HETMR: Hybrid Deployment of MapReduce Jobs on Heterogeneous Hardware

Alexandros Koliousis,[†] Paolo Costa,[†‡] Peter Pietzuch[†] and Alexander Wolf[†]
[†]*Imperial College London* [‡]*Microsoft Research Cambridge*

Abstract

Cloud providers offer access to hardware accelerators, such as FPGAs and GPUs, according to a pay-per-use model. Data-parallel processing frameworks, such as MapReduce, make it easy for users to express parallel jobs, but an open challenge remains how such jobs can exploit accelerators in a cloud setting. While dedicated MapReduce frameworks for specific accelerators exist, users lack decision support on when accelerators can deliver substantial benefit for their jobs.

We observe that the performance improvement attained by accelerators, if any, is dependent on both the *data* and the *phase* (map or reduce) of the MapReduce job. Based on a generalised model of shared-memory MapReduce, we describe HETMR, a MapReduce framework that can execute jobs in a *hybrid* fashion across CPUs and accelerators. While previous attempts at a hybrid solution simply allocate portions of job data to full MapReduce implementations residing on the CPU and accelerator, HETMR instead assigns a phase to each device and provides a means to move the data between the two. To avoid unnecessary accelerator costs and determine the best phase assignment, HETMR first profiles the MapReduce job on the CPU, yielding a measure of potential speedup. Depending on the outcome, it then deploys the job in either a CPU-only or hybrid fashion by drawing on a library of designs. Our experimental evaluation shows that HETMR's hybrid execution model improves job completion by up to 6 times with an FPGA.

1 Introduction

The difficulty of scaling to ever higher clock frequencies means that recently proposed computer architectures are increasingly heterogeneous, incorporating general-purpose graphics processing units (GPGPUs) and field-programmable gate arrays (FPGAs) besides general-purpose multi-core CPUs. To stay competitive, cloud providers have begun to offer tenants access to specialised hardware accelerators such sa GPGPUs and FPGAs. For example, Amazon provides access to cluster GPU instances as part of its EC2 compute cloud [1]; Maxeler enables tenants to share expensive FPGA-based dataflow engines through their MaxCloud [2].

When cloud offerings include hardware accelerators, tenants are typically charged according to a per-pay use model. Due to their higher cost, tenants pay a premium for hardware accelerators: while Amazon only charges a slightly higher cost for virtual machine (VM) instances that include commodity GPGPUs, more specialised accelerators such as FPGAs command a substantially higher price [2]. This raises the challenge for cloud tenants to decide *when* and *how* to employ more expensive accelerators for their cloud-based compute jobs.

Data-parallel processing frameworks such as MapReduce [12] and Spark [31] have made it easy to express parallelism for compute-intensive jobs. A developer only needs to provide *map* and *reduce* functions, while the framework transparently manages data parallelism, data movement and failure recovery. As a result, they have been adopted as programming models for shared-memory deployments of compute-intensive jobs on multi-core CPUs [29, 27], FPGAs [23] and GPGPUs [16, 15].

We observe, supported by empirical evidence in Section 5, that not all MapReduce jobs benefit from execution on hardware accelerators. This means that tenants of a heterogeneous cloud with accelerators have to make a trade off between the higher cost of using accelerators and the reduction in job completion time that this may incur. In addition, we show that some MapReduce job achieve the fastest completion time when executing in a *hybrid* fashion across a CPU and an accelerator: e.g. by executing the map function on an FPGA and a the reduce function on the CPU, it is possible to leverage their respective strengths.

We describe **HETMR**, a system that executes MapReduce job in a hybrid fashion, utilising the resources of a multicore CPU and an FPGA accelerator. Based on an online profiling step, HETMR decides if a MapReduce job has the potential to benefit from FPGA accelerating by considering the cost of data movement across the PCIe bus. Based on the specific bottleneck that the MapReduce job has, HETMR selects either an accelerated map or reduce kernel from a library of FPGA designs. It then executes the MapReduce job in a hybrid fashion, managing the data movement between the CPU and the FPGA in order to avoid performance degrading pipeline stalls.

Our library of FPGA kernel for MapReduce applications leverages a range of implementation techniques that increase the performance of data movement between the CPU and FPGA. In addition, our FPGA designs avoid memory conflicts and exploit an implementation for data-parallel combiners on the FPGA.

We evaluate the benefit of hybrid MapReduce execution across a CPU and FPGA using a prototype implementation of HETMR. Our results shows that a range of MapReduce jobs can benefit from hybrid execution, thus reducing the costs to tenants when deploying jobs in heterogeneous clouds.

The remainder of this paper is organised as follows. Section 2 gives background on accelerators in data centres, MapReduce implementations on different platforms and sample applications. In Section 3, we discuss the challenges when making decisions about the hybrid execution of MapReduce jobs, and propose a simple deployment strategy for such application in heterogeneous environments. Section 4 describes the architecture and implementation of our HETMR system. We present experimental evaluation results in Section 5. The paper finishes with a discussion of related work (Section 6 and conclusions (Section 7).

2 Background

Although co-processors are intended to speed up expensive computations otherwise carried out on the host CPU (which is why they are often referred to as *accelerators*), programmers typically find new ways to leverage them beyond their initial purpose. The most recent example is the GPU, which began as a floating-point co-processor and has evolved into a general-purpose, parallel thread-processing engine. With the development of CUDA and related technologies, the GPU is now a convenient programming platform for a wide variety of tasks. The FPGA, which began as a platform for experimenting with new processor designs, has evolved into a legitimate co-processor platform in its own right. For instance, the investment company JP Morgan uses FPGAs to accelerate the execution of complex financial models, attaining a reported 30-fold performance increase at 6% reduced power consumption per node [28].

The use of GPUs and FPGAs is growing at the same time as the CPU itself is evolving to include increasing numbers of cores. Rather than competing, however, these trends are complementary. A multi-core CPU can provide improved performance across a set of coarse-grained tasks (typically whole applications) by running those tasks in parallel, one on each core, rather than interleaved or sequentially. At the level of an individual task, the CPU core uses mechanisms such as out-of-order execution, speculation, and branch prediction to mask latency and improve single-thread performance. However, provision of these mechanisms reduces the silicon area available for data-parallel computations. The GPU and FPGA, on the other hand, are platforms designed to improve throughput by exploiting high degrees of data parallelism. They do so, however, by sacrificing the latency of any given thread. For example, the clock speed of an FPGA is orders of magnitude slower than that of a CPU, yet can deliver higher throughput for certain jobs. This distinction between *latency-* and *throughput-oriented* compute platforms is subtle and can lead programmers to make inappropriate design choices [14].

Accelerators in data centres. In the context of a cloud data centre, GPUs and FPGAs provide a throughput-oriented alternative to the standard latency-oriented CPU compute resource. Consider that with CPUs, the only way to increase the throughput of a data-parallel job is to allocate more cores, typically spanning multiple CPUs; this is the standard practice for today's large-scale MapReduce jobs. With a throughput-oriented platform, the possibility exists to attain not only increased throughput, but also with fewer resources. Recognising this, Amazon was famously an early adopter of hosted GPUs,[1] but is now being joined by companies such as Nimbix,[2] Peer1,[3] Penguin[4] and CASS.[5] FPGAs are also beginning to migrate into cloud data centres, with Maxeler's MaxCloud[6] a prime example.

Outfitting a data centre with CPUs, GPUs and FPGAs presents challenges to both operators and users. One is *cost*: The commodity GPUs that are sold with commodity CPUs (e.g. for game acceleration) are insufficient as general-purpose hosting platforms in a data centre. NVIDIA and AMD are responding by offering a new generation of high-end, and therefore more costly, GPU co-processors specifically targeting the cloud data centre (the Tesla and FirePro series, respectively). FPGAs are even more expensive in absolute terms. These increased vendor costs are, of course, passed on to the client through higher usage charges as compared to stan-

[1]`http://aws.amazon.com/hpc-applications`
[2]`http://www.nimbix.net/`
[3]`http://www.peer1.com`
[4]`http://www.penguincomputing.com/`
[5]`http://www.cass-hpc.com/solutions/hoopoe`
[6]`http://www.maxeler.com/products/maxcloud/`

dalone CPUs.

A second challenge is *heterogeneity*: The presence of radically different compute platforms leads to complexity in the deployment and management of applications. In particular, not all jobs can be accelerated by an accelerator and, indeed, may exhibit degraded performance if deployed on the wrong platform. Given the higher cost structures of using an accelerator in a data centre, the client would like to know up front which platform will give them the best performance. They can then combine this with a cost analysis to make a good decision.

MapReduce on different platforms. As our focus here is on MapReduce applications, we need to understand the impact of the different compute platforms on the performance of MapReduce jobs. In this paper we are not particularly concerned with how best to program a MapReduce application on a CPU, GPU or FPGA, which is itself an important area of study. Rather, we want to understand how best to make a *deployment decision* assuming good implementations are available for the alternative platforms.

Classical MapReduce is structured as a *map phase* and a *reduce phase*. In the map phase, input data are divided into multiple pieces, called *shards*, that are processed by *mappers*. A mapper usually consumes more than one shard and emits one of more *key/value pairs*. In the reduce phase the data are also processed in multiple pieces, but on a per key basis. Between the map and reduce phases, the *shuffle phase* groups and sorts the intermediate data emitted by the mappers and then presents those data to the reducers. As an optimisation, *combiners* can in some cases be attached to mappers to perform localised data reduction, in particular when the reduction function is associative and commutative. This serves to reduce the volume of intermediate data that must be moved during the shuffle phase.

We make two critical observations (Section 3). First, assuming the input data reside on the host CPU DRAM, the effective throughput of an accelerator tasked with running a MapReduce job is limited by the speed of the PCI express (PCIe) bus that connects it to the host CPU. (The situation is similar for Infiniband or Ethernet connections to an accelerator.) For example, data are streamed to the NVIDIA GPU via a 6GB/s PCIe bus and to the Virtex FPGA via a 2GB/s PCIe bus. Therefore, the speed of the PCIe bus not only dictates the computational throughput of the accelerator, but also the potential speed up attained over a CPU. Second, the throughput of a MapReduce application can be bottlenecked by either the map or the reduce computation. Therefore, one phase might turn out to be a significantly better candidate for acceleration than the other. Both data movement and computational load turn out to be important factors in the deployment decision (Section 3.3).

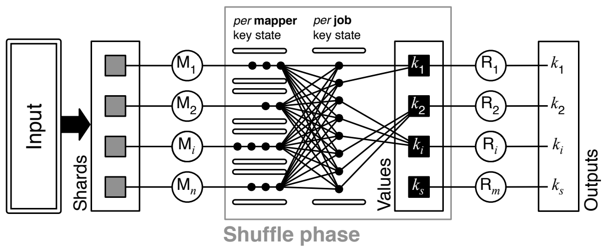


Figure 1: Shared-memory MapReduce

Shared-memory MapReduce. In classical MapReduce, input data are first moved to the mappers, then intermediate data moved between the mappers and the reducers, and finally the result data moved back toward the client. These data movements in a typical implementation of MapReduce involve expensive file system reads and writes.

A significant performance improvement can be attained by instead taking advantage of large main memories to hold the intermediate data of the shuffle phase. In fact, the shuffle phase becomes nearly trivial in such a *shared-memory MapReduce*, where a hash table can be used to store intermediate results by key, thereby avoiding the need to sort the data (Figure 1). Of course, this assumes that the intermediate data fit within the available memory. Again, combiners can be used to help reduce those data.

Phoenix++ [27], which targets symmetric, multithreaded CPUs, is the latest in a series of shared-memory MapReduce systems [21, 29]. Compared to its predecessors, Phoenix++ improves mainly on cache memory locality during the shuffle phase. The CPU-based Phoenix++ system has been closely tracked by corresponding implementations on GPUs. MapCG-shared [8] is one of the latest systems attempting to place a general MapReduce framework inside a GPU. In line with Phoenix++, this system advocates the use of combiners and dedicates a large part of shared memory to their use. Although it yields better performance in comparison to its GPU predecessors [15, 16], it has poor performance in comparison to CPU-based Phoenix++. For example, the Histogram application is 150 times slower on MapCG-shared than on Phoenix++. The reason is simple: although the GPU provides fast access to cache memory, the threads compete for that access, negatively impacting data parallelism.

Example applications. Our work is informed and evaluated by looking at several example applications and a range of jobs (datasets) for those applications. The different datasets are created by varying some important parameter controlling the nature of the dataset. Abstractly, the applications fall into two classes: machine learning and correlation. For machine learning we use *logistic regression* (LR), while for correlation we use *string match*

(StM) and *similarity* (Sim). The correlation applications are further specialised into whether they use the *Levenshtein* (-Lev) or the *Smith-Waterman* (-SW) distance algorithms. This yields five concrete applications.

The datasets for the two classes of applications are quite different. For machine learning, the input is N d-dimensional points, where we vary d from 2^2 to 2^{26}. In order to keep the datasets of the same size we also vary N from 2^{26} to 2^2. Thus, the product of $N \times d$ is consistently 2^{28}. A point size of 4 bytes then results in a fixed input size of 1GB. The dataset for correlation is drawn from a Wikipedia repository. In the case of StM, the specific application is to correlate the Wikipedia article titles with a given set of words, while in the case of Sim, it is to find a correlation in the article titles written by each author (i.e., a distance between the titles). The data-dependent variable for both StM and Sim is the distribution of article title lengths (binned into 10 percentiles). In StM we make the simplifying assumption that the application programmer "knows" the set of words that will need to be matched so that we can restrict the data-dependent variable to the Wikipedia articles themselves.

3 CPU-only vs. Hybrid Deployments

As pointed out in the previous section, it is critical to consider which phase of a shared-memory MapReduce job might most benefit from acceleration, since the bottleneck will appear either in the map or in the reduce. Moreover, the throughput of the PCIe bus between the CPU and the accelerator can counter any advantage that the accelerator might provide in executing a phase, since it acts as a cap on throughput. Overall, using an accelerator does not necessarily lead to an improvement in the performance of a MapReduce job. Therefore, one must be careful in choosing the most appropriate deployment.

In this section we first substantiate our observations about MapReduce phase bottlenecks and cost of data movement across the PCIe bus. From that, we then delineate a simple procedure for deciding how best to deploy a particular job, either CPU-only or hybrid. We evaluate this procedure in Section 5.

3.1 Bottlenecks in MapReduce

The execution time of any given MapReduce job is usually dominated by a particular phase, thereby acting as a throughput bottleneck. A common assumption is that the bottleneck phase is consistently the map. However, this is not the case. In particular, the presence or absence of combiners—essentially a design decision made by the programmer of a MapReduce application—turns out to strongly determine the bottleneck phase. Consider that when a job uses combiners, all key values have already been reduced by the second stage of data movement, thus essentially rendering the reduce function to a noop. In contrast, the absence of combiners leaves intact lists of intermediate values per key and, therefore, requires the reducers to expend greater computational effort.

More specifically, the presence of combiners shifts the computational cost of jobs to mappers relative to reducers because the number of partitions (data parallelism cardinality) of jobs with combiners equals the number of input chunks. In other words, the parallelism strategy is established *before* the map phase: the input data set is partitioned into chunks, chunks are processed in parallel and then the results of each computation are aggregated. A large number of machine learning applications follow this pattern, including k-means clustering, linear and logistic regression, neural networks and principal component analysis [10]. The same property holds for arithmetic applications such as fast multiplication [26].

On the other hand, the absence of combiners shifts the computational cost of jobs to reducers relative to mappers, because the data parallelism cardinality of jobs without combiners is determined by partitioning intermediate key/value pairs based on some relation of the input data. In other words, the parallelism strategy is established *after* the map phase. The role of mappers is simply to ensure that intermediate key/value pairs are grouped accordingly. For example, graph algorithms are parallelised, in the map phase, by splitting the input graph into subgraphs to be processed in the reduce phase [18, 25]. Similarly, in database joins, mappers group together pairs of values for reducers [3]. Such reduce functions typically have computations that are exceedingly difficult to parallelise.

To summarise, we recognise two broad classes: *map-intensive* applications, consisting of a computationally expensive map function, an associative and commutative combiner function, and a trivial (noop) reducer; and *reduce-intensive* applications, consisting of a computationally inexpensive map function whose main purpose is to balance the load among reducers (usually by means of some universal hash function), a buffer combiner, and a computational expensive reducer (typically polynomial in time complexity).

Figure 2 illustrates the bottleneck effect by showing the relative amount of time spent in the map and reduce phases for each of the five example MapReduce applications. Each data point represents a job. The size of the point is related to the total amount of time. We can see that StM in both its variants is highly map-intensive, since their mappers were designed to include combiners that render the reducer irrelevant. Both Sim jobs, on the other hand, do not have combiners, which means relatively more computation takes place in the reducers. Furthermore, we can see that as the jobs become more complex (i.e., their run times increase), they also become

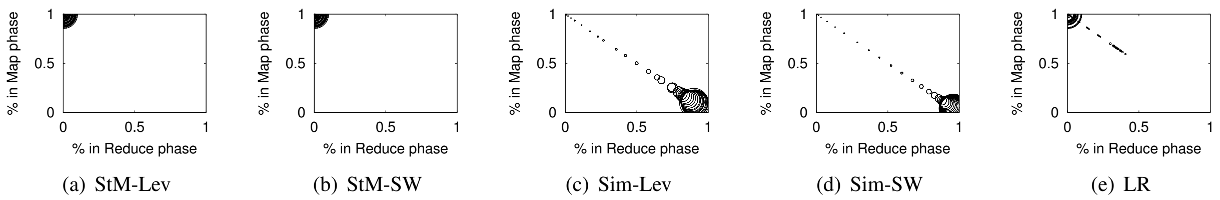


Figure 2: Bifurcation of Map/Reduce applications: map-intensive or reduce-intensive

more reduce-intensive. Finally, LR is shown to be a map-intensive application.

3.2 Data movement cost

When deploying MapReduce jobs in a hybrid fashion across a CPU and an FPGA, it is important to account for the data movement costs beween the CPU and the accelerator. Typically, this is limited by the maximum throughput supported by the PCIe bus, which interconnects the CPU memory with the FPGA on-board DRAM memory. The PCIe bus bottleneck means that a MapReduce job deployed in a hybrid fashion theoretically cannot achieve a throughput that is higher than 1.3 GB/s.

In practice, as we show as part of our experimental evaluation in Section 5, the maximum achievable data movement rate between the CPU and FPGA is lower than the PCIe throughput limit due to *data transformation cost*: when data is exchanged after the map phase ran on the CPU, the data sent over the PCIe bus must be transformed and laid out in a way that is compatible with the data ingestion pipelines of the FPGA reduce kernel (see Section 4.3). This transformation incurs a computational cost, and it may also increase the amount of intermediate data to be transferred over the PCIe bus.

3.3 Deployment procedure

Next we describe a deployment procedure that, by profiling the data movement requirements of a MapReduce job, can make a decision about the potential reduction in job completion time that a given MapReduce job may experience as part of a hybrid deployment.

The essence of the procedure involves two steps: *profiling* a sample and *selecting* an implementation based on the profile. We assume a scenario in which CPU-only and hybrid implementations of an application are placed into a library managed by a cloud data centre platform. A customer comes to the data centre with a particular job to be run, which amounts to presenting the platform with a dataset. The customer wishes to maximise end-to-end throughput, but minimise their costs, so the hybrid implementation and its associated higher costs should be used only if there is an acceleration benefit to be gained.

Step 1: Profiling on a sample. We use an instrumented version of the CPU-only shared-memory implementation of the application to *predict* which implementation will provide the given job with a higher end-to-end throughput. The instrumentation measures the throughput attained by the CPU-only implementation at each data movement point: insertion into the map phase, extraction from the map phase, insertion into the reduce phase and extraction from the reduce phase. Critical to the utility of this prediction is that we can obtain sufficient information by running the profile on only a relatively *small sample* of the input dataset. The sample must, of course, be representative of the dataset. We make the assumption that the customer is able to provide such a sample. In our evaluation in Section 5 we use the trivial sampling technique of taking a small prefix of the input dataset.

Step 2: Selecting an implementation. The metrics collected in Step 1 will tell us whether and where the job would cause the PCIe throughput cap to be exceeded. The hybrid implementation will induce a particular need for the PCIe to move data into and out of the bottleneck phase. Taking the metrics together with the implementation requirements, we can determine if the hybrid implementation would respect the PCIe cap for that job. If it does not, then we choose the CPU-only implementation. Notice that if both map-intensive and reduce-intensive hybrid implementations are available, we are given more freedom in making a selection.

4 HetMR Design

In this section we describe the design of the HetMR system, which executes a MapReduce job across a CPU and an FPGA based on the outcome of the decision procedure (Section 4.1). We give an overview of the architecture of the HetMR system (Section 4.2). After that, we describe some of the implementation details of map and reduce phases on an FPGA for different classes of jobs, which permit HetMR to achieve superior performance across a range of applications.

4.1 Overview

The HetMR system implements a hybrid shared-memory MapReduce model that can execute (i) both the map and reduce phases on a multi-core CPU; (ii) execute the map phase on the CPU and the reduce phase on an FPGA; or (iii) execute the map phase on the FGPA and the reduce phase on the CPU.

As part of its architecture, HetMR must manage the execution of map and reduce phases and the data move-

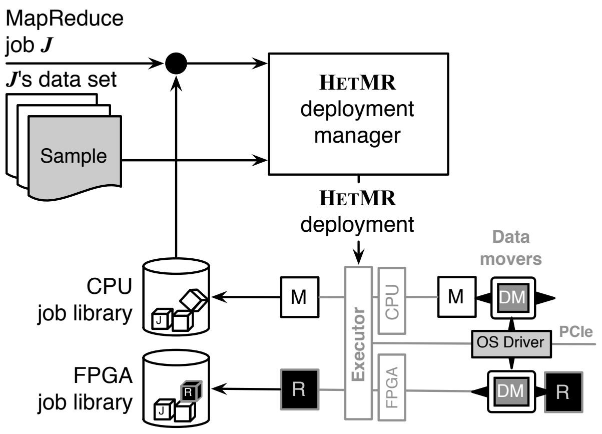


Figure 3: Overview of the HETMR architecture

ment between them:

Execution environment. HETMR must offer an execution environment for map and reduce functions both on the CPU and the FPGA. While this is trivial to achieve for a CPU, which can execute functions implemented in a general-purpose programming language according to a shared-memory MapReduce model, it is challenging for an FPGA, which is not programmable in a general-purpose fashion. Instead, designs for FPGA kernel implementations for the map and reduce phases of different applications must be prepared ahead of job deployment time.

Therefore, we adopt an approach in which HETMR has access to a *library* of FPGA designs for different MapReduce applications, which can be executed on demand. The FPGA designs focus on either the map or reduce phase for a given application based on the bottlenecks specific to that application (Section 3.1).

Data movement. Under hybrid execution of a MapReduce job, the HETMR system must manage the movement of data between the map and reduce phases. It therefore includes run-time functionality that moves data from the CPU memory to the FPGA memory, and back. It also manages the data movement from the FPGA memory to the FPGA on-chip memory, and back.

The challenge is to intercept the MapReduce data flow at the right stage of the computation in order to avoid stalling either the CPU or FPGA resources. In addition, the moved data must be laid out within a contiguous memory region and aligned in a way that enables data parallelism across multiple FPGA pipelines. When data is moved from the FPGA back to CPU memory, it should be laid out in such a way that the CPU can resume processing using its multiple threads.

4.2 Architecture

As shown in Figure 3, the HETMR design includes five main components: (i) the *deployment manager* makes a decision about how to execute a given job; (ii) the *CPU executor* runs map and reduce tasks on a multi-core CPU; (iii) the *FPGA executor* can initiate the execution of job-specific map or reduce kernels, which are selected from (iv) the *FPGA job library*; and (v) the *CPU-FPGA data mover* manages the movement of data between the CPU memory and the FPGA memory over a PCIe bus.

4.2.1 Deployment manager

The deployment manager implements the decision procedure (Section 3.3) to determine how a given job can benefit from FPGA acceleration by deciding to deploy a given phase on the FPGA. As described in Section 3, the particular phase deployed to the FPGA is determined by: (i) which phase exhibits the job's throughput bottleneck and (ii) whether the required data movement can be achieved within the throughput limit of the CPU/FPGA bus.

The deployment manager receives a MapReduce job specification and, based on its decision, passes the individual map and reduce tasks to the CPU and FPGA executors for execution.

4.2.2 CPU executor

The CPU executor provides an execution environment for map and reduce tasks according to a shared-memory MapReduce model (Section 2):

Threads. It manages a set of *execution threads*, which are permanently assigned to individual CPU cores. Each thread processes a queue of map or reduce tasks that is populated after the input data have been partitioned. Threads process either map or reduce tasks until their task queue becomes empty, and there are no tasks to steal from other threads.

Dataflow. Similar to previous shared-memory MapReduce frameworks such as Phoenix++, the CPU executor uses a two-stage *data movement* that groups intermediate key/value pairs in memory. There are two main data structures that hold intermediate key/value pairs: *local state* and *shared state*. Local state is replicated *n* times, each populated by one of the *n* threads that execute map tasks. Shared state is accessible by all threads.

In the first stage of data movement, map tasks accumulate intermediate key/value pairs as part of their local state. After they have finished, they copy the data into a shared memory space. The executor maintains pre-allocated memory slots per key and per thread so that synchronisation overheads between threads are minimised. In the second stage of data movement, threads execute reduce tasks that merge together the results of each map task according to the key.

Map. During the map phase, each thread executing a map task maintains a key-indexed dictionary and either (a) *accumulates* key/value pairs by means of a *combiner function* or (b) *buffers* key/value pairs by means of concatenating intermediate values. The first data movement above occurs when there are no more input partitions to process, at which point map tasks copy their private state into shared state.

Reduce. After all map tasks have finished, the reduce phase begins. Each reduce task collects all values associated with a given key, and copies them to a list—the input to the reduce function. During this data movement, it is also possible to accumulate values with combiner functions. The shared state is large enough to hold the output of all map tasks. Reduce tasks do not compete to access shared state because it is accessed per key.

4.2.3 FPGA executor

The FPGA executor is responsible to executing the map or reduce phase of an application on the FPGA. It obtains appropriate designs for job-specific map and reduce task implementations from the FPGA job library. Once it has received a given task design, it loads the design into the FPGA. It then invokes the CPU-FPGA data mover to initiate the transfer of data into the deployed FPGA kernel.

After data movement, the input data reside in FPGA memory. The FPGA executor sets the number of cycles execute on the FPGA based on the input size and the degree of parallelism on-chip. It also sets any static job-specific parameters, such as the number of dimensions in the input data, in registers or block RAM (BRAM). The executor interfaces with the FPGA kernel through two memory controllers for input and output, respectively. It initialises the memory controllers to the start address so that the deployed FPGA kernel can read or write data. Finally it signals the kernel to start synchronous data transfer, and it awaits an interrupt that signals that the last output byte has been written to memory by the FPGA kernel.

4.2.4 FPGA job library

The FGPA job library contains a set of FPGA for application-specific FPGA kernel designs for map and reduce tasks that can be accelerated. Internally, each *computation kernel* is complemented by two *memory kernels* that handle the data movement from DRAM to the kernel and back. These memory kernels are the same across designs; different designs simply configure them with different parameters.

A kernel can have more than one input and output stream. With typical FPGA hardware, there is a limited number of streams available between the memory and the computation kernel (e.g. up to 16 streams in total).

4.2.5 CPU-FPGA data mover

The CPU-FPGA data mover manages the data transfer from the CPU memory to the FPGA memory and vice versa. It handles the transcoding of data into correct data formats, the memory layout of the data, and initiates data transfers to ensure that FPGA kernels that execute map or reduce tasks have required input data.

The data mover supports data represented in a range of data structures, including lists, arrays and vectors, that can be used to represent input or intermediate key/value pairs as used by the CPU executor. In addition, it also manages the contiguous CPU memory region in which data can reside. When it is passed a pointer by the CPU or FPGA executors, it can transfer a specific number of bytes between memory types. It assumes that the data in the FPGA memory have already been aligned and laid out appropriately by the FPGA executor.

4.3 FPGA implementation

Next we provide details on some of the FPGA implementation challenges when implementing map or reduce phases on the FPGA in order to interface them with CPU execution without a performance penalty.

First, we describe our solution to provide fast read memory access through an optimised data layout, which leverages the knowledge of the application semantics to deal with potential memory conflicts arising during write operations (Section 4.3.1). After that, we discuss our implementation of data-parallel combiner functions on the FGPA (Section 4.3.2).

4.3.1 Memory access

The key challenge related to the FPGA kernel designs of map and reduce tasks is to ensure that the hardware pipelines are not stalled on memory accesses (both for reads and writes). Due to the low clock frequency of the FPGA, pipeline stalls drastically reduce performance, possibly outweighing the benefits of running a map or reduce phase on the FPGA in the first place.

Optimised data layout for memory reads. Depending on the nature of the phase being executed on the FPGA, data can be read either from the main CPU memory, through the PCIe bus, or directly from the on-board FPGA memory through the DRAM bus. Efficient memory access crucially determines performance. For example, many machine-learning jobs require multiple iterations during the map phase and so intermediate data are stored inn the on-board memory; some applications, such as word count, only require a single pass and, hence, data are only read through the PCIe bus.

The PCIe bus offers a much lower throughput compared to the DRAM bus. On our hardware set up (Section 5), it exhibits a peak performance of 16 bytes per cycle, corresponding to 1.49 GB/s; in contrast, the DRAM bus has a peak throughput of 38.4 Gbps (i.e. 384 bytes/-cycle).

This difference in memory access throughput introduces a challenge: an unoptimised data layout can severely reduce the rate at which data can be read. For example, consider a reduce phase implementing an edit distance function (e.g. the Levenshtein or Smith-

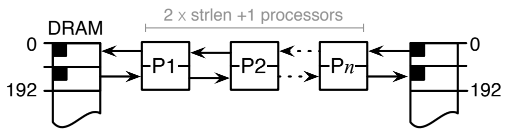


Figure 4: String distance calculator as a systolic array of processors P1...Pn where n is a function of the string length (strlen). Each processor in the pipeline reads and writes 2 bytes per cycle, one from left and one from right. The pipeline can be replicated 96 times.

Waterman distance), in which two text strings are compared character-by-character. A natural way to implement this comparison would be to use a *systolic architecture* [19] such as the one depicted in Figure 4. However, this constrains the read throughput to only 2 bytes/cycle (i.e. 0.19 GB/s) because of the loop dependencies: the second character of a string is valuable to the computation only after the first has been processed.

Based on our experience implementing many MapReduce applications, we identify two main recurring patterns to optimise the data layout and overcome the above problem:

Circular memory access. Machine-learning applications in MapReduce are typically written in an iterative fashion where data are read from the on-board memory at the beginning of each iteration. To efficiently support this, we have built a memory controller optimised for circular memory access. This memory controller has an extra counter that resets the read address to the start address of the section of the data that has to be iterated.

Systolic array. To solve the issue of loop dependencies when dealing with a systolic array implementation, we utilise the following layout: rather then storing the strings sequentially, we rearrange them in memory so that the first 192 bytes correspond to the first character of 96 string pairs, the next 192 bytes to the second character, and so on. This allows the FPGA executor to feed 96 arrays in parallel with 2 bytes/cycle each, thus achieving the maximum throughput available.

Avoiding memory write conflicts. Another potential cause of low FPGA processing throughput is the occurrence of memory access conflicts between concurrent pipelines on the FPGA. This event typically predominates during the map phase when intermediate key/value pairs are written into an array structure, realised through registers or with on-chip BRAM. If data cannot be written immediately, the FPGA pipeline stalls and throughput decreases.

We implement two different approaches to address this issue, thus reducing the impact of memory conflicts. The first is *application-specific* and exploits the knowledge of the application semantics: if all FPGA pipelines generate the same key set—which would generate a large number of conflicts in a naive design—we extend the map kernel to also include a data-parallel combiner function that aggregates all keys. This removes the need to store each individual key/value pair in memory, thus avoiding the occurrence of write conflicts.

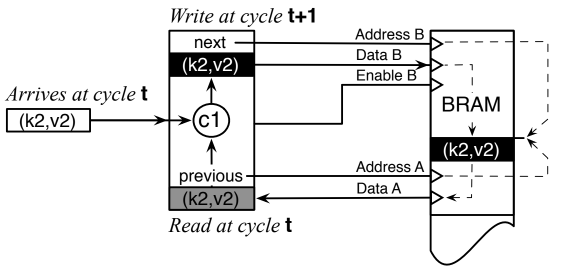


Figure 5: A $k2$ combiner object using a dual-port BRAM. Port A is READ_ONLY and port B is WRITE_ONLY.

In general, however, it is hard to predict memory access usage. In this case, we adopt an *application-agnostic* approach that dedicates a separate DRAM write stream to each pipeline. This approach, however, is limited by the hardware to 16 streams and excess use of resources for buffering between them.

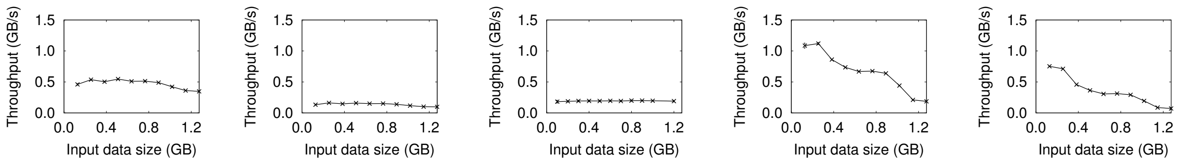
4.3.2 Data-parallel combiners

When a given MapReduce application supports combiner functions, the job performance can be improved using a data-parallel design of combiner functions on the FPGA. In addition, such a class of applications is also more likely to result in a design that fits in its entirety on the FPGA.

In its simplest form, an FPGA pipeline for a combiner lags behind a map pipeline by only a few cycles, assuming that all keys fit in cache memory. Given a newly emitted key/value pair $(k2, v2)$, a combiner requires at least one cycle to read the previous value associated with key $k2$ and increment it with $v2$, and at least one cycle to write the accumulated value back to the same physical memory location. Figure 5 shows one such combiner implemented with a dual-port BRAM.

We assume that all keys can be read in parallel in the same cycle because they fit in cache memory. Similarly, in the next cycle, all values can also be updated in parallel. However, when multiple FPGA pipelines request access to the same physical memory address in the same cycle, we are required to multiplex both the address bus (Address A, Figure 5) and the data bus (Data B).

If this multiplexing is known in advance (e.g. if all FPGA pipelines generate the same key set), it is straightforward to attach one such data-parallel combiner to the map kernel, thus fitting the entire job on-chip. Otherwise, we must deal with memory access conflicts by partitioning the key space of intermediate data.



(a) StM-Lev (b) StM-SW (c) LR (d) Sim-Lev (e) Sim-SW

Figure 6: CPU throughput against sample data size for different MapReduce jobs

Application	Abbrv.	Size (GB)
Logistic regression	LR	1
Similarity (Levenstein)	Sim-Lev	1.3
Similarity (Smith-Waterman)	Sim-SW	1.3
String match (Levenstein)	StM-Lev	1.3
String match (Smith-Waterman)	StM-SW	1.3

Table 1: Datasets

5 Evaluation

Our evaluation consist of two parts. First, we want to understand the impact of the sample when profiling a job. Next, we want to verify that our deployment manager correctly selects the fastest deployment option. Our result show that when the sample is representative of the full data set, our deployment manager *always* makes the right decision. Further, they indicate that if the job execution time is dominated by the map, the sample size is largely irrelevant, while it becomes critical for reduce-intensive job.

These result confirm the feasibility and effectiveness of our approach but, at the same time, it stresses the importance for the user to properly select the sample.

5.1 Experimental setup

We performed all the CPU runs (including profiling) of our experiments using Phoenix++ on a Maxeler's 1U MPC-C series node [2]. This comprises two Intel Xeon 5650 chips, with six hyper-threaded cores and 12 MB L3 cache per chip; totalling 24 execution threads. The CPU is connected to a Xilinx Virtex-6 FPGA board via the PCIe bus.

In our experiments, we consider a set of representative MapReduce jobs, detailed in Table 6(a). Logistic Regression and String match represent map-intensive jobs; the Similarity application represents reduce-intensive jobs. Logistic regression is an ML job which is strongly structured in terms of both inputs and outputs, while String match and Similarity have only structured outputs. As input data for the latter two, we use a snapshot of 1.274 GB of Wikipedia articles. For logistic regression, instead, we randomly generate the numbers using a uniform distribution

We run each experiment 10 times and we plot the average of the results. We do not show error bars in the chart as the standard deviation across run was always below 5%.

.

5.2 On stability and sample size

Figure 6 shows the impact of the sample data size on the CPU throughput. For jobs in Figure 6(a)- 6(c), the throughput obtained is largely independent from the sample size. This is important because it means that small samples can be used, which drastically reduces the profiling time. Based on our results, a small sample size of 100 MB seems sufficient to obtain a good estimation of the expected CPU throughput when running the entire job. Even assuming a conservative throughput of 100 Mbps, this would lead to a profiling time shorter than 10 s. We consider this overhead negligible in practice.

Sim-Lev (Figure 6(d)) and Sim-SW (Figure 6(e)), instead, show a different behaviour as throughput sharply decreases with larger sample size. The reason is that the first three jobs are *map-intensive* while these other two are *reduce-intensive*. For the first category, the size of the data to move to the FPGA is identical to the input data of the job. Conversely, for the reduce phase, the size of the data moved to the FPGA depends on the output of the map phase. This explains why we observe such variability.

5.3 HETMR decisions

Next, we want to assess the correctness of the decision taken by the deployment manager. In Figure 7, we show the throughput achieved by our map-intensive jobs when running with the sample data set size fixed; but, to model different input data, on the x axis we vary a job-specific (data-dependent) parameter. For example, for Logistic Regression we vary both the number of dimensions per point and number of points simultaneously, keeping the input size fixed at 1GB. For the Wikipedia data set, we sample from the distribution of title lengths (based on percentiles).

The goal of these charts is twofold. First, they confirm that while size is not a critical parameter of the input size, the *type* of data has a direct impact on the throughput achieved by the job. This is why HETMR cannot use static information but it has to rely on the profiling stage to derive the expected throughput.

The second important results shown in the charts is that our deployment manager always makes the correct decision. Interestingly, just looking at the PCIe through-

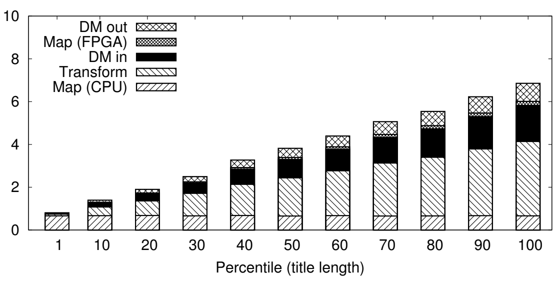


Figure 8: Map phase breakdown for StM-Lev.

put would have not given the correct results in some cases, e.g., in the StM-Lev

The reason is that, as explained in §4.3.1, in order for the job to efficiently run on the accelerator, the input data must be pre-processed to change the data layout in memory. In this specific case, it turns out that this pre-processing stage was actually dominating the execution time, thus outweighing the expected benefits.

This is clearly visible in Figure 8 in which we show the breakdown of the execution time. When the input data size increases, the cost of changing the data layout becomes more prominent and for large input this becomes the main bottleneck.

6 Related Work

In this section, we discuss prior related work to the HETMR approach.

Heterogenous MapReduce. As CPU heterogeneity emerged in today's computing environments, so did performance imbalances in data-parallel processing frameworks. In this context, one of the most widely studied frameworks in the literature is MapReduce [12]—not only in shared-nothing clusters, but also within the confines of a single shared-memory machine. In both these environments, the proposed solutions to tackle performance imbalances are similar: faster processors "steal" tasks from slower ones, albeit their associated limitations manifest at different architecture levels (e.g. data movement overheads shift from network and disk I/O to slow memory inter-connects and access conflicts in cache memory).

Tarazu [4] and its extension, Pikachu [13], are two recent MapReduce schedulers that aim to balance the load between "slow" and "fast" processors in a cluster. This loose distinction between processors is realised as a ratio of the rate at which each processor type consumes data chunks (termed *progress rate*; see LATE [30]). This ratio is estimated at run-time, after a job phase starts on both processor types. It is then used to partition (reshuffle) the input and intermediate data accordingly. These systems are agnostic of any differences in the micro-architecture between slow and fast processors and operate solely on two performance indicators: their progress rate and CPU utilisation.

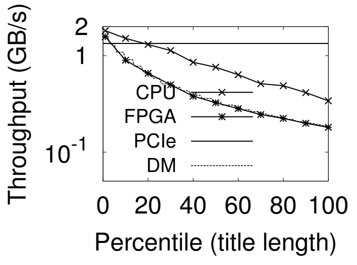
HETMR is complementary to this line of work in two ways: it pinpoints the loose terms "slow" and "fast" on specific hardware architectures on a per job basis; and, by doing so, it makes data movement overheads, those incurred during reshuffle, explicit.

Mate-CG [17] works under the same assumption as Tarazu and Pikachu—that faster nodes should process a larger fraction of the data than slower nodes—but it considers a cluster of GPGPU-enhanced nodes. The system further assumes that accelerators are throughput-oriented processors. As such, they should operate on large chunks of data to be effective. Therefore, their data partitioning policy is biased towards accelerators. Mate-CG deals only with iterative MapReduce jobs so that it can reshuffle data chunks at every iteration. Furthermore, it assumes that GPGPU implementations do not suffer from performance penalties inherent to that architecture (e.g. code branching or memory access conflicts).

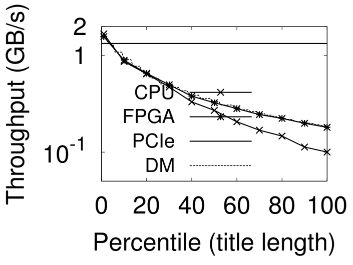
A recent study [6] suggests that many, if not all, MapReduce jobs fit well within the confines of a single shared-memory machine: there is enough thread parallelism and memory to support today's workloads. Earlier on, a series of MapReduce systems (led by Phoenix [21, 29, 27]) studied how intermediate key/value pairs should be laid out in cache memory in symmetric multi-core systems. In parallel, Mars [15], MapCG [16], MapCG-shared [8, 9] have also attempted to wall in MapReduce within a GPGPU. The problem with those GPGPU-only solutions is that the higher they move in the memory hierarchy, the more severe the performance penalties become, mainly due to memory access conflicts and synchronisation between shared and main memory. Our proposed alternative is a hybrid execution model.

Closely related to our notion of hybrid execution is MapCG-shared [9]. It proposes a map-dividing scheme (data partitioning) and a pipelining scheme when different phases run on different architectures. The latter is essentially the streaming model that CUDA proposes. However, their model is confined only to jobs with combiners—ignoring jobs without combiners. They also do not decide when to use either a hybrid or CPU solution. One of the main feature of our hybrid approach is that it allows for different hardware architecture designs to be combined within MapReduce.

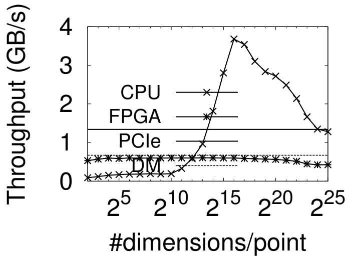
FPMR [23] is a MapReduce framework specifically designed for FPGAs. It does not consider a hybrid solution but demonstrates that jobs can actually fit in the FPGA. One of the main features of FPMR is a common data path to share read-only job parameters between pipelines. This is indeed complementary to our work in some cases, because shared state has to be replicated and wired independently to each pipeline. There is also the case, however, that one of the streams can be dedicated



(a) StM-Lev



(b) StM-SW



(c) LR

Figure 7: CPU throughput against FPGA throughput. The PCI line identifies the PCIe throughput while DM is the data movement upper bound (see § 3.2)

to input parameters, as we do with the logistic regression application.

Beyond MapReduce models. GPGPU- or FPGA-based frameworks have attempted to maintain the simplicity of the MapReduce programming model, but they have not made explicit the performance penalty due to memory management. HETMR provides mechanisms to move data between the CPU and its accelerators and, for some cases, move data within an accelerator.

Dandelion [22], LINQits [11], Accelerator [24], Liquid Metal [7], and PetaBricks [20, 5] promise increased performance and reduced energy consumption in mainstream heterogeneous computing. Part of this promise can be attributed to code generation and compiler optimisation techniques that deal with the intricacies of different hardware architectures internally. Orthogonal to this line of our work on HETMR for FPGAs because it imposes structure on how to handle key/value pairs both on- and off-chip.

7 Conclusions

This paper has delved into the nature of map and reduce computations and has demonstrated that care must be taken when deciding if it is appropriate to use an accelerator to improve performance. We have presented a system for supporting the decision process and deploying MapReduce jobs in a hybrid fashion involving both a CPU and an accelerator.

The decision procedure can be improved in various ways to account for more specifics of applications and accelerators. One direction is to account more precisely for the computational load of map and reduce functions. Another is to consider a third possible deployment, which we refer to as *packed*, wherein both the map and reduce are placed on the accelerator when the cost of doing so is low.

References

[1] High Performance Computing on Amazon Web Services. `http://aws.amazon.com/hpc-applications/`.

[2] Maxeler Technologies, MPC-C Series. `http://www.maxeler.com/products/mpc-cseries/`.

[3] AFRATI, F. N., SARMA, A. D., MENESTRINA, D., PARAMESWARAN, A., AND ULLMAN, J. D. Fuzzy Joins Using MapReduce. In *Proceedings of the 28th IEEE Int'l Conference on Data Engineering (ICDE)* (2012).

[4] AHMAD, F., CHAKRADHAR, S. T., RAGHUNATHAN, A., AND VIJAYKUMAR, T. N. Tarazu: optimizing MapReduce on heterogeneous clusters. In *Proceedings of the 17th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[5] ANSEL, J., CHAN, C., WONG, Y. L., OLSZEWSKI, M., ZHAO, Q., EDELMAN, A., AND AMARASINGHE, S. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).

[6] APPUSWAMY, R., GKANTSIDIS, C., NARAYANAN, D., HODSON, O., AND ROWSTRON, A. Scale-up vs Scale-out for Hadoop: Time to rethink? In *SoCC* (2013).

[7] BACON, D., RABBAH, R., AND SHUKLA, S. FPGA Programming for the Masses. *Queue 11*, 2 (Feb. 2013), 40–52.

[8] CHEN, L., AND AGRAWAL, G. Optimizing MapReduce for GPUs with effective shared memory usage. In *Proceedings of the 21st Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[9] CHEN, L., HUO, X., AND AGRAWAL, G. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (2012).

[10] CHU, C.-T., KIM, S. K., LIN, Y.-A., YU, Y., BRADSKI, G., NG, A. Y., AND OLUKOTUN, K. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. MIT Press, Cambridge, MA, 2007, pp. 281–288.

[11] CHUNG, E. S., DAVIS, J. D., AND LEE, J. Linqits: Big data on little clients. *SIGARCH Comput. Archit. News 41*, 3 (June 2013), 261–272.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[13] GANDHI, R., XIE, D., AND HU, Y. C. PIKACHU: How to Rebalance Load in Optimizing Mapreduce on Heterogeneous Clusters. In *Proceedings of the USENIX*

Annual Technical Conference (ATC) (2013).

[14] GARLAND, M., AND KIRK, D. B. Understanding throughput-oriented architectures. *Communications of the ACM 53*, 11 (2010), 58–66.

[15] HE, B., FANG, W., LUO, Q., GOVINDARAJU, N. K., AND WANG, T. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)* (2008).

[16] HONG, C., CHEN, D., CHEN, W., ZHENG, W., AND LIN, H. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th Int'l Conference on Parallel architectures and compilation techniques (PACT)* (2010).

[17] JIANG, W., AND AGRAWAL, G. MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters. In *Proceedings of the IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)* (2012).

[18] KARLOFF, H., SURI, S., AND VASSILVITSKII, S. A model of computation for mapreduce. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010).

[19] LIPTON, R. J., AND LOPRESTI, D. A Systolic Array for Rapid String Comparison. In *Proceedings of the Chapel Hill Conference on VLSI* (1985).

[20] PHOTHILIMTHANA, P. M., ANSEL, J., RAGAN-KELLEY, J., AND AMARASINGHE, S. Portable performance on heterogeneous architectures. In *Proceedings of the 18th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[21] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th Int'l Symposium on High Performance Computer Architecture (HPCA)* (2007).

[22] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)* (2013).

[23] SHAN, Y., WANG, B., YAN, J., WANG, Y., XU, N., AND YANG, H. FPMR: MapReduce framework on FPGA. In *Proceedings of the 18th ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays (FPGA)* (2010).

[24] SINGH, S. Computing without processors. *Commun. ACM 54*, 8 (Aug. 2011), 46–54.

[25] SURI, S., AND VASSILVITSKII, S. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th Int'l Conference on World Wide Web (WWW)* (2011).

[26] SZE, T.-W. The two quadrillionth bit of pi is 0! distributed computation of pi with apache hadoop. In *CloudCom* (2010), IEEE, pp. 727–732.

[27] TALBOT, J., YOO, R. M., AND KOZYRAKIS, C. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the 2nd Int'l Workshop on MapReduce and its Applications* (2011).

[28] WESTON, S., MARIN, J.-T., SPOONER, J., PELL, O., AND MENCER, O. Accelerating the Computation of Portfolios of Tranched Credit Derivatives. In *Proceedings of the IEEE Workshop on High Performance Computational Finance (WHPCF)* (2010).

[29] YOO, R. M., ROMANO, A., AND KOZYRAKIS, C. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the IEEE Int'l Symposium on Workload Characterization (IISWC)* (2009).

[30] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Job Scheduling for Multi-User MapReduce Clusters. Tech. Rep. UCB/EECS-2009-55, University of California, Berkeley, April 2009.

[31] ZAHARIA, M., CHOWDHURY, M., DAS, T., ET AL. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).