-device bandwidth since data resides in the same shared memory. Thus, we additionally measure the latency using our custom benchmark that supports unified memory feature [50]. Benchmarking code is compiled with *nvcc* compiler from CUDA toolkit with `-arch=sm_50`, `-arch=sm_30` and `-arch=sm_32` flags on i7+GPU, Amazon g2.2xlarge and Jetson+GPU, respectively. For the Maxwell GTX 750 Ti GPU, the processing performance is the same on both Kayla+GPU and i7+GPU systems, peaking at more than 1500 GFLOPS when performing single-precision arithmetic.

Interestingly, to deliver this performance, the i7+GPU draws more than two times the power of the Kayla+GPU, as shown in Table A.1. We attribute this difference to at least three factors. First, by analyzing benchmark behaviour, we discover that it highly utilizes one CPU core. From our CPU analysis, one Intel i7 core draws around 25 W compared to around 2.5 W for one Cortex-A9 core. Second, the PCI Express on Kayla system consists of only four lanes, being more energy-efficient but having lower throughput. Although we are not able to measure PCI Express stand-alone energy usage, we believe it is much lower on the Kayla system. Third, since Kayla board is a low-power, embedded-class system, it uses more efficient circuitry, including power-regulators. In terms of idle power, the discrete Maxwell GPU consumes more than 10 W. Hence, the idle power of Kayla doubles by adding this GPU. In contrast, Jetson's idle power is the same since the

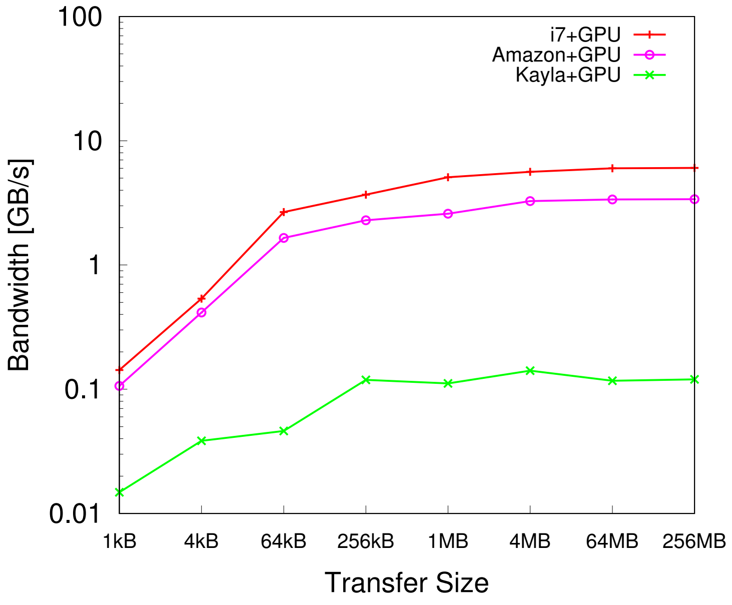


Figure A.10: Host-device transfer bandwidth

integrated GPU is not activated during idle periods. These results emphasize the energy efficiency of the Jetson system.

For host-device transfers, i7+GPU obtains the best results, being two times higher than g2.2xlarge and 40 times higher compared to the wimpy Kayla+GPU, as shown in Figure A.10. This explains the poor performance of this configuration for MapReduce workloads that require significant transfers. In terms of latency, Jetson with integrated GPU is surprisingly not the best in all cases. While for larger transfers Jetson+GPU has ten times lower latency, for small transfers of less than 128kB the i7+GPU and g2.2xlarge show lower latency. As expected, Kayla+GPU has the highest latency. This, together with the low bandwidth, seriously affects the ability of Kayla+GPU to process data-intensive applications.

It is a known fact that wimpy systems have smaller memories than brawny systems. Less well-known is the performance of these memories. We evaluate main memory bandwidth using *pmbw* 0.6.2 (Parallel Memory Bandwidth Benchmark) tool [23]. Figure A.12 plots the memory bandwidth comparison of all systems, while Figure A.13 shows the comparison within ARM big.LITTLE system. In-

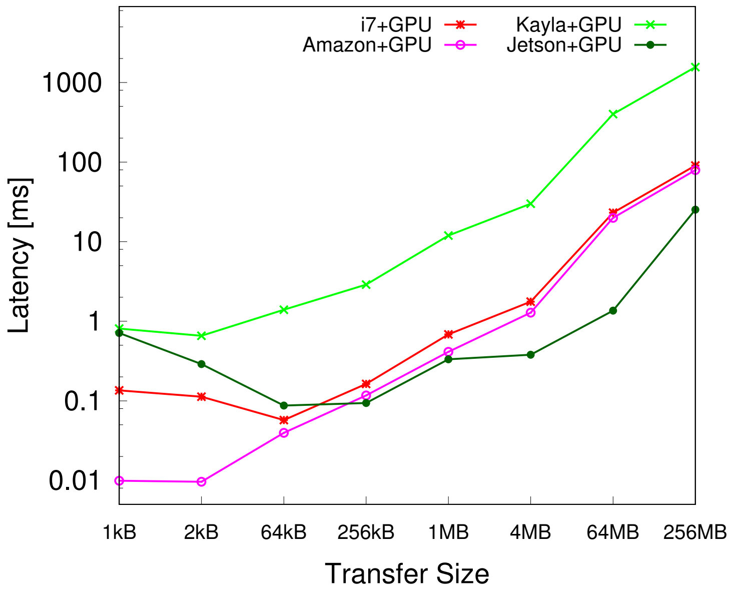


Figure A.11: Host-device transfer latency

terestingly, Jetson exhibits main memory bandwidths close to those of Xeon and i7 systems, while Odroid XU and Kayla exhibit two and four times lower bandwidths. We attribute this high memory bandwidth of Jetson to newer technology and better memory controller implementation. When data fits into cache, Xeon, i7 and g2.2xlarge have bandwidths of 450 GB/s. The four Cortex-A15 cores on Jetson TK1 exhibit 6-8 times lower bandwidth, while the A15 cores on Odroid XU exhibit around ten times lower bandwidth compared to the brawny systems.

Within Odroid XU, the bandwidth of big.LITTLE configuration is the same as big-only configuration when accessing main memory, but lower when accessing the cache. This is attributed to the cache penalty of switching from little to big cores. The bandwidth of little cores is two times lower compared to big cores.

For storage and network throughput and latency, we use Linux tools such as *dd*, *ioping*, *iperf* and *ping*. Read and write throughput is in the range of 100 to 200 MB/s, except for Kayla and for write throughput on Odroid XU. We attribute lower throughput on Kayla to a poor implementation of SATA 2 controller, while on Odroid there is an issue with the NAND flash eMMC storage. When using

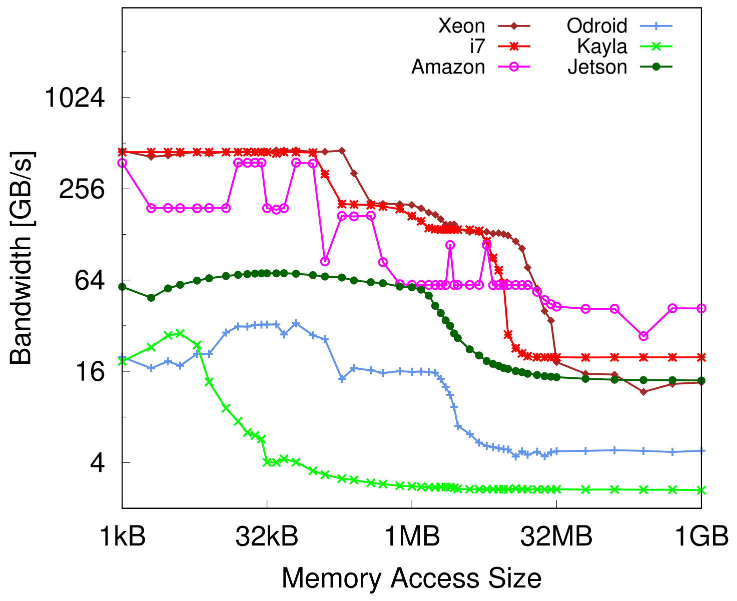


Figure A.12: Memory bandwidth comparison

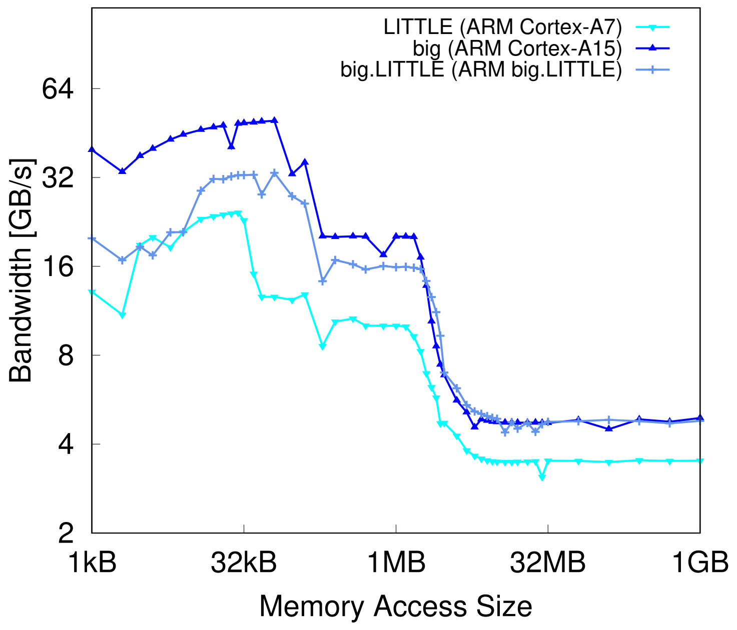


Figure A.13: Memory bandwidth comparison of ARM big.LITTLE

little cores, the throughput is even smaller, suggesting that disk driver and file system have high CPU usage. Since modern operating systems are caching small files in memory, we also measured buffered read throughput. The results are correlated with memory bandwidth values considering that only one core is used. For example, buffered read on big ARM Cortex-A15 cores has 1.2 GB/s throughput, while the main memory bandwidth when using all four cores is 4.9 GB/s. Brawny systems use their superior memory size and bandwidth to outperform the wimpy systems at buffered reads. In term of networking, systems with native Gigabit Ethernet interface reach TCP bandwidths of more than 900 Mbits/s. On Odroid XU, the Gigabit interface delivers only 300 Mbits/s TCP bandwidth, being limited by the USB 3.0 connection. In summary, wimpy systems have significantly lower performance, especially at memory level, which negatively impacts data analytics performance.

Appendix B

Data-parallel Execution on Many-core Heterogeneous Systems

With the explosion of Big Data analytics, datacenter designers still advocate the usage of high-performance systems, such as those based on Intel Xeon or AMD Opteron CPUs [56]. However, these systems have high idle power and generate large amounts of heat. Hence, a significant part of datacenter wasted energy is due to cooling. On the other hand, many datacenter jobs, such as Big Data analytics as opposed to compute-intensive applications, stress storage and network subsystems rather than the CPU.

In the last five years, mobile device market has driven the improvement of energy-efficient systems such as those based on ARM processors. These low-power systems are now capable of running modern operating systems and a full range of applications. As phones gradually evolved into smartphones, their CPUs became increasingly complex. Initially, these processors had single in-order cores, also called *little* cores due to their low power consumption and low performance. Gradually, these devices have integrated out-of-order, high performance cores which are called *big* cores. With the entrance into *dark silicon* era, ARM developed heterogeneous big.LITTLE architecture [19]. This architecture combines little and big cores on the same processor. The typical big.LITTLE implementation uses ARM Cortex-A7 in-order, low-power cores together with ARM Cortex-A15 out-of-order, power hungry cores. Mobile applications typically use this architecture by scheduling background tasks on little cores and critical task on big cores. But it is not clear how these platforms are able to handle server-class or Big Data analytics workloads.

In this section, we present a measurement-driven analysis of Big Data MapReduce processing on low-power ARM big.LITTLE systems, in comparison with traditional Intel Xeon server systems. This analysis is part of our high-level approach of modeling Big Data execution and efficiently mapping parallel tasks on heterogeneous resources. We first summarize the hardware and software setup, and then present the measurement-driven analysis of running a series of well-known Big Data workloads on Hadoop. Lastly, we evaluate the total cost of ownership (TCO) of running Big Data workloads on heterogeneous systems.

B.1 Setup

To characterize Big Data execution on small nodes and to compare them with traditional server-class nodes, we measure total execution time and total energy at cluster level. We run typical Big Data analytics MapReduce applications on Hadoop 1.2.1, the open-source implementation for MapReduce framework. We select a subset of the applications presented in Section 4.1 to stress each subsystem, such as the CPU using Pi and Kmeans, the memory using Terasort and the I/O using WordCount and Grep. Selected applications are presented in Table B.1. All

Table B.1: Big Data workloads

Workload	Input Type	Input Size	Bottleneck
Terasort	synthetic	12 GB	I/O, Memory
Pi	synthetic	16×10^9 samples	CPU
Kmeans	Netflix	4 GB	CPU, Memory
WordCount	Wikipedia	12 GB	I/O
Grep	Wikipedia	12 GB	I/O

these applications are part of Hadoop examples, except Kmeans which is adapted from PUMA benchmarking suite [2]. The input dataset for Kmeans is taken from the same source and trimmed to 4 GB. For benchmarks that take text input, we use Wikipedia's articles latest dump and trim it to 12 GB. This size is chosen such that, even when running on a six nodes cluster, each system processes 2 GB of data which is more that can fit into 2 GB of Odroid XU main memory since the OS and Hadoop are also using memory. All workloads are executed three times and the average execution time and energy is reported.

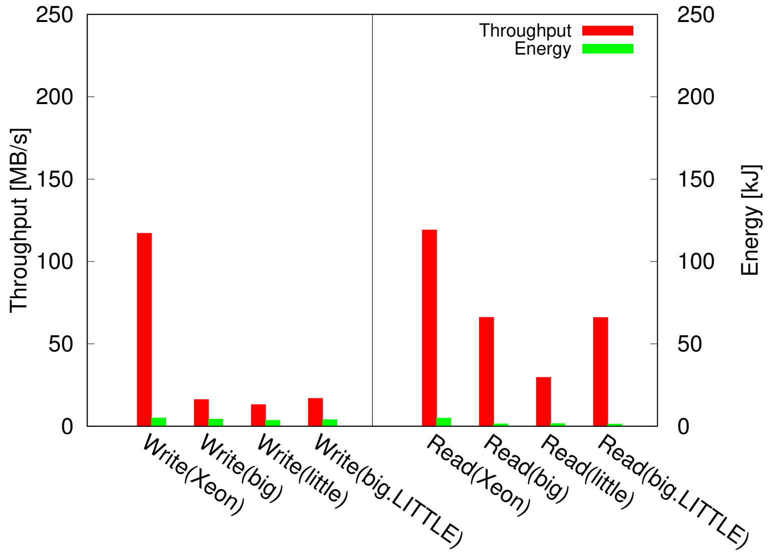
When running MapReduce programs, we set the number of slots equal to the number of cores on each node. For a fair comparison, we physically disable four cores of the Xeon system and set the number of slots to four. We run the workloads on clusters of one, two, four and six Odroid XU, respectively, Xeon E5-2603 based systems. For power and energy measurements, we use Yokogawa WT210 power monitor connected to cluster's AC input line. A controller system is used to start the benchmarks and to collect all the logs. This setup is similar to the one depicted in Figure 4.1.

B.2 Time-Energy Analysis

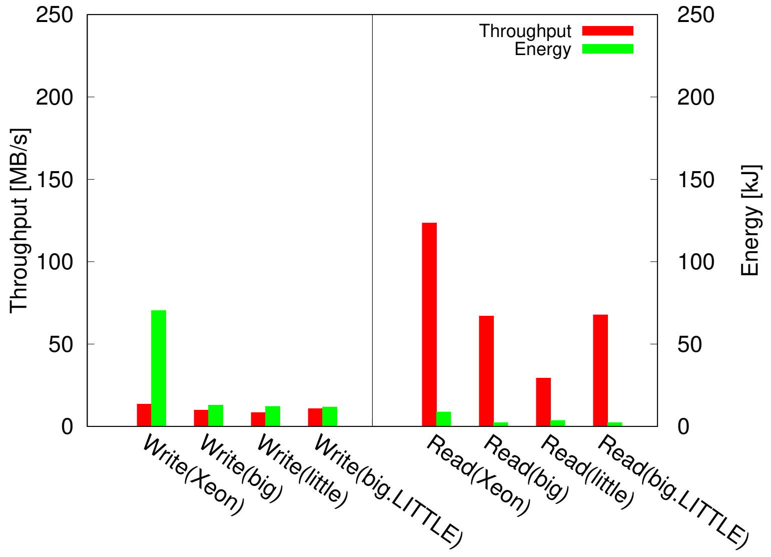
First, we analyze the performance of Hadoop Distributed File System (HDFS) which is the underlying file system for many Big Data frameworks such as Hadoop, Hive, Spark, among others. We measure the throughput and energy usage of HDFS read and write distributed operations using Hadoop's *TestDFSIO* benchmark with 12 GB input. Figure B.1 plots the throughput, as reported by *TestDFSIO*, and measured energy consumption of write and read on single node and 6-node clusters. The throughput significantly decreases when writing on multiple nodes, especially for Xeon nodes. This decrease occurs because of HDFS replication mechanism, which, by default, replicates each block three times. The additional network and storage operations due to replication increase the execution time and lower the overall throughput. This observation is validated by the less visible degradation of throughput for read operation. The increasing execution time of write on multiple nodes leads to higher energy consumption, especially for Xeon nodes. On a 6-node cluster, the write throughput of Xeon is two times higher compared to ARM, but the energy usage is more than four times bigger. For read, Xeon's throughput is three times better that ARM's big.LITTLE, while the energy ratio is five. On ARM nodes with little cores, the execution times of HDFS write and read operations increase due to lower JVM performance. Hence, the energy consumption is higher compared to running on big and big.LITTLE configurations.

In summary ARM big.LITTLE is more energy-efficient than Xeon when executing HDFS read and write operations, at the cost of 2-3 times lower throughput.

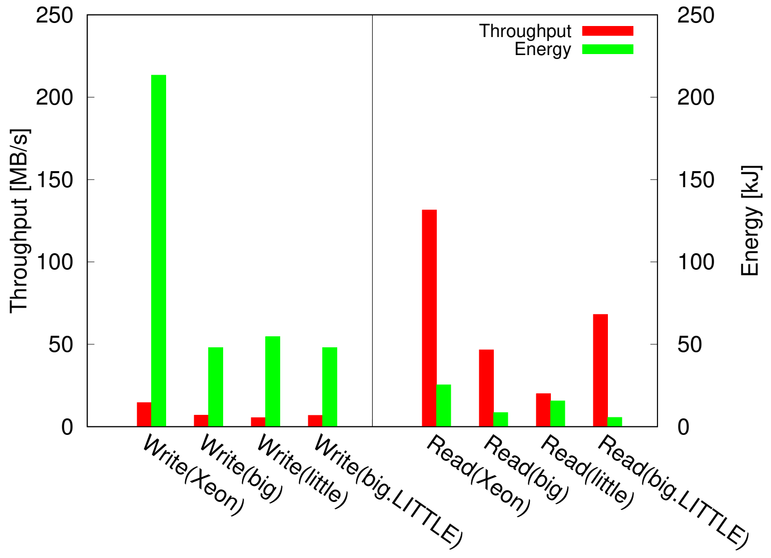
Second, we evaluate time performance and energy-efficiency of Hadoop by running six widely-used workloads, as shown in Table B.1. We use default Hadoop settings, except that we set the number of slots to four such that it equals the number of cores on each node. Using this configuration, all workloads run without errors, except for Terasort and Kmeans which fail on Odroid XU due to insufficient memory. After experimenting with more alternative configurations, we found two that allow both programs to finish without failure. Firstly, we decrease the number of slots to two on Odroid XU. Secondly, we keep using four slots but limit the



(a) On one node



(b) On two nodes



(c) On six node

Figure B.1: HDFS performance

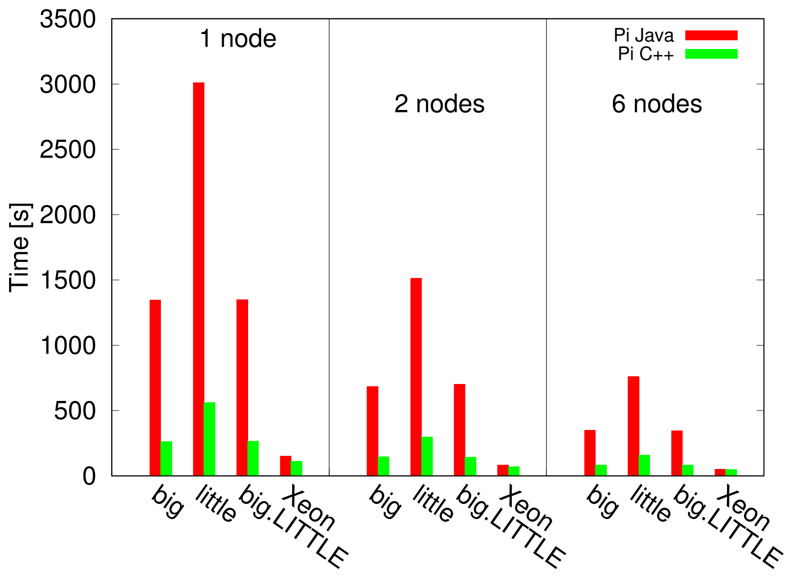


Figure B.2: MapReduce Pi estimator in Java and C++

`io.sort.mb` to 50 MB, half of its default value. These two settings have different effects on the two programs. For example, on 4-node cluster, Terasort running on two slots is 10-20% faster than using a limited `io.sort.mb`. This result is due to the fact that Terasort is data-intensive, hence, it benefits less form using more cores but having limited memory buffer. On the other hand, Kmeans benefits more from running on four slots, being 20% faster on big cores and 35% faster on little cores, compared to running on two slots. Kmeans is a compute-intensive workload executing a large number of floating point operations in both map and reduce phases. Thus, it benefits from running on higher core counts. In the remainder of this paper, we present the results on two slots for Terasort, and on four slots with `io.sort.mb` of 50 for Kmeans, when running on ARM big.LITTLE nodes.

When running the experiments, we observe low performance of Pi on Odroid XU. Compared to Xeon, Pi on big and big.LITTLE runs 7-9 times slower, and on little cores up to 20 times slower. This is surprising because Pi is compute-intensive and we show in Section A.2 that the performance ratio between Xeon

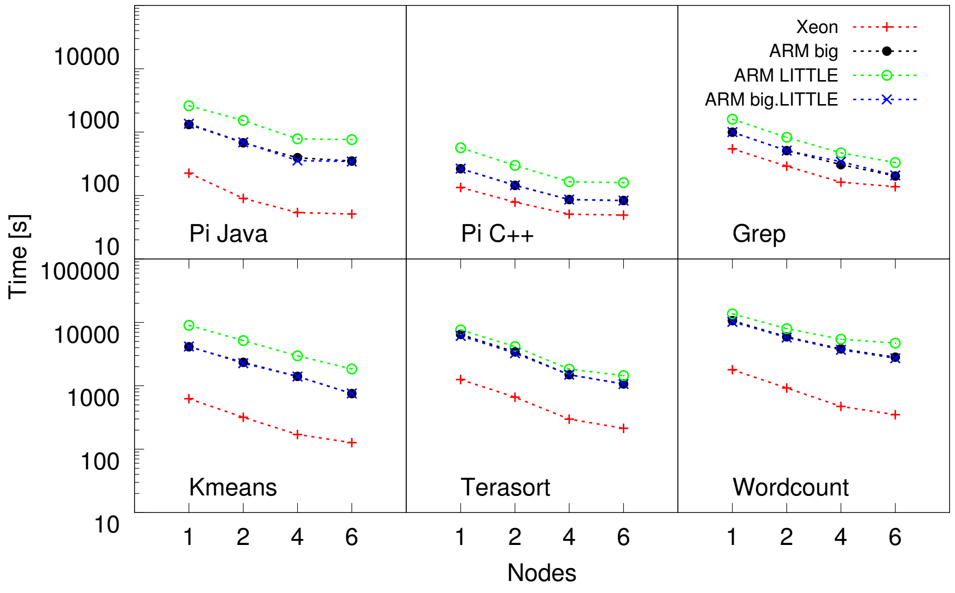


Figure B.3: MapReduce scaling

and ARM cores is at most five. We further investigate the cause of this result. Firstly, we profile TaskTracker execution on Odroid XU. We observed that JVM spends 25% of the time in `__udivsi3`. This function emulates 32-bit unsigned integer division in software, although the Exynos 5410 SoC on Odroid XU board supports `UDIV` hardware instruction. But other SoCs may not implement this instruction, since it is defined as optional in ARMv7-A ISA [18]. Thus, JVM uses the safer approach of emulating it in software. Secondly, we port Pi in C++ and run it using Hadoop Pipes mechanism. We use the same *gcc* compilation flags as for native benchmarks in Section A.2. The comparison between Java and C++ implementations is shown in Figure B.2. Compared to original Java version, C++ implementation is around five times faster on ARM nodes and only 1.2 times faster on Xeon-based nodes. With this minor software porting, we obtain a significant improvement in execution time which leads to energy savings, as we further show. In the remainder of this section, we show the results for both Pi Java and Pi C++ implementations.

We present time and energy performance of the six workloads on Xeon and

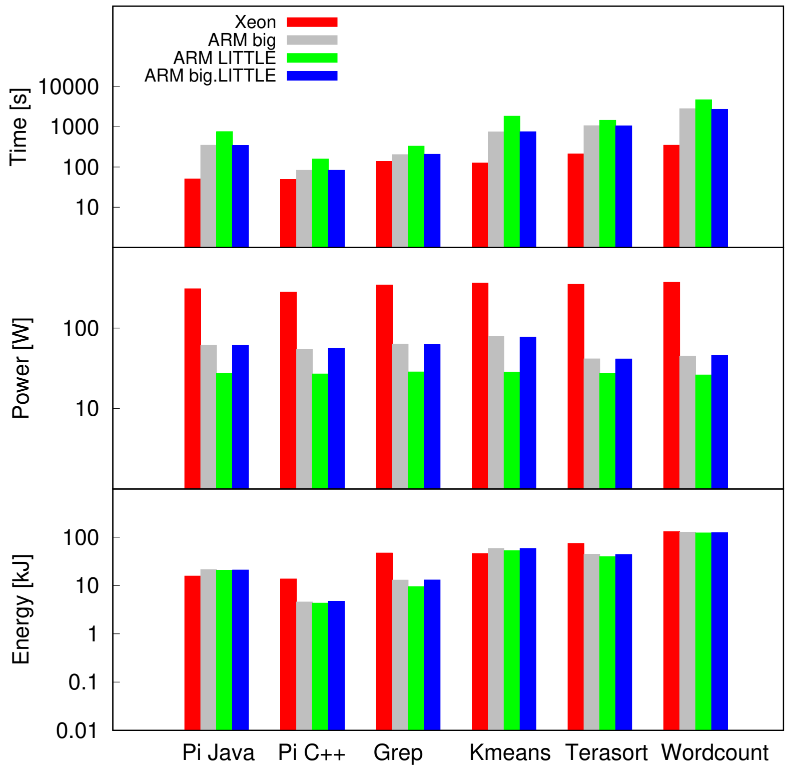


Figure B.4: MapReduce on 6-node cluster

ARM clusters. First, since scalability is a main feature of MapReduce framework, we investigate how does Hadoop scale on clusters of small nodes. We show time scaling in log scale on four cluster sizes in Figure B.3. All workloads exhibit sub-linear scaling on both Intel and ARM nodes, which we attribute to housekeeping overheads of Hadoop when running on more nodes. When the overheads dominate the useful work, the scaling degrades. For Pi workload running on six nodes there is too little useful work for mappers to perform, hence, there is not much improvement in the execution time on both types of servers. On the other hand, Kmeans and Grep exhibit higher speedup on the 6-node ARM cluster compared to Xeon because the slower ARM cores have enough compute-intensive work to perform.

Figure B.4 shows the time, power and energy of 6-node clusters using log scale. Based on the energy usage, the workloads can be categorized into three classes:

Table B.2: MapReduce performance-to-power ratio[1]

Workload	**Xeon**				**ARM (Odroid XU)**											
					big				**LITTLE**				**big.LITTLE**			
	1	**2**	**4**	**6**	**1**	**2**	**4**	**6**	**1**	**2**	**4**	**6**	**1**	**2**	**4**	**6**
Pi Java	1.44	**1.58**	0.88	0.63	0.68	0.60	0.60	0.56	0.78	0.83	0.80	0.58	0.67	0.60	0.61	0.57
Pi C++	2.51	1.89	1.04	0.71	3.23	3.03	2.95	2.64	**4.56**	4.37	4.01	2.78	3.33	2.95	2.78	2.56
Grep	0.56	0.46	0.27	0.21	1.03	0.93	0.92	0.92	**1.47**	1.34	1.31	1.27	1.03	0.93	0.86	0.92
Kmeans	**0.50**	0.41	0.25	0.22	0.21	0.19	0.19	0.20	0.28	0.25	0.23	0.23	0.21	0.19	0.18	0.20
Terasort	0.28	0.22	0.15	0.14	0.31	0.25	0.30	0.27	**0.35**	0.28	**0.35**	0.30	0.32	0.25	0.30	0.27
Wordcount	0.17	0.14	0.09	0.08	0.12	0.11	0.10	0.09	**0.18**	0.16	0.12	0.10	0.12	0.11	0.10	0.10

- Pi Java and Kmeans execution times are much larger on ARM compared to Xeon. Both workloads incur high CPU usage on ARM, which results in high power usage. The combined effect is a slightly higher energy usage on ARM nodes.

- Pi C++ and Grep exhibit a much smaller execution time gap. Both are compute-intensive and have high power usage, but overall, their energy usage is significantly lower on ARM.

- Wordcount and Terasort are data-intensive workloads, as indicated by lower power usage on ARM compared to the other workloads. They obtain better execution time on Xeon due to higher memory and storage bandwidths. However, time improvement does not offset the higher power usage of Xeon, therefore, energy on ARM is lower.

We sum up by showing the PPR of all workloads on all cluster configurations as a heat-map in Table B.2. PPR is defined as the amount of useful work performed per unit of energy. For workloads that scan all input, we compute the PPR as the ratio between input size and energy. For Pi, the input file contains the number of samples to be generated during the map phase. Hence, we express the PPR as millions of samples (Msamples) per unit of energy. Higher (green) PPR represents a more energy-efficient execution. In correlation with our classification, Pi Java and Kmeans exhibit better PPR on Xeon, while all other workloads have the

[1]Values represent 10^6 samples per Joule for Pi, and MB per Joule for the other workloads.

highest PPR on ARM little cores. As indicated in Table B.2, single node achieves maximum PPR because there is no communication overhead and fault tolerance mechanism as on multi-node clusters.

B.3 Time-Energy Performance Equivalence

In the previous section, we have shown that a single wimpy node always exhibit longer MapReduce execution time compared to a single brawny node. An important question is how many wimpy nodes achieve the same execution time as one brawny node and how much energy are they using. Figure B.5 answers this question by showing how many ARM-based nodes can achieve the execution time of one Xeon node. We select ARM big.LITTLE configurations which exhibit the closest execution time compared to one Xeon. For compute-intensive workloads, such as Pi C++ and Grep, two wimpy nodes achieve the same execution time and use less energy compared to one Xeon node. For I/O- and mix-intensive workloads, six wimpy nodes can perform the job of one Xeon node, but use more energy. For Wordcount, the difference between six ARM nodes and one Xeon node is large, and thus, we estimate based on the scaling behavior that eight ARM nodes exhibit a closer execution time, but use almost double the energy.

B.4 Cost Analysis

We analyze the total cost of ownership (TCO) of executing Big Data applications on emerging low-power ARM servers, in comparison with traditional x86-64 servers. We derive lower and upper bounds for per hour cost of compute- and data-intensive workloads on a single node. We consider Pi and Terasort as representatives for compute- and data-intensive workloads, respectively. Moreover,

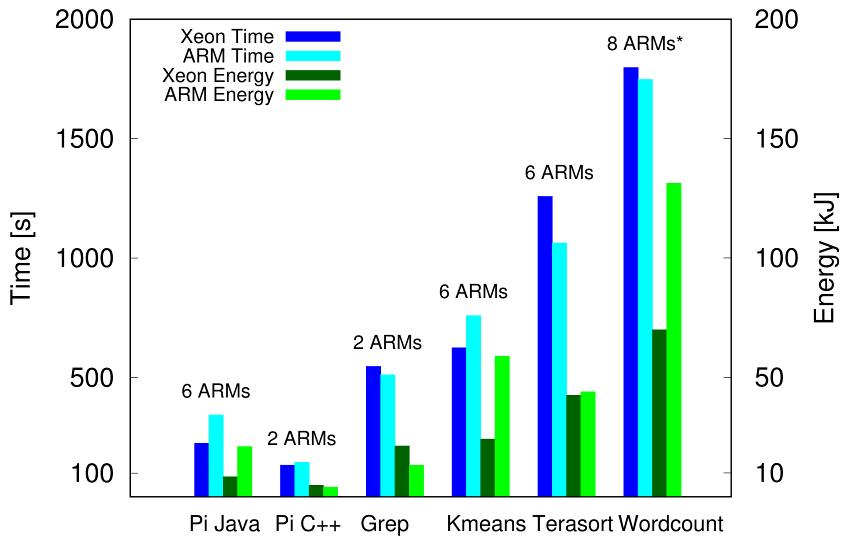


Figure B.5: Xeon-ARM performance equivalence

we use execution time and energy results of Pi C++ implementation because it better exploits ARM nodes.

Throughout this section, we use a series of notations and default values as summarized in Table B.3. All costs are expressed in US dollars. The values in Table B.3 are either based on our direct measurements or taken from the literature, as indicated[2]. For example, we assume three years of typical server lifetime and 12 years lifetime for a datacenter [56]. For typical server utilization, we consider a lower bound of 10%, typical for cloud servers [69], and an upper bound of 75% as exhibited by Google datacenters [56].

The cost of electricity is a key factor in the overall datacenter costs. But electricity price is not the same all-over the world. Thus, we consider more alternatives for servers' location and electricity price [97]. Among these alternatives, we select a lower bound of 0.024 $/kWh (price of electricity in Russia) and an upper bound of 0.398 $/kWh (price in Australia). Although we acknowledge that datacenter location may also influence equipment, hosting and manpower costs, throughout

[2]Listed values are marked with * if they are taken from the literature, with $^+$ if they are based on our measurements, and with $^\#$ if they represent output values.

Table B.3: TCO notations and values

Notation	Value	Description
$C_{s,Xeon}$	\$1100 $^+$	cost of Xeon-based server node
$C_{s,ARM}$	\$280 $^+$	cost of ARM-based server node
T_s	3 years *	server lifetime
U_l	10% *	low server utilization
U_h	75% *	high server utilization
C_d	#	datacenter total costs
T_d	12 years *	datacenter lifetime
C_p	#	electricity total costs
C_{ph}	*	electricity cost per hour
P_a	$^+$	average server power
$P_{p,Xeon}$	55 W $^+$	Xeon-based node peak power
$P_{p,ARM}$	16 W $^+$	ARM-based node peak power
$P_{i,Xeon}$	35 W $^+$	Xeon-based node idle power
$P_{i,ARM}$	4 W $^+$	ARM-based node idle power

this study we consider only the difference in electricity price.

B.4.1 Marginal Cost

We begin by describing a simple cost model which incorporates equipment and electricity costs. This model estimates the marginal cost of self-hosted systems, being suitable for small, in-house computing clusters. Total cost is

$$C = C_s + C_p \tag{B.1}$$

where electricity cost for server lifetime period is:

$$C_p = T_s \cdot C_{ph} \cdot (U \cdot P_a + (1 - U) \cdot P_i) \tag{B.2}$$

We further investigate the effects of server utilization and idle power on marginal cost. As we define lower and upper bounds for server utilization, there are two scenarios for evaluating electricity costs. Firstly, given a low Xeon server utiliza-

Table B.4: Effect of server utilization on marginal cost

Job type	**Utilization ratio [%]**	**Server ratio**	**Min cost [$/h]**		**Max cost [$/h]**	
			Xeon	**ARM**	**Xeon**	**ARM**
compute-intensive	10:20	1:1	0.043	0.011	0.044	0.013
data-intensive	10:49	1:1	0.043	0.011	0.056	0.013
compute-intensive	75:82	1:2	0.043	0.022	0.060	0.031
data-intensive	75:86	1:6	0.043	0.065	0.059	0.079

tion of 10% and the execution times of Pi and Terasort workloads on Xeon and ARM nodes, we obtain two values for ARM-based server utilization. For Pi, ARM server exhibits 20% utilization, while for Terasort, the utilization increases to almost 50%. Secondly, given the upper bound of 75% for Xeon server utilization, we obtain over 100% utilization for ARM server. Thus, we must employ more than one ARM server to execute the workload of one Xeon. We use server substitution ratios derived in Section B.3 and depicted in Figure B.5. For compute-intensive Pi, we use two ARM servers with 82% utilization to achieve the performance of one Xeon server. For data-intensive Terasort, we use six ARM servers with 86% utilization to execute the same workload as one 75%-utilized Xeon. The six ARM servers occupy less space than one rack-mounted traditional server but may have a higher equipment cost. We present the results for both scenarios as cost per hour in Table B.4. For low utilization, the cost per hour of ARM is almost four times lower compared to Xeon. Moreover, compute- and data-intensive jobs have the same cost. On the other hand, the cost of highly utilized servers is slightly higher. Surprisingly, for data-intensive jobs, ARM incurs up to 50% higher cost because six ARM servers are required to perform the work of one Xeon.

Next, we investigate the influence of idle power, as a key factor in total electricity costs. This influence may be alleviated by employing energy-saving strategies, such as All-In Strategy [65]. This strategy assumes that servers can be inducted to a low-power state during inactive periods. At certain intervals, they are woken-up

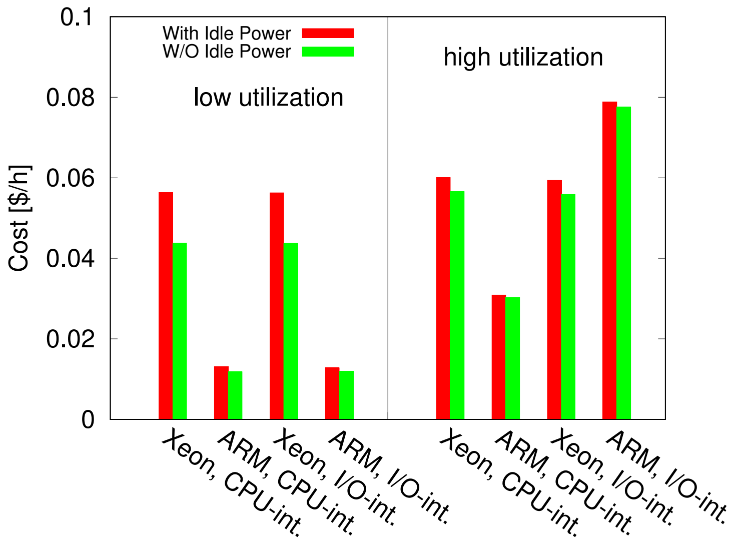


Figure B.6: Effect of idle power on marginal cost

to execute the jobs, and afterwards put back to sleep. In reality, servers consume a small amount of power in deep-sleep or power-off mode and may incur high power usage during wake-up phase. However, we assume that during inactive periods servers draw no power, and perform the study on both utilization scenarios described above. With these assumptions, the influence of idle power is more visible on low-utilized Xeon servers, as shown in Figure B.6. In this case, putting Xeon servers to sleep can reduce hourly cost by 22%. For ARM servers, cost reduction is 6–10% since idle power is much lower. At high utilization, the reductions are smaller because the servers are active most of the time.

B.4.2 Total Cost

We analyze a more complex TCO model which includes datacenter costs. We use Google TCO calculator which implements the model described in [56]. For this model, total cost is

$$C = C_d + C_s + C_p \tag{B.3}$$

Table B.5: Effect of server utilization on TCO

Job type	**Utilization ratio [%]**	**Server ratio**	**Min cost [$/h]**		**Max cost [$/h]**	
			Xeon	**ARM**	**Xeon**	**ARM**
compute-intensive	10:20	1:1	0.066	0.018	0.086	0.025
data-intensive	10:49	1:1	0.066	0.017	0.085	0.021
compute-intensive	75:82	1:2	0.066	0.035	0.086	0.051
data-intensive	75:86	1:6	0.066	0.104	0.085	0.127

We conduct our study based on the following assumptions regarding all three components of the TCO model. Firstly, datacenter costs include capital and operational expenses. Capital expenses represent the cost for designing and building a datacenter. This cost depends on datacenter power capacity, and it is expressed as price per Watt. We use a default value of 15 $/W as in [56]. Operational expenses represent the cost for maintenance and security, and depend on datacenter size which, in turn, is proportional to its power capacity. We use a default value of 0.04 $/kWmonth [56]. Secondly, for server costs, beside the equipment itself, there are operational expenses related to maintenance. These expenses are expressed as overhead per Watt per year. We use the default value of 5% for both types of servers. Moreover, for building a real datacenter, the business may take loan. The model includes the interest rate for such a loan. We use a value of 8% per year, although for building a datacenter with emerging ARM systems this rate may be higher due to potential risk associated with this emerging server platform. Thirdly, electricity expenses are modeled based on the average power consumption. In addition, the overhead costs, such as those for cooling, are expressed based on the Power Usage Effectiveness (PUE) of the servers. For the employed Xeon servers, we use the lowest PUE value of 1.1 representing the most energy-efficient Google servers [56]. For ARM servers, we use a higher PUE of 1.5 to incorporate less energy-efficient power supply and the power drawn by the fan, which is up to 1.5 W and represents ∼10% of the 16 W peak power.

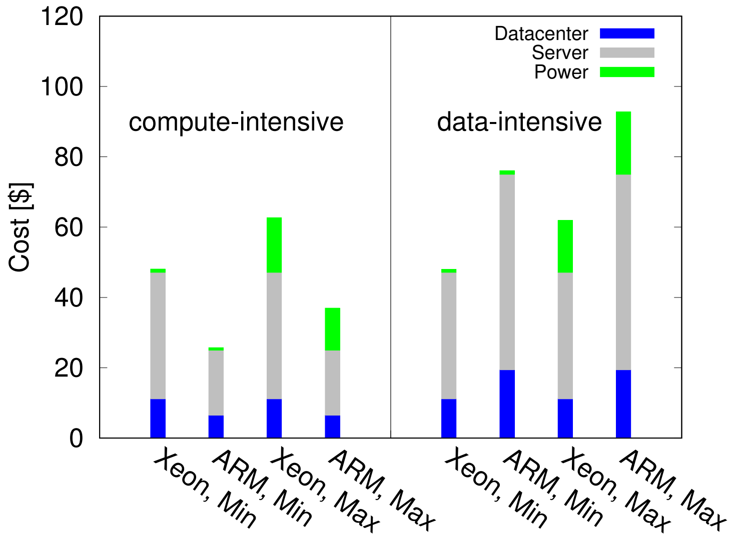


Figure B.7: Costs per month

In Figure B.7 we present TCO values for high utilization scenario. We show these values as break-down of monthly cost into datacenter, server equipment and power costs, as defined in Equation B.3. The cost is dominated by equipment expenses. For data-intensive workloads, equipment and power expenses of the six ARM nodes make low-power servers more expensive than traditional Xeon. We summarize TCO values for both utilization scenarios in Table B.5.

B.5 Summary

In this section, we have presented a performance study of executing Big Data analytics on emerging low-power nodes in comparison with traditional server nodes. Using clusters of Odroid XU boards representing high-end ARM big.LITTLE architecture, and Intel Xeon systems as representative of traditional server nodes, we have evaluated the time, energy and cost performance of well-known MapReduce workloads exercising CPU cores, memory and I/O in different proportion. The results show that there is no *one size fits all* rule for the efficiency of the two types of server nodes. However, small memory size, low memory and I/O bandwidth, and software immaturity concur in canceling the lower-power advantage of ARM nodes. For compute-intensive Pi estimator implemented in Java, a software-emulated instruction results in ten times slower execution time on ARM. Implementing this workload in C++ improves the execution time by a factor of five, leading to almost four times cheaper data analytics on ARM servers compared to Xeon. For data-intensive workloads, such as Terasort, six ARM nodes are required to perform the work of one Xeon node with 75% utilization. This substitution leads to 50% higher TCO of ARM servers. In future, with the development of 64-bit ARM server systems having bigger memory and faster I/O, and with software improvements, ARM-based servers are well positioned to become a serious contender for traditional Intel/AMD server systems.

Appendix C

MoSS Programming Example

In this section, we present MoSS API and an example of implementing Grep, a well-known MapReduce application, in MoSS. MoSS API is listed in Table C.1 being divided in three parts, (i) general usage functions, (ii) string manipulation functions and (iii) functions for conversion between string and numerical types. General usage functions are used to initialize GPU kernel processing, to retrieve <*key, value*> pairs, to emit results and to handle additional data needed for processing. These additional data are required by some applications in the Map phase. For example, Grep requires a string representing the regular expression to be matched against each line of the input. These additional data structures can be passed to GPU kernels using `gpuAllocExtra()` and `gpuAllocCopyExtra()` functions.

Function	Description
`void gpuInit(TaskContextGPU* ctx)`	initializes Map/Reduce context data structures on GPU
`void gpuGetKey(TaskContextGPU* ctx, char** key)`	returns the key from the input <*key,val*> pair
`void gpuGetValue(TaskContextGPU* ctx, char** val)`	returns the value from the input <*key,val*> pair
`void gpuEmit(TaskContextGPU* ctx, char* key, char* val)`	outputs a <*key,val*> pair
`int gpuIdleThread(TaskContextGPU* ctx)`	returns true if calling thread is out of worker threads range
`void gpuAllocExtra(TaskContextGPU* ctxHost, TaskContextGPU* ctxDev, size_t size)`	allocates GPU memory needed by Map/Reduce CUDA kernels
`void gpuAllocCopyExtra(TaskContextGPU* ctxHost, TaskContextGPU* ctxDev, void* src, size_t size)`	allocates GPU memory and copied data needed by Map/Reduce CUDA kernels
`SettingsPipesGPU* getSettingsPipesGPU()`	get MoSS Pipes settings
`void gpuStrCpy(char* src, char* dst)`	copies string `src` into `dst`
`void gpuStrCpyLen(char* src, char* dst, int* len)`	copies string `src` into `dst` and puts the length in `len`
`char* gpuStrTok(char** pstr, char delim)`	returns next substring delimited by `delim`
`int gpuStrSearch(char* str, char* substr)`	searches for a substring in a string
`void gpuStrToInt(char* str, s32Int* n)`	converts string to 32-bit integer
`void gpuStrToLong(char* str, s64Int* n)`	converts string to 64-bit integer
`void gpuStrToFloat(char* str, float* z)`	converts string to single precision floating point
`void gpuStrToDouble(char* str, double* z)`	converts string to double precision floating point
`int gpuIntToStr(s32Int n, char* str)`	converts 32-bit integer to string and returns string length
`int gpuIntToHexStr(s32Int n, char* str)`	represents 32-bit integer in hexa and returns string length
`int gpuLongToStr(s64Int n, char* str)`	converts 64-bit integer to string and returns string length
`int gpuFloatToStr(float z, char* str, int ndec)`	converts single precision floating point to string using `ndec` decimals and returns string length
`int gpuDoubleToStr(double z, char* str, int ndec)`	converts double precision floating point to string using `ndec` decimals and returns string length

Table C.1: MoSS API

Listing C.1: Grep application in MoSS

```
#include "hadoop/Pipes.hh"
#include "hadoop/Pipes.cu"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

using namespace std;


#ifndef MaxCudaBlocks
#define MaxCudaBlocks           1
#endif

#ifndef MaxCudaThreadsBlock
#define MaxCudaThreadsBlock     256
#endif

#define MaxLine         8192

#define MaxRegex        8

#define GrepMapKey      "match"

#define GrepRegexKey    "mapred.mapper.regex"

// map kernel on the GPU
__global__ void mapKernel(HadoopPipes::MapContextGPU* context) {
        char* line;
        char exp[MaxRegex];

        HadoopPipes::gpuInit(context);
        if (HadoopPipes::gpuIdleThread(context))
                return;

        HadoopPipes::gpuGetValue(context, \&line);

        HadoopPipes::gpuStrCpy((char*)context->extra, exp);

        if (HadoopPipes::gpuStrSearch(line, exp))
                HadoopPipes::gpuEmit(context, exp, "1");
}

class GrepMapper : public HadoopPipes::Mapper {
public:
        string exp;

        GrepMapper( HadoopPipes::TaskContext& context ) {
                HadoopPipes::JobConf* conf =
                        (HadoopPipes::JobConf*)context.getJobConf();
                if (conf != NULL) {
                        exp = conf->get(GrepRegexKey);
                }
                else {
                        exp = "";
                }
#ifdef F_MAP_GPU
                gpuInit(context.getMapContextGPUHost(),
                        context.getMapContextGPUDevice());
#endif
        }

        // map function on the CPU
        void map(HadoopPipes::MapContext& context) {
                string line = context.getInputValue();
                int pos = line.find(exp);
                if (pos >= 0 && pos < line.size())
```

```
                        context.emit(exp, "1");
        }

        // initialize GPU regex
        void gpuInit(HadoopPipes::MapContextGPU* ctxHst,
                     HadoopPipes::MapContextGPU* ctxDev) {
              HadoopPipes::gpuAllocCopyExtra(ctxHst, ctxDev, (void*)exp.c_str(),
              exp.length() + 1);
        }

        // map function on the GPU
        void gpuMap(HadoopPipes::MapContextGPU* context,
                    int gpuBlocks, int gpuThreads) {
                mapKernel<<<gpuBlocks, gpuThreads>>>(context);
        }
};

// . . .

int main(int argc, char *argv[]) {
        HadoopPipes::SettingsPipesGPU* settings =
                HadoopPipes::getSettingsPipesGPU();
        settings->cudaBlocks = MaxCudaBlocks;
        settings->cudaThreadsBlock = MaxCudaThreadsBlock;
        settings->cudaMaxInBuff = MaxLine;
        settings->cudaMaxOutBuff = MaxLine;

#ifdef F_MAP_GPU
#ifdef F_MAP_CPU
        settings->flagMapCPU = 1;
        settings->flagMapGPU = 1;
#else
        settings->flagMapCPU = 0;
        settings->flagMapGPU = 1;
#endif
#else
        settings->flagMapCPU = 1;
        settings->flagMapGPU = 0;
#endif

        return HadoopPipes::runTask(
                HadoopPipes::TemplateFactory<GrepMapper, GrepReducer >());
}
```

Listing C.1 presents MoSS code for Grep[1]. In addition to Hadoop Pipes C++ code, developer has to (i) define a GPU Map kernel, (ii) write handling code inside Mapper class and (iii) set MoSS Pipes parameters in `main()` function. Grep GPU kernel uses MoSS API and has the same functionality as its CPU counterpart. This kernel is called from `gpuMap()` function inside Mapper class. Moreover, Grep needs an additional data structure inside Map kernel. This additional data structure is a string representing the regular expression to be matched against each line of the input. The string is passed to GPU kernel through `gpuAllocCopyExtra()`

[1]Specific MoSS code is highlighted using lightgreen background.

function called in Mapper class constructor. In the end, the developer sets MoSS execution parameters such as GPU thread count, input/output buffer size and whether the CPU, GPU or both are to be used for processing.

In summary, MoSS provides an expressive API to handle data-parallel processing on heterogeneous systems with GPU which is seamlessly integrated with Hadoop.