Accelerating MapReduce on a Coupled CPU-GPU Architecture

Linchuan Chen Xin Huo Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{chenlinc,huox,agrawal}@cse.ohio-state.edu

***Abstract*—The work presented here is driven by two observations. First, heterogeneous architectures that integrate a CPU and a GPU on the same chip are emerging, and hold much promise for supporting power-efficient and scalable high performance computing. Second, MapReduce has emerged as a suitable framework for simplified parallel application development for many classes of applications, including data mining and machine learning applications that benefit from accelerators.**

This paper focuses on the challenge of scaling a MapReduce application using the CPU and GPU together in an integrated architecture. We develop different methods for dividing the work, which are the *map-dividing scheme*, where map tasks are divided between both devices, and the *pipelining scheme*, which pipelines the *map* and the *reduce* stages on different devices. We develop dynamic work distribution schemes for both the approaches. To achieve high load balance while keeping scheduling costs low, we use a *runtime tuning method* to adjust task block sizes for the *map-dividing scheme*. Our implementation of MapReduce is based on a continuous reduction method, which avoids the memory overheads of storing key-value pairs.

We have evaluated the different design decisions using 5 popular MapReduce applications. For 4 of the applications, our system achieves 1.21 to 2.1 speedup over the better of the CPU-only and GPU-only versions. The speedups over a single CPU core execution range from 3.25 to 28.68. The *runtime tuning method* we have developed achieves very low load imbalance, while keeping scheduling overheads low. Though our current work is specific to MapReduce, many underlying ideas are also applicable towards intra-node acceleration of other applications on integrated CPU-GPU nodes.

I. Introduction

Even though GPUs are a major component of the high performance computing landscape today, computer architectures have already taken another step towards more effective utilization of the GPUs, in the form of *integrated chips*. In these chips, a multi-core CPU and a GPU are integrated in silicon. Both the two devices share the same physical memory, which makes it possible to copy data between devices at high speeds. The AMD Fusion architecture integrates a GPU that is OpenCL programmable on the CPU chip, which has been available for some time. Intel released their latest generation *Ivy Bridge* processor in late April, 2012; this chip also integrates a GPU that supports OpenCL. NVIDIA has announced their on-going project *Denver*, which will integrate their ARM-based CPUs with the NVIDIA GPUs on the same chip. These newly emerged or upcoming heterogeneous architectures will efficiently accelerate computations on PCs or embedded systems.

Parallel programming has always been a hard problem, and programming GPUs is no exception. Though CUDA (and subsequently, OpenCL) have simplified GPU programming, using all processing power of an integrated CPU-GPU is a challenging problem. Even before the emergence of integrated CPU-GPU architectures, there has been some work towards exploiting CPU and GPU together for accelerating a single application [15], [24], [19], though no standard programming systems have emerged.

Specialized or domain specific programming systems can ease programming. For example, the emergence of MapReduce paradigm for parallel programming has coincided with the growing prominence of data-intensive applications [4]. Though initially developed in the context of data processing applications, the MapReduce abstraction has also been found to be suitable for developing a number of applications that perform significant amount of computation (e.g. machine learning and data mining algorithms). These applications can be accelerated using multi-core CPUs, many-core GPUs or other similar heterogeneous computing devices. As a result, there have been several efforts on supporting MapReduce on multi-core systems [18], [25], as well as on GPUs [2], [9], [5], [20], [21], [13]. One of these implementations, MapCG [10], also tried accelerating an MapReduce application using both a CPU and a discrete GPU. However, there were no consistent speedups over a single device, as data transfers between the host and the GPU and frequent kernel launches incur significant overheads.

Based on the previous success of MapReduce framework on clusters as well as many-core and multi-core architectures, one can expect MapReduce to help application development and performance for the emerging fused CPU-GPU architectures as well. However, it is not clear as to how the MapReduce framework should exploit integrated CPU-GPU architectures.

This paper focuses on the design, implementation, and performance evaluation of a MapReduce framework on the integrated architectures, using an AMD Fusion chip as a representative example. We focus on the parallelism within a single chip, though the work can be extended to clusters of such nodes using the same mechanisms as the current cluster-based MapReduce implementations. We consider two

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

different schemes for partitioning the work between CPU cores and GPU cores within a chip. The first scheme is the *map-dividing scheme*, which dynamically distributes the input to each scheduling unit on both devices, and each device conducts *map* and *reduce* simultaneously. The other scheme is the *pipelining scheme*, which runs the *map* and *reduce* stages on different devices, i.e., each device only executes one stage of MapReduce. In our design, we use one processing core on the CPU to be the scheduler, and use each of the remaining CPU cores and each streaming multiprocessor on the GPU as one scheduling unit, in order to achieve flexible load balancing. With the goal of avoiding runtime data transfer between the CPU and the GPU, we put the input data and the scheduling information in the *zero copy buffer*. To support self-adaptive load balancing for the *map-dividing scheme*, our system uses a *runtime tuning method* to adjust the task block size for each scheduling unit. To lower the memory overhead, we use an alternative design of MapReduce based on *continuous reduction*, which pipelines the intermediate key-value pairs from the *map* stage to the *reduce* stage directly. For the *pipelining scheme*, we try both dynamic load balancing and static load balancing based implementations.

We have evaluated our implementations using 5 commonly used MapReduce benchmarks. For each application, we compare different scheduling schemes against a multi-core CPU-only version, a many-core GPU-only version, a sequential program, and a MapReduce version executed on a single CPU core. For four of the five applications we used, the best CPU-GPU version has a speedup of between 1.21 and 2.1 over the better of the CPU-only and the GPU-only versions, showing clear benefits from exploiting both CPU and the GPU. The relative speedups over the use of a single CPU core range between 3.25 and 28.68. We have also tested the influence of the task block size to the load imbalance and the efficiency of the *runtime tuning method*, and shown that our *runtime tuning method* achieves good load balance with negligible overheads.

II. Background

This section provides background information on heterogeneous computing architectures and the MapReduce paradigm.

A. Heterogeneous Computing Architecture

We focus on the AMD Fusion chip, a representative integrated CPU-GPU chip we used to develop our framework.

The Fusion architecture integrates an AMD Opteron CPU and a GPU on the same silicon chip. The processing component of the on-chip GPU is similar to the discrete GPUs, such as the NVIDIA GPUs and the AMD Radeon GPUs. The processing component of the GPU is composed of a certain number of streaming multiprocessors. Each streaming multiprocessor, in turn, contains a set of simple cores that perform in-order processing of the instructions, and use SIMD parallelism. Furthermore, threads running on a streaming multiprocessor are divided into *wavefronts*, which are similar to warps in NVIDIA GPUs. Each wavefront of threads are co-scheduled on the streaming multiprocessor and execute the same instruction in a given lock-step.

The GPU does not have its own device memory. Instead, it has access to the *system memory* with the CPU. Unlike a GPU which is connected to the host through a PCIe bus, the data transfers between memory and both processing units are through the same high speed bus, and are controlled by a unified memory controller. The system memory is however divided into two parts: the *device memory* and the *host memory*. The device memory and host memory are designed for the GPU and the CPU, respectively, though both of them can be accessed by either device. Such accesses are possibly through a type of memory buffer called the *zero copy buffer*.

Starting with the CUDA-based (NVIDIA) GPUs, a *scratch-pad* memory, which is programmable and supports high-speed accesses, has been available private to each streaming multiprocessor. The scratch-pad memory is termed as the *shared memory*. On the Fusion architecture, the size of the shared memory is 32 KB.

In the OpenCL programming standard, the CPU and the GPU can be viewed to have the same logical architecture. Every processing core on the CPU is similar to one streaming multiprocessor on the GPU, and corresponds to one workgroup (thread block) in the OpenCL programming model. Each workgroup contains a number of threads, which has been discussed before. The same OpenCL code can run on both devices without modification.

B. MapReduce

MapReduce [4] was proposed by Google for application development on data-centers with thousands of computing nodes. It can be viewed as a middleware system that enables easy development of applications that process vast amounts of data on large clusters. Through a simple interface of two *user-defined* functions, *map* and *reduce*, this model facilitates parallel implementations of many real-world tasks, ranging from data processing for search engine support to machine learning [3], [7].

The two user-defined functions, *map* and *reduce*, play the following roles. The *map* function takes a set of input instances and generates a set of corresponding intermediate output key-value pairs. The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the *reduce* function. The *reduce* function accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per *reduce* invocation.

The main benefits of this model are in its simplicity and robustness. MapReduce allows programmers to write functional style code that is easily parallelized and scheduled. Though MapReduce was developed for large data centers, it has been shown to be effective for a number of class of applications, including data mining and machine learning applications, which benefit from accelerators like GPUs. Thus, there has been considerable work towards supporting MapReduce on modern GPUs [2], [9], [10], [13], [21]. Since the emerging coupled CPU-GPU architectures are a natural evolution of GPUs, it is clearly desirable to develop optimized MapReduce implementations for these architectures.

III. System Design

This section describes the main challenges involved, the design decisions we made, and the scheduling schemes we developed.

A. Challenges and Our Approach

There are several challenges that need to be addressed to utilize both the CPU and the GPU on a coupled architecture.

Memory Requirements of the MapReduce Framework: The execution of a MapReduce application tends to generate a large number of key-value pairs. These key-value pairs not only take up considerable memory, but also need frequent read/write operations to the device memory, leading to a low compute-to-device-memory-access ratio. Furthermore, shuffling the intermediate key-value pairs requires frequent synchronization and data movement.

Efficiently Utilizing the Memory Hierarchy on the GPU: Modern GPU architectures incorporate a memory hierarchy. At the top of the memory hierarchy is a small but fast *shared memory*, which is private to each streaming multiprocessor. Shared memory supports high-speed access and fast atomic operations. However, it is challenging to utilize it for the large number of key-value pairs that most MapReduce applications produce.

Utilizing CPU and GPU for a Single Application: We need to investigate how to effectively utilize both the CPU and the GPU for accelerating a single MapReduce application. Both CPU and GPU have significant computing power, though both of the them are better suited for certain task over others. At the same time, different MapReduce applications spend a different fraction of their time on map and reduce stages.

Task Scheduling across the CPU and the GPU: Dynamic load balancing across the CPU and the GPU is challenging, because of the overheads of certain operations on these architectures.

We now briefly outline how the above challenges were addressed. To lower the memory requirement by the MapReduce framework, we base our design of MapReduce on the distinct *continuous reduction* model. In this design model, an implicit data structure, named *reduction object*, is used. Key-value pairs generated by the *map* stage can be inserted and reduced to the reduction object directly, if the reduction operation in the MapReduce application is associative and commutative. This reduces the memory overhead of storing key-value pairs, and makes it possible to take advantage of the limited-sized shared memory.

To utilize CPU and GPU together for a single application, we propose two different approaches. The first is called *map-dividing* scheme, where map operations are divided between the CPU and the GPU. The second is called the *pipelining scheme*, where one device is used for map operations and the other is used for reduce operations.

To address the problem of dynamic workload scheduling on the CPU and the GPU, we develop novel master-worker load balancing models. We use one thread on the CPU device as the scheduler, which runs outside of the OpenCL kernels. The remaining CPU cores, as well as the streaming multiprocessors on the GPU, are the workers executing the tasks scheduled by the scheduler. Our proposed methods avoids the need of using atomic operations in task scheduling.

We now elaborate on the above ideas.

B. MapReduce based on Continuous Reduction

In the *continuous reduction* based implementation model, the *map* function, which is defined by the users, works in the same way as in the traditional MapReduce. It takes one or more input units and generates one or more key-value pairs. However, each key-value pair is inserted to the *reduction object* at the end of each *map* operation. Thus, every key-value pair is merged to the *output results* immediately after it is generated. A data structure storing these intermediate values of output is referred to as the *reduction object*. Every time a key-value pair arrives, the reduction object locates the index corresponding to the key, and reduces this key-value pair to that index in the reduction object. The reduction object is transparent to the users, who write the same *map* and *reduce* functions as in the original specification.

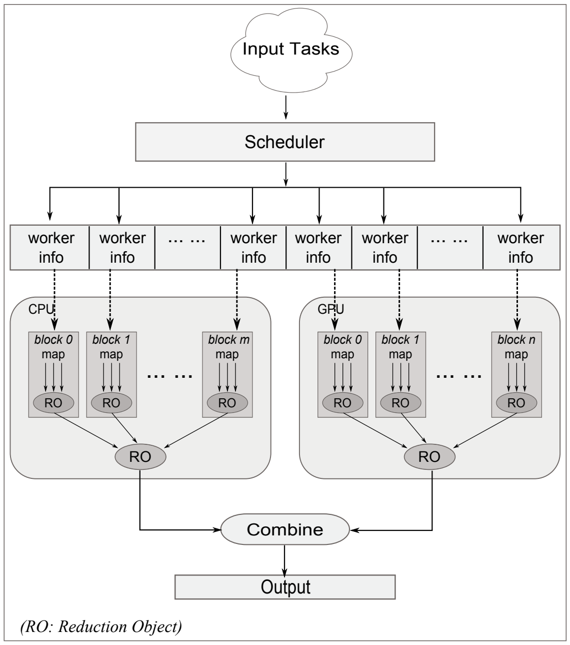
The method described above is based on the assumption that the operation in the *reduce* stage is associative and commutative, which covers most of the MapReduce applications, with some exceptions like a sorting application. If the *reduce* operation is non-associative-and-commutative, all the intermediate key-value pairs still need to be stored in the reduction object, and a shuffling (sort) is needed after all the key-value pairs are generated by the *map* stage. Our framework supports the parallel *in-object sort*, which is based on the bitonic sort [8].

The main advantage of this approach is that the memory overhead of a large number of key-value pairs is avoided since key-value pairs are reduced immediately after they are generated. This, in turn, is likely to allow us to utilize the shared memory on a GPU, especially if the number of distinct keys is not too large. In such case, continuous reductions can be first applied in the shared memory.

However, due to the limited size of shared memory (32 KB on the Fusion architecture) and the possibly large number of distinct keys, we need to address one potential problem of shared memory overflow. For such cases, we have support for *swapping* shared memory reduction objects to the device memory reduction object.

C. Map-Dividing Scheme

The first implementation we created is based on the map-dividing scheme. As shown in Figure 1, all the workers follow the same execution steps. The scheduler dispatches the input tasks to each worker through the worker info array. Every



(RO: Reduction Object)

Fig. 1. *Map-dividing Scheme*

worker corresponds to one workgroup, which in turn consists of a number of working threads. After a task block is scheduled to a workgroup, the workload is evenly distributed to all the threads within that workgroup. Every thread processes the input tasks based on continuous reduction. It first invokes the user-defined *map* function, and generates the key-value pair(s), which are immediately inserted into the reduction object.

Note that besides a device reduction object in each device, we keep a private copy of the reduction object for each worker. This design is based on two considerations. First, for the GPU, each workgroup places its private copy of the reduction object in the shared memory. The shared memory supports faster read/write speed and faster synchronization. Thus the reduction in the shared memory can greatly improve the reduction performance. Second, for the CPU, as well as for the GPU, using private copies can reduce the contention overhead. If all the threads within one device update the same copy of the reduction object, contention on the buckets of the reduction object will be high.

After each worker receives the end message from the scheduler and finishes all its remaining tasks, all the threads belonging to this worker will merge the private reduction object to the device reduction object within each device. The merge of two reduction objects is to retrieve the key-value pairs from one reduction object and insert them to another reduction object in parallel. The merge operation may also be conducted during the process of computation. As we discussed in Section III-B, the size of the shared memory on each streaming multiprocess of the GPU is limited, thus if the private copy of the reduction object is full, overflow handling is needed.

After the end of the computation on each device, the two device memory reduction objects from the CPU and the GPU need to be merged together. The merge process is done in parallel by the GPU, rather than by the CPU, since GPUs are good at processing highly parallel operations. This is efficient for applications with large reduction objects, such as matrix multiplication.

The key challenge in this approach is dividing the map operations between the CPU and GPU cores. The relative speed of operations on the two devices can vary significantly depending upon the application. Therefore, we need dynamic scheduling. However, there are many challenges in supporting an efficient and effective scheme. An intuitive dynamic load balancing method between a CPU and a GPU will involve exiting the kernel and get a task block when a device is idle, as is also used in MapCG [10]. This method is not efficient since each task block assignment requires a kernel launch. A possible way of avoiding frequent kernel launch is to utilize the *zero copy memory buffer*, so that each device can access a *global task offset* from within the kernel. Because different devices access the same data structure in an atomic way, this turns out to be very expensive on current architectures.

Thus, we have developed this new dynamic scheduling method based on the master-worker model. The thread running on the first core of the CPU works as the scheduler. The remaining CPU cores and streaming multiprocessors on the GPU are workers. The entire workload is placed in the zero copy buffer, where data can be accessed by both the scheduler and the workers. The communication between the scheduler and the workers is through a data structure referred to as the *worker info array*, which is also placed a zero copy buffer, since the worker info must be accessed by both the scheduler and the workers from both devices. Task assignments and requests between the scheduler and the workers are sent through the worker info array. Each element in the worker info array is used independently between the scheduler and one worker.

Scheduling Message	Task Info
has_task	$task\_index$
	$task\_size$

TABLE I
WORKER INFO ELEMENT

Each worker info element contains two types of information: the *scheduling message* and the *task info*. As shown in Table I, the scheduling message only includes a flag: has_task, and the task info includes $task\_index$ and $task\_size$. Algorithm 1 shows the major steps of this scheduling scheme. After the execution starts, the scheduler iterates on the has_task flags for all the workers. If the scheduler finds one worker's flag to be 0, it assigns one task block to this worker. The task assignment includes setting the $task\_index$ and $task\_size$ fields in this worker's worker info element. After that, it sets has_task flag to 1 to inform the worker that a new task block has been assigned to it. The first thread (`thread 0`) of each worker (workgroup) keeps on polling the has_task flag in the corresponding worker info element, and other threads just wait until `thread 0` detects a newly scheduled task block. When `thread 0` sees the value of this flag to be 1, it gets the task information, which includes $task\_index$ and $task\_size$, and then sets the value of the flag as 0, notifying the scheduler that a new task block can be scheduled to this worker again. After that, the task block is processed by all the threads of this worker. As we said earlier, the global input is stored in the zero copy buffer. The value of $task\_index$ indicates the starting position of a particular task block in the global input. $task\_size$ represents the size of the task block that it is going to process. The flag has_task is also used to send the end message to each worker. When all the workload has been dispatched, the scheduler sets the value of this flag in each worker info element as -1, notifying the workers to jump out of the kernel after processing their remaining tasks.

Algorithm 1: Task Scheduling in the Map-dividing Scheme

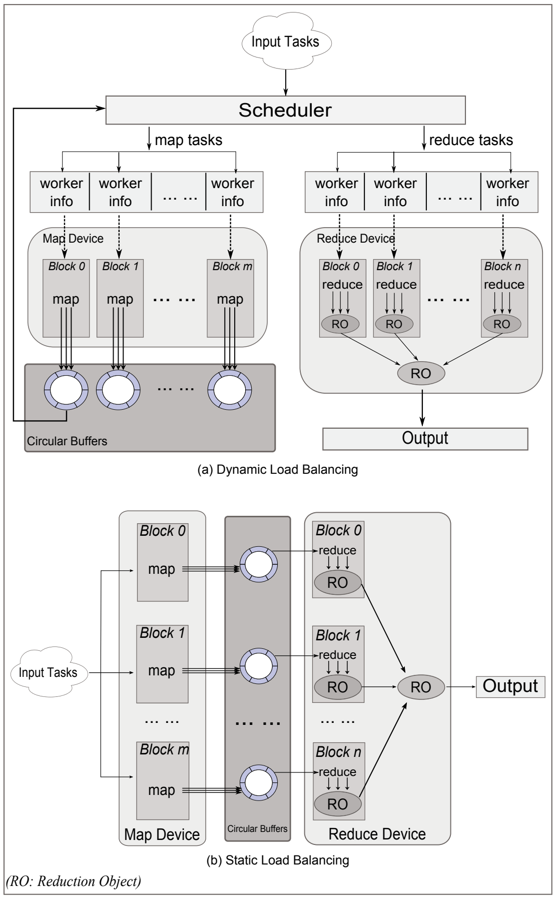
```
$task\_offset \leftarrow 0$;
/*the main steps of the task scheduling*/
while true do
    foreach worker $i$ do
        if $worker\_info[i].has\_task = 0$ then
            /*assign a map task block to map worker $i$*/
            $worker\_info[i].task\_index \leftarrow task\_offset$;
            $worker\_info[i].task\_size \leftarrow TASK\_BLOCK\_SIZE$;
            $worker\_info[i].has\_task \leftarrow 1$;
            $task\_offset += TASK\_BLOCK\_SIZE$;
        end
    end
    /*end the loop when all map tasks are scheduled*/
    if $task\_offset \geq total\_num\_tasks$ then
        break;
    end
end
/*the remaining step of the task scheduling*/
Send end messages to all the workers;
```

D. Pipelining Scheme

Observing the fact that GPUs tend to be better at compute-intensive applications, while CPUs are more effective in handling control flow and data retrieval, we propose another scheme, which we refer to as the *pipelining scheme*. In this scheme, each device is only responsible for doing one stage of the MapReduce, for example, the GPU might perform all *map* operations and the CPU might perform all the *reduce* operations, or vice-versa. However, since the *map* stage involves a larger amount of computation, and the *reduce* stage tends to involve more branch operations and data retrieval, we expect the version where GPU performs map and CPU performs reduce to perform better.

This scheme involves a producer-consumer model. The key-value pairs are produced by the map workers, and are consumed by the reduce workers. Thus, buffers need to be used between the two different kinds of workers. For this scheme, we consider both the dynamic load balancing and the static load balancing.

Subgraph (a) in Figure 2 shows the processing structure of the pipelining scheme with dynamic load balancing. The dynamic load balancing used here is similar to what is used in the *map-dividing scheme*. However, there are several differences also. For each map worker, one *circular buffer* is used. Every circular buffer consists of a number of key-value blocks. Input tasks are only scheduled to the map device. The key-value pairs generated by a map worker are not inserted to the reduction object, but instead, they are stored into one key-value block in the circular buffer. When one key-value block is full, the next empty block in the circular buffer is used. Each key-value block is a reduce task block, and each time one key-value block is scheduled to one reduce worker. All the key-value blocks in all the circular buffers form a universal reduce task pool, and are scheduled by the scheduler to reduce workers dynamically. Since the circular buffers are accessed by both devices, therefore they are placed in zero copy buffers.



(a) Dynamic Load Balancing

(b) Static Load Balancing

(RO: Reduction Object)

Fig. 2. *Pipelining Scheme*

The scheduling procedure is shown in Algorithm 2. In each scheduling cycle, the scheduler not only assigns map tasks to map workers, but also assigns reduce tasks to reduce workers. The map tasks are scheduled in the same way as that in the *map-dividing scheme*. If an idle reduce worker is found, the scheduler will search the key-value blocks in all the circular buffers. The first full key-value block to be found will be scheduled to the idle reduce worker. If no full buffer is found

Algorithm 2: Task Scheduling in the Pipelining Scheme with Dynamic Load Balancing

```
$map\_task\_offset \leftarrow 0$;
$block\_index \leftarrow 0$;
/*the main steps of the task scheduling*/
while true do
    /*schedule map tasks*/
    foreach map worker $i$ do
        if $worker\_info[i].has\_task = 0$ then
            Assign a map task block to map worker $i$;
            $map\_task\_offset + = TASK\_BLOCK\_SIZE$;
        end
    end
    /*schedule reduce tasks*/
    foreach reduce worker $j$ do
        if $worker\_info[j].has\_task = 0$ then
            $i \leftarrow block\_index$;
            foreach key-value block $blocks[i]$ do
                if $blocks[i].full \&\& !blocks[i].scheduled$ then
                    /*assign $blocks[i]$ to worker $j$*/
                    $worker\_info[j].task\_index \leftarrow i$;
                    $worker\_info[j].task\_size \leftarrow blocks[i].size$;
                    $worker\_info[j].has\_task \leftarrow 1$;
                    $blocks[i].scheduled \leftarrow 1$;
                    $block\_index \leftarrow$
                    $(block\_index + 1)\%total\_num\_blocks$;
                    $i \leftarrow block\_index$;
                end
            end
        end
    end
    /*end the loop when all map tasks are scheduled*/
    if $map\_task\_offset \geq total\_num\_tasks$ then
        break;
    end
end
/*the remaining steps of the task scheduling*/
Traverse the key-value blocks, assign full blocks to reduce workers;
Send end messages to all the map workers;
Wait for all map workers to finish execution;
Schedule the key-value blocks which contain data to reduce workers;
Send end messages to all the reduce workers;
```

in one searching iteration, the scheduler will skip this worker and continue with next worker. This prevents the scheduler from spinning on the empty key-value blocks.

Each key-value block not only stores key-value data, but also maintains status information. The $full$ flag is set by the map workers and the reduce workers. Value 1 means this block is full or is ready to be scheduled to a reduce worker, while value 0 means this block is not full and can be used by a map worker to store key-value pairs. Whenever a map worker fills up a key-value block, it sets its $full$ flag to 1, and finds next empty block to use. Whenever a reduce worker consumes a block, it sets its $full$ flag to 0.

The $scheduled$ flag indicates whether a key-value block has already been scheduled to a reduce worker; this flag helps to avoid repeatedly scheduling the same block to different workers. When a reduce worker finishes processing the key-value pairs in a block, it sets its $scheduled$ flag back to 0.

When all the input tasks have been processed, the main scheduling cycle ends. Before exiting the entire scheduling process, the scheduler has some remaining work to do. First, it traverses all the key-value blocks on the map device and assigns full blocks to reduce workers. This step guarantees that the map workers which are still emitting key-value pairs have available key-value blocks to use. Next, the scheduler sends end messages to all the map workers since there are no longer any map tasks to be scheduled. After all the map workers finish execution, the scheduler iterates on all the key-value blocks, and the blocks which contain data are scheduled to idle reduce workers. When all the blocks have been scheduled, the scheduler sends end messages to the reduce workers.

Although dynamic load balancing provides better load balance, it brings scheduling overheads, and furthermore, one CPU core is only used exclusively for scheduling. In the *pipelining scheme*, each device is only responsible for one type of tasks, and all the processors on one device have the same processing speed, thus load imbalance may not be a significant problem. To investigate the possible benefit of avoiding dynamic scheduling, we also exploit the static load balancing to this scheme.

Figure 2 (b) shows the static load balancing for the *pipelining scheme*. We can see that the scheduler is no longer used. All the processors on the CPU participate in computation. Input tasks are evenly and statically distributed to each *map* worker. Like in the dynamic load balancing, each *map* worker has one *circular buffer*, which contains a number of key-value blocks. All the key-value blocks in all the circular buffers form a universal reduce task pool. This time, the key-value blocks are not assigned to the *reduce* workers dynamically, instead, they are evenly mapped to the *reduce* workers. *map* workers find empty key-value blocks to store key-value pairs, and *reduce* workers find full key-value blocks to consume. Synchronization is guaranteed by reading and setting the $full$ flag in each key-value block, in the same way as is described in the dynamic load balancing version. When all the *map* workers finish their jobs, they set the $finish$ flag in each key-value block to be true, notifying the *reduce* workers the end of the execution.

E. Load Imbalance and Runtime Tuning

So far in our discussion, the task block size for the dynamic scheduling in our framework has been assumed to be fixed. In practice, however, determining the appropriate size is a nontrivial problem. As we can expect, a large size can cause load imbalance, whereas a small size can result in high scheduling overheads.

To overcome the difficulty of choosing optimal task block size, which can vary depending upon the application, we have developed a *runtime tuning method*. The method works in the following way: At the beginning, a fixed and relatively small block size is chosen for all the workers. We use these blocks to *probe* the processing speed of each worker. We count the total number of task blocks consumed by each worker, till a certain percentage of the work is performed. After this point, the scheduler adjusts the task block size for each worker by using the profiling information.

Let C_k be the total number of task blocks scheduled on the worker k, and let n be the total number of workers. A large block size is chosen as the $initial\_size$ (200,000 in our implementation), and then the block size for each worker is calculated in the the following way:

$$tuned\_size_k = \frac{C_k}{\left(\sum_{i=1}^{n} C_i\right)/n} \times initial\_size \qquad (1)$$

After this tuning, the job scheduling frequency for each worker will approximate the same, which reduces both the likelihood of load imbalance and the overhead of scheduling. When the scheduling process is reaching the end, the runtime system further reduces the block size of each worker in order to reduce the difference among the finish times of the workers. This idea of reducing the block sizes at the end of the scheduling process is also used in the *pipelining scheme*. Overall, by maintaining large block sizes during the middle of the scheduling, the runtime tuning mechanism reduces the scheduling overhead. By tuning the task block size for each worker according to its computing ability, it improves the load balance among workers.

IV. Experimental Results

We have extensively evaluated our framework with different applications, covering various types of applications for which MapReduce is used. We compare the performance of different approaches we have described in this paper against CPU-only and GPU-only implementations. We also analyze the influence of the task block size on the performance and show the benefits of our runtime tuning method.

A. Experimental Setup and Applications

Our experiments were conducted using an AMD Fusion APU (A8-3850) which integrates a 2.9 GHz quad-core CPU and a HD6550D GPU. The GPU has 5 streaming multiprocessors (SM), and each SM has 80 Radeon cores on it. Thus there are 400 Radeon cores on this GPU, with a clock rate of 600 MHz. Each streaming multiprocessor has a shared memory of size 32 KB. The system memory used in this machine is an 8 GB DDR3 memory with 1600 MHz clock frequency. The system memory is partitioned into a 512 MB device memory and a 7680 MB host memory.

We selected five commonly used MapReduce applications. These applications cover both *reduction-intensive* and *map computation-intensive applications*, where the former represents applications which have a large number of key-value pairs for each unique key, and thus, reduction computation time is significant. In contrast, the latter represents applications that spend most of their time in map stage. Among our applications, K-means clustering (KM) is one of the most popular data mining algorithms [12], which is a reduction-intensive application. Word Count (WC) is our second application, and can also be reduction-intensive if the number of distinct words is small. If the number of distinct words is very large, however, the reduction object cannot be held in shared memory, and then the overflow handling is needed. Naive Bayes Classifier (NBC) is a simple classification algorithm based on observed probabilities. When implemented using MapReduce, the total number of distinct keys is 30, i.e., it is also reduction intensive and has a small reduction object.

Application	*Dataset Size*
K-means, K = 40 (KM)	20,000,000 points
Word Count, 500 Distinct Words (WC)	100 MB text file
Naive Bayes Classifier (NBC)	207 MB vehicle dataset
Matrix Multiplication (MM)	2000*2000 matrices
KNN, K = 40 (kNN)	20,000,000 points

TABLE II
Applications and Datasets Used in our Experiments

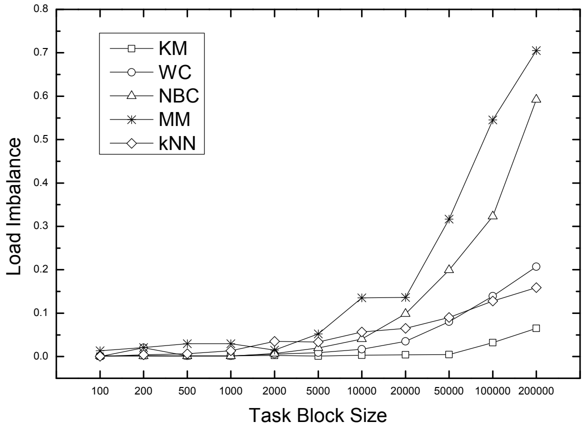


Fig. 3. Load Imbalance under Different Task Block Sizes

The fourth application is Matrix Multiplication (MM), which is a map computation-intensive application. If the two matrices being multiplied are of size $x \times y$ and $y \times\ z$ matrix, then the total number of distinct keys is $x \times z$. Thus, for this application, we use the device memory reduction object only. The *reduce* stage is not needed in this application, and as a result, the pipelining scheme is also not feasible. Our last application is the k-nearest neighbor search (kNN) [1].

The datasets used in the applications are summarized in Table II.

B. Impact of Task Block Size (Map-Dividing Scheme)

To investigate the influence of the task block size to the load imbalance between the CPU and the GPU, we execute each of the applications using the *map-dividing scheme* with different task block sizes. To quantify the load imbalance, we use the following expression:

$$load\_imbalance = \frac{|T_{CPU} - T_{GPU}|}{max\left(T_{CPU}, T_{GPU}\right)} \qquad (2)$$

where, T_{CPU} represents the computation time by the CPU, and T_{GPU} represents the computation time by the GPU.

The change of the load imbalance with the task block size increasing is illustrated in Figure 3. We can see that for the applications we used, the overall trend is that the load imbalance increases as the task block size increases, as we expect. MM has the highest load imbalance among all the applications, because a GPU workgroup is much faster than a

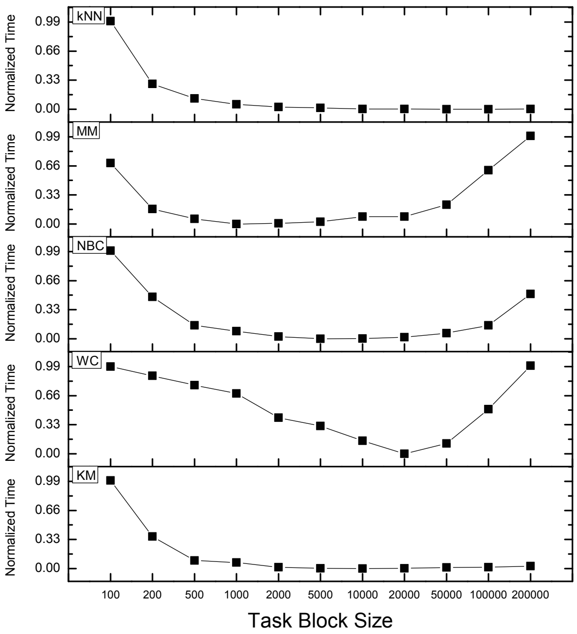


Fig. 4. Computation Time under Different Task Block Sizes

CPU workgroup. KM has the lowest overall load imbalance, as the relative speed difference between the CPU and the GPU is relatively small.

We will also like to note that if our runtime tuning method is used, the load imbalance is below 0.03 for all applications.

Figure 4 shows the normalized computation time of each application under different task block sizes. As we would expect, initially the computation time decreases with increasing task block size, as the scheduling overhead decreases. After a certain point, the computation time increases due to the increase in load imbalance. Different applications have different optimal values for the task block size. The computation times of KM and kNN do not change much from the task block size of 2000. As we indicated earlier, KM is not likely to have a load imbalance, as the processing speeds of the CPU and the GPU are almost identical. WC, NBC and MM favor smaller task block sizes since the GPU processing speed differs very much with the CPU. Thus, a large block size can lead to an idle workgroup.

The *runtime tuning* does help to achieve near optimal performance for each application, which can be seen from Figure 5.

C. Comparison of the Performance from Different Approaches

To see the efficiency of our proposed schemes, we compare the total execution time of each application using different executing schemes. The total execution time includes the computation time, the data copy time, and the combination time. Note that for CPU-only versions, as well as for the versions where *map* tasks are only executed by the CPU, no data copy is required since input data is originally in the host memory. For *map-dividing scheme* versions, because the input is stored in the *zero copy buffer*, which is also located in the host memory, no data copy is involved. Combination time is the time spent on combining the reduction objects from both devices, and does not apply to the pipelining scheme.

The schemes we compare here include:

- **CPU:** CPU-only version.
- **GPU:** GPU-only version.
- **MDO:** *map-dividing scheme* with a manually chosen optimal task block size.
- **TUNED:** *map-dividing scheme* using *runtime tuning method*.
- **GMCR:** *pipelining scheme*, the GPU does the *map* stage and the CPU does the *reduce* stage. We further create two different versions of these, which are **GMCRD** (with dynamic load balancing) and **GMCRS** (with static load balancing).
- **CMGR:** *pipelining scheme*, the CPU does the *map* stage and the GPU does the *reduce* stage. Again, **CMGRD** and **CMGRS** are the versions with dynamic and static partitioning, respectively.

Note that for CPU-only and GPU-only versions, we have implemented both static and dynamic workload partitionings. For each application, we choose the better version to compare with our multi-device versions.

The test results for all the applications are shown in Figure 5. For KM, GMCRS achieves the best performance (2.1 times faster than the CPU-only version), and is even faster than GMCRD. Static load balancing eliminates the scheduling overhead and does not require a CPU processing core to be dedicated to scheduling. Both dynamic and static versions of GMCR are faster than all the other schemes. Although GMCR is a *pipelining scheme*, and therefore incurs I/O overheads to store the intermediate key-value pairs, this scheme works very well for an application like KM, which has a computation heavy *map* stage and a reduction intensive *reduce* stage. This allows effective utilization of GPU cores for floating point calculations in *map* stage, and use of CPU cores for reductions, which involves more control flow and data movement. For KM, the *map-dividing scheme* and its runtime-tuning version also achieve a speedup of 1.31 over the CPU-only version. GMCR is clearly faster than the CMGR, since executing map on CPUs and reductions on GPUs leads to ineffective utilization of both resources.

For WC, the CPU-only version is 2.7 times faster than the GPU-only version, as the hashing process and locating the keys in the reduction object is not executed efficiently on a GPU. The *map-dividing scheme* version has the maximum speedup of 1.24 over the CPU-only version. GMCRS and GMCRD do not perform better than the CPU-only version due to the I/O overhead. Similar as in KM, GMCR performs much better than CMGR, and static load balancing for *pipelining scheme* is more efficient than dynamic load balancing.

The speed gap between the CPU-only and the GPU-only for NBC is also large, i.e., CPU is 4.2 times faster. NBC has a small number of distinct keys (30 in total), which makes the reduction object very small, and leads to high contention

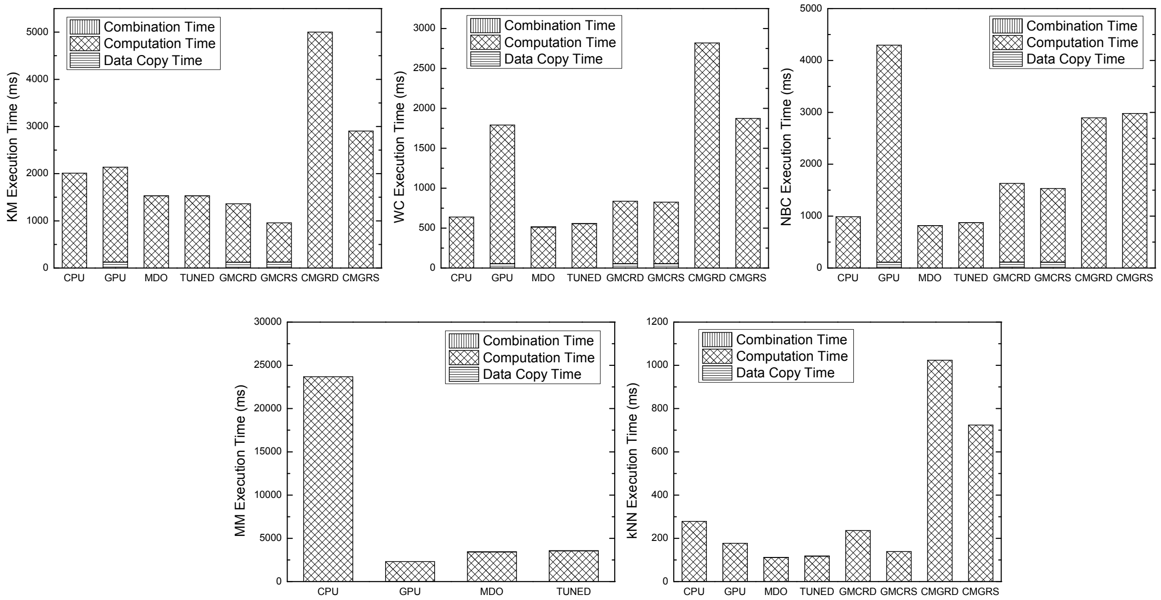


Fig. 5. Execution Time of Different Schemes for Five Applications (KM, WC, NBC, MM, and kNN)

among the threads on the GPU side. The best performance is achieved by the *map-dividing scheme*. MDO version has a speedup of 1.21 and the TUNED version has a speedup of 1.13 over the CPU-only version.

MM turns out to be very different than other applications. First, note that the *pipelining scheme* is not feasible for MM because there is no *reduce* stage in this application. Second, because GPUs are good at processing highly parallel and compute-intensive applications, the CPU-only version takes more than 10 times as long as the GPU-only version. Furthermore, because of the regularity of the application, static load balancing used in the GPU-only version already evenly distributes the tasks to all the workgroups, ensuring high load balance. The dynamic load balancing schemes cannot further improve the load balance. On the contrary, in the map-dividing version, the scheduling overheads and the combination costs slow down the application. Thus, the dynamic scheduling schemes are even slower than the GPU-only version.

We observe that the GPU-only version for kNN is 1.57 times faster than the CPU-only version. This speed difference is relatively small, which makes the combination of the two devices efficient. MDO and its *runtime tuning* version achieve 1.5 speedup over the GPU-only version. GMCRS also has a speedup of 1.27 over the GPU-only version. Similarly, for the *pipelining scheme*, the GMCR versions are faster than the CMGR versions, and the static load balancing is faster than the dynamic load balancing. Note that in this application, the GPU-only and GMCR versions also have their input data in the *zero copy buffer*, thus no data copy time is involved. The reason is that this application involves less computation, and if the input data is copied from the host memory to the device memory, the data copy time would take a large fraction of the total execution time. For other applications, since their data copy time only takes a small percentage, we put their input in the device memory, which supports faster access speeds.

Note that for WC and NBC, the *pipelining scheme* performs worse than the single device versions. This is due to the high I/O overhead incurred by storing the key-value pairs in buffers. For both of these applications, the *map* stage is not very compute-intensive and the total execution time is mainly bound by the *reduce* stage. In the future, if coupled architectures support faster I/O, the pipelining schemes may perform better. Also, for the pipelining scheme, static load balancing is more efficient than dynamic load balancing for most cases, implying the benefit of avoiding the scheduling overhead.

Overall, by integrating both the CPU and the GPU, our proposed scheduling schemes across different devices achieve considerable speedups over the single device versions in most applications. Data copy and the combination times take only a small or modest percentage of the overall execution time (below 15% and 0.5%, respectively). At the same time, analyzing the results for MM, we also note that for applications where the execution times on different devices vary significantly, and who already have very good load balance within each device, it is better to use only the faster device.

D. Overall Speedups from Our Framework

To see the efficiency and the scalability of our framework, we examine the relative and absolute speedups we are able to obtain from the use of the coupled CPU-GPU architecture. For this purpose, we compare the best CPU-GPU version with a *sequential* version (written in C/C++ and executed by a single core of the CPU) and a single CPU core MapReduce version (implemented using our MapReduce framework but executed on a single CPU core). Different versions of each application follow the same logic/algorithm.

	Execution times (ms)				
	KM	WC	NBC	MM	kNN
Sequential	7042	2017	2655	98810	1004
MapReduce (1 core)	7804	2057	2712	93647	1154
Best CPU-GPU (Relative Speedup)	959 (7.34x)	516 (3.91x)	818 (3.25x)	3445 (28.68x)	112 (8.96x)

TABLE III
PARALLEL EFFICIENCY OBTAINED FROM OUR FRAMEWORK

The execution times are shown in Table III. For all the applications, the MapReduce single CPU core version spends no more than 15% extra time compared with the sequential version. Note that for MM, the MapReduce single core version is even slightly faster than the sequential version. This is likely due to the different compiler optimizations used in OpenCL and C.

The speedup of the best CPU-GPU version for each application varies. MM has the maximum speedup, as the GPU is much faster than the CPU for this application. KM has very good load balance between the CPU and the GPU, and thus, integrating the 4 CPU cores and 5 GPU SMs achieves 7.34 speedup over the sequential implementation. In the case of kNN, we also achieve a good load balance and the GPU is faster than the CPU. The overall speedup is 8.96, which is also large. WC and NBC have significant load imbalance between the CPU and the GPU, and the CPU is much faster than the GPU. Thus, the speedups are limited by the parallel efficiency on 4 CPU cores.

V. RELATED WORK

In the past 3-5 years, a lot of efforts have been done to port MapReduce to multi-core CPUs, many-core GPUs, as well as heterogeneous CPU+GPU with decoupled GPUs. To the best of our knowledge, our work is the first to exploit coupled CPU-GPU architectures for MapReduce.

Ranger *et al.* [18] implemented a shared-memory MapReduce library Phoenix for multi-core systems, and Yoo *et al.* [25] optimized Phoenix specifically for large-scale multi-core systems. Mars [9] was the first attempt to harness GPU's power for MapReduce applications. MapCG [10] was a subsequent implementation which was shown to outperform Mars. It also supports scheduling MapReduce across both the CPU and the NVIDIA GPU, which was shown to be inefficient due to the high scheduling overhead between the two devices. In MapCG, a dynamic task queue is maintained, and one kernel launch is required for every task scheduling.

MITHRA [6] was introduced by Farivar *et al.* as an architecture to integrate the Hadoop MapReduce with the power of GPGPUs in the heterogeneous environments, specific for Monte-Carlo simulations. GPMR [21] was a recent project to leverage the power of GPU clusters for large-scale computing by modifying the MapReduce paradigm. Catanzaro *et al.* [2] also built a framework around the MapReduce abstraction to support vector machine training as well as classification on GPUs. StreamMR [5] was a MapReduce framework implemented on AMD GPUs. Shirahata *et al.* [20] have extended Hadoop on GPU-based heterogeneous clusters. They enabled *map* tasks to be scheduled onto both CPUs cores and GPUs devices.

Outside of MapReduce, there have been other efforts on using CPU and GPU simultaneously for a single application. Qilin system [15] has been developed with an adaptable scheme for mapping the computation between CPU and GPU simultaneously. Their approach requires training a model for data distribution based on curve-fitting. Our work, in comparison, is based on dynamic work distribution. Teodoro *et al.* [22] describe a runtime framework that selects either of a CPU core or the GPU for a particular task.

Dynamic task scheduling has been studied for even a longer time. Guided self-scheduling scheme [17] initially allocates larger chunk sizes to reduce scheduling overhead, and subsequently reduces the chunk size towards the end of the computation. Several variations of the guided self-scheduling were developed [11], [16], [23], each with a different strategy for adjusting chunk size. Another effort [14] uses a combination of static and dynamic strategies. Our scheduling policy is specifically designed for a coupled CPU-GPU architecture, and exploits the iterative characteristic of MapReduce applications.

VI. CONCLUSIONS AND FUTURE WORK

This paper focuses on scheduling MapReduce tasks across CPU and GPU on a coupled CPU-GPU chip. We propose two different scheduling schemes: the *map-dividing scheme*, which dynamically divides the map tasks to both devices; and the *pipelining scheme*, which executes the *map* and *reduce* stages on different devices. To automatically achieve high load balance while keeping scheduling costs low, we use a *runtime tuning method* for the *map-dividing scheme*. Our implementation of MapReduce is based on continuous reduction, an design alternative which avoids the memory overhead of storing key-value pairs and makes it suitable to utilize the small but fast shared memory.

We have evaluated the performance of our framework using five applications. For 4 of the applications, our system achieves 1.21 to 2.1 speedups over the better of the CPU-only and GPU-only versions. The speedups over a single CPU core execution range from 3.25 to 28.68. The runtime tuning method we have developed achieves very low load imbalance, while keeping scheduling overheads low.

In the future, we will extend our framework to clusters consisting of integrated CPU-GPU nodes. We will also examine how the underlying ideas in our framework can be applied to other applications and programming models.

References

[1] David W. Aha, Dennis F. Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, pages 6:37–66, 1991.

[2] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[3] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, pages 281–288, 2006.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[5] Marwa Elteir, Heshan Lin, and Wu-chun Feng. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *ICPADS '11*, Tainan, Taiwan, December 2011.

[6] Reza Farivar, Abhishek Verma, Ellick Chan, and Roy Campbell. MITHRA: Multiple Data Independent Tasks on a Heterogeneous Resource Architecture. In *CLUSTER*, pages 1–10. IEEE, 2009.

[7] Dan Gillick, Arlo Faria, and John Denero. MapReduce: Distributed Computing for Machine Learning. 2008.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: The 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, 2010.

[9] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, pages 260–269, 2008.

[10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing Parallel Program Portable between CPU and GPU. In *PACT*, pages 217–226, 2010.

[11] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, August 1992.

[12] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[13] F. Ji and X. Ma. Using shared memory to accelerate mapreduce on graphics processing units. In *IPDPS '11: The 25th IEEE International Parallel & Distributed Processing Symposium*, pages 805–816, 2011.

[14] J. Liu and V. A. Saletore. Self-scheduling on distributed-memory machines. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 814–823, New York, NY, USA, 1993. ACM.

[15] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[16] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 104–113, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[17] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, December 1987.

[18] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of 13th HPCA*, pages 13–24, 2007.

[19] Vignesh T. Ravi, Wenjing Ma, Vignesh T. Ravi, and Gagan Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations'. In *Proceedings of International Conference on Supercomputing (ICS)*, 2010.

[20] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters. In *CloudCom '10*, pages 733–740, 2010.

[21] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IPDPS*, 2011.

[22] George Teodoro, Timothy D. R. Hartley, Ümit V. Çatalyürek, and Renato Ferreira. Run-time Optimizations for Replicated Dataflows on Heterogeneous Environments. In *HPDC*, pages 13–24, 2010.

[23] T.H. Tzen and L.M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4:87–98, 1993.

[24] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 244–255, New York, NY, USA, 2009. ACM.

[25] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *IISWC*, pages 198–207, 2009.