Multi-GPU MapReduce on GPU Clusters

Jeff A. Stuart
Department of Computer Science
University of California, Davis
stuart@cs.ucdavis.edu

John D. Owens
Department of Electrical and Computer Engineering
University of California, Davis
jowens@ece.ucdavis.edu

Abstract

We present GPMR, our stand-alone MapReduce library that leverages the power of GPU clusters for large-scale computing. To better utilize the GPU, we modify MapReduce by combining large amounts of map and reduce items into chunks and using partial reductions and accumulation. We use persistent map and reduce tasks and stress aspects of GPMR with a set of standard MapReduce benchmarks. We run these benchmarks on a GPU cluster and achieve desirable speedup and efficiency for all benchmarks. We compare our implementation to the current-best GPU-MapReduce library (runs only on a solo GPU) and a highly-optimized multi-core MapReduce to show the power of GPMR. We demonstrate how typical MapReduce tasks are easily modified to fit into GPMR and leverage a GPU cluster. We highlight how total and relative amounts of communication affect GPMR. We conclude with an exposition on the types of MapReduce tasks well-suited to GPMR, and why some tasks need more modifications than others to work well with GPMR.

1. Intro

While the performance of single-core CPUs has stagnated [1], both the programmability and performance of the graphics processor (GPU) have increased dramatically in recent years, with a broad variety of applications demonstrating order-of-magnitude gains in both performance and price-performance. GPUs particularly excel at the throughput-oriented workloads that are characteristic of scientific computation and large-scale server applications.

However, programmers and scientists focus most of their efforts on single-GPU development, neglecting the domain of GPU clusters. GPUs have yet to effectively tackle problems of true scale and we see two primary reasons for this: programming multi-GPU clusters is non-trivial and lacks powerful toolsets and APIs, and the GPU is often treated as a slave device in most GPU-computing applications. We tend to implement large-scale problems on CPU clusters and develop powerful toolsets and APIs for multiple CPUs. One such toolset is the large-dataset processing model called MapReduce [2], first developed at Google and widely used both at Google and elsewhere.

MapReduce makes programming clusters easier via a high-level model that puts the focus on the problem and not the mundane details inherent to distributed applications (e.g. communication, resource allocation). And while MapReduce targets data-parallel problems, it also performs well on other large-scale tasks. This has resulted in interest in developing new algorithms within the MapReduce programming model, as well as porting algorithms to fit MapReduce.

As with MapReduce, data-parallel processing is handled well by GPUs, even a GPU cluster. But existing GPU-MapReduce work only targets solo GPUs, and only in-core algorithms. We show how to implement MapReduce on a GPU cluster and hope to open GPU computing to larger application domains.

Implementing MapReduce on a cluster of GPUs poses several challenges. First, multi-GPU communication is difficult as GPUs cannot source or sink network I/O; thus supporting dynamic and efficient communication across many GPUs is hard. Second, GPUs do not have inherent out-of-core support and virtual memory. Third, a naive GPU-MapReduce implementation abstracts away the compututational resources of the GPU and possible optimizations. Finally, the MapReduce model does not explicitly handle the system architecture inherent with GPUs.

Our library, GPU MapReduce (GPMR for short, pronounced G-Primer), specifically tackles these chal-

lenges: data movement, out-of-core data management, and maintaining full GPU access, as well as modifying MapReduce to be more natural and efficient on GPUs. However, GPMR specifically targets the MapReduce programming model, not a fully-featured MapReduce implementation - GPMR is stand-alone and does not sit atop Hadoop or another MapReduce package. In particular, it does not handle fault tolerance, and it does not provide a distributed file system (it is storage agnostic). Within the model, though, we implement specific extensions for the GPU, including batching *Maps* and *Reduces* via *Chunking* to maintain GPU utilization, adding *Accumulation* to the *Map* substage, adding a *Partial Reduction* substage, handling out-of-core datasets, and assembling the MapReduce pipeline to achieve a high overlap of communication and computation. We evaluate our library not only on problems that scale well but also on problems with significant scalability challenges; these are presented in Section 5.

2. Background

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

2.1. The GPU and CUDA

The GPU is a many-core machine with multiple SIMD multiprocessors (SM) that can run thousands of concurrent threads. GPMR leverages CUDA [3] and we use its terminology to describe the GPU. Up to 512 GPU threads are grouped into scheduling units (blocks). Within a block, threads are grouped into 32-wide SIMD execution units (warps). An SM can keep many blocks resident at once, but only executes one warp's threads in lockstep at a time. Context switching between warps (even from different blocks) has negligible overhead, allowing the GPU to hide long-latency operations. An SM has thousands of registers and kilobytes of scratchpad memory (shared memory). Registers are tied to specific threads, so shared memory is used for thread communication. Nickolls et al. [3] has a more extensive discussion of CUDA and how it maps to the GPU.

Kirk and Hwu's book [4] outlines techniques for efficient GPU programs, three of which we summarize here. Efficient programs require many threads and blocks to keep the GPU full, but each block should use few resources to allow many blocks to remain simultaneously resident on an SM. Memory accesses from a warp are most efficient when accessing consecutive addresses in memory (called coalescing). And because the transfer cost between CPU and GPU is high, and even the cost of memory accesses on the GPU is high compared to the cost of arithmetic operations, efficient programs must perform a lot of work per data element to amortize memory-transfer costs.

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

2.2. MapReduce

MapReduce is a programming model that combines two separate higher-order functions in functional-programming languages, *map* and *reduce* (also known as *fold*). Programmers specify *map* and *reduce* functions. The input to *map* is a set of data items; each map invocation outputs a sequence of independent key-value pairs. All like-keyed values are grouped together and passed to a *reduce* function which processes them to output a sequence of new values.

Google's MapReduce implementation was built for large-scale data processing, large data sets are input to many mapping nodes. Intermediate data is streamed back out to buffer cache, partitioned, and sent to processing nodes to be reduced. Google uses MapReduce for much of its internal data processing, and because of its success Google popularized MapReduce with industry and academia. Since that time, developers and researchers have created many MapReduce packages. We compare GPMR to two popular packages, Phoenix [8] (C++), and Mars [9] (CUDA).

i-MapReduce [10] (previously CGL MapReduce) is another MapReduce package, and potentially the first to use data streams instead of hard disk access. Intermediate data values and reduction results are streamed directly to new mapper and reducer nodes for further processing. This allows for an efficient, iterative MapReduce algorithm with many consecutive MapReduce processes.

Recent efforts have gone towards porting MapReduce to parallel processors like GPUs and IBM's Cell. Catanzaro et al. created a single-node library for GPUs [11] but the focus was on many small tasks; the main contribution was creating an efficient small-sequence sort on the GPU. Mars was the first large-scale GPU system, though its scalability is limited; it

uses only one GPU and in-GPU-core tasks. Another shortcoming is that the library, not the user, schedules threads and blocks, making it hard to fully exploit certain GPU capabilities (e.g. inter-block communication). MapCG [12] is another GPU-based MapReduce library. The main goal was to allow for portable multicore MapReduce code that could run on the GPU. Like Mars, MapCG only offers limited scalability as it uses but one GPU.

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

3. Moving from the CPU to the GPU

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

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

Processing items in chunks yields additional benefits. Each GPU thread knows the item on which every other GPU thread is working, which allows for more natural programming on the GPU and block-level communication within a chunk. One of the contentious points with chunking is the trade off between abstraction and performance. Sticking to the design principles of GPMR, we want to give full access to the GPU (i.e. allow block-wide reductions, manually schedule threads per block, etc.) to make GPMR as fast and efficient as possible, so chunking simply makes sense. We cannot speak for every developer, but we believe that chunking provides a good level of abstraction from the original data. And since chunking also allows developers full access to the GPU, they can leverage higher-level GPU operations.

We now turn to optimization opportunities from restructuring the pipeline. Any experienced parallel programmer would say that the key to parallel efficiency is to reduce communication times as much as possible and to overlap communication with computation. All of our optimizations focus on reducing communication times at the expense of more computation time. We believe that communication is almost always the bottleneck and GPU computation is relatively cheap.

Each optimization essentially reconfigures the MapReduce pipeline[1]. The first optimization is a variation on an existing optimization in MapReduce, *Combine*. Before the library transmits pairs to *Reducers*, like-keyed pairs can be *Combined*, which does not emit any new keys but instead generates a single value to be associated with a key, ensuring that the node only sends one value per key to a *Reducer*. This is not a new optimization, but it requires a GPU implementation. GPMR must use an efficient storage strategy for pre-emitted key-value pairs to stream them back down to the GPU for combination. Unlike in Hadoop, *Combine* happens only when all *Maps* complete in order to minimize network traffic as much as possible. This differs from the Hadoop *Combine* in that it typically only combines values from the same *Map* instance. The purpose of the GPMR *Combine* substage is to reduce

1. As there are many combinations, showing all with a full explanation of where they make the most sense is beyond the limits of this paper.

network traffic at the expense of added PCI-e transfer time and GPU computation time.

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

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

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

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

4. Implementation

We wrote GPMR in C++ and CUDA and designed it to be easy-to-use and extensible while still allowing full access to the GPU. Every part of the MapReduce pipeline is programmable by the user, though we provide default implementations of the *Partitioner*, *Scheduler*, and *Sorter*.

Each GPU is controlled by a separate process and each process executes the MapReduce pipeline. The three primary stages to the MapReduce work flow are *Map*, *Sort*, and *Reduce*. *Map* is divided into many separate substages, while *Sort* and *Reduce* are each indivisible. All stages and substages are customizable by the user. Figure 1 shows a diagram of a typical GPMR work flow.

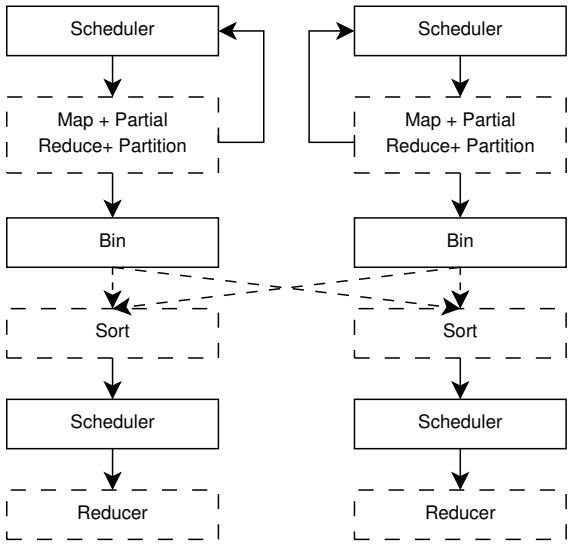

Figure 1: The MapReduce work flow: The GPU executes dashed-outline stages, the CPU executes solid-outline stages. The dashed line transition between *Bin* and *Sort* stage denotes that GPMR transfers data to another processor, potentially over the network. GPMR individually streams chunks to GPUs and executes a *Mapper* on each. If supplied, GPMR executes a *Partial Reduction* and then if no *Combiner* is supplied, a partition. GPMR copies the key-value sets to the CPU and transmits them over the network while the scheduler loops over a new chunk. If the user supplies a *Combiner*, GPMR stores all key-value pairs in CPU memory (due to the size limitations of the GPU memory) until all *Maps* complete, and then executes the *Combiner* on each unique key before partitioning the key-value pairs. Once each process receives all of its pairs, GPMR executes *Sort* followed by *Reduce*.

4.1. Map Stage

The *Map* stage is broken into several substages; the *Map* itself, *Accumulation*, *Partial Reduction*, *Combination*, and *Partition*. Depending on which of these substages (described earlier) the user activates, this stage can take many forms.

Map processes an input chunk and outputs a set of key-value pairs. The entire chunk is copied to the GPU via a user-supplied function (typically a wrapper for *cudaMemcpyAsync*) at once and processed by one or more user-supplied kernels. This model allows the raw use of the GPU while still maintaining the MapReduce model of data-independent elements. *Map* works on one chunk at a time as GPMR assumes each chunk and its output will consume most of the GPU memory. Chunking also gives us an efficient, out-of-core technique for GPU MapReduce. We can use chunks that are a fraction of the size of available memory, allowing us to *Map* or *Reduce* a chunk while simultaneously streaming another chunk to or from the GPU.

One particular facet of the *Map* stage (and the *Reduce*) is the need for load balancing. GPMR tracks the per-GPU work in a dynamic queue. If one GPU finishes its work in its local queue and other GPUs have much more work to do, we shift chunks between the local queues. This is important as due to this requirement, chunks must implement a serialization method.

Partition. The *Partition* substage divides key-value pairs into buckets to be sent to each *Reducer*, where most likely a *Reducer* resides on a different node. The *Partitioner* arranges all key-value pairs for a specific *Reducer* consecutively, thus requiring only one network send per *Reducer*. If the user omits *Partition*, all pairs are sent to a single *Reducer* (best for jobs with small intermediate data).

We supply a default round-robin *Partitioner* for integer keys. But we made the *Partitioner* extensible for a few reasons. First, we impose no strict definition of a key and therefore we cannot assume keys are four-byte integers. Hence we cannot simply use modulation or division to determine the destination *Reducer*. Second, even when keys are integer values, there is no best-performance distribution for all MapReduce jobs (e.g. round-robin vs. consecutive blocks).

Bin. The *Bin* substage is responsible for transmitting partitioned key-value pairs to reduce tasks. This is typically done either through network I/O or a distributed file system; GPMR is storage and transfer-mechanism agnostic, so it makes no explicit assumptions about which the programmer uses. *Bin* is the only stage of the pipeline executed on the CPU as opposed to the GPU. This is because the GPU cannot interact with the file system or network. Since GPMR evaluates the *Bin* substage on the CPU, GPMR takes advantage of modern multicore processors by running it in a separate thread, yielding a more thorough overlap of communication with the mapping computation.

The Map Pipeline. Fitting all of the *Map* substages together leads to many possible configurations. Knowing the effects of each individual piece helps to create a more efficient overall pipeline. Users can mitigate a large number of intermediate key-value pairs by implementing *Partial Reduction* or *Combination*. They can optimize for a small number of unique keys by using *Accumulation*, at the cost of not overlapping communication with computation. A pipeline with a large set of key-value pairs and many unique keys can be more efficient via *Partial Reduction* and tuning the size of each chunk to allow overlap in computation and communication.

We summarize the effects of the most common pipeline configurations. Using *Accumulation* eliminates the need for *Partial Reduce* and *Combine*. *Mapping* without *Accumulation* or *Combination*, and optionally with *Partial Reduce*, causes partitioning after every *Map* completes. Using *Accumulation* or *Combination* causes the library to execute only one partition per full *Map* stage. This happens after the *Combine* substage/final *Accumulation*. Not using *Accumulation* or *Combination* allows for *Binning* to take place concurrently with *Maps*. Conversely, using *Accumulation* or *Combination* mandates that *Binning* only happens once all *Maps* finish.

4.2. Sort Stage

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

4.3. Reduce Stage

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

4.4. Mapping Applications to GPMR

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

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

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

5. Methodology

5.1. Cluster Configuration

To test GPMR, we used the *Accelerator* cluster at the National Center for Supercomputing Applications. The cluster has a total of 32 nodes, each with an NVIDIA Tesla S1070. The S1070 features 4 NVIDIA GT200 GPUs each with 4 GB of RAM (though for testing purposes, we limit RAM usage to 1 GB). Each node has 2 dual-core 2.4 GHz AMD Opterons and 8 GB of RAM. The nodes are connected via QDR Inifiniband connected to generation-1 PCI-e. Due to concurrent users on the cluster, we performed tests on up to 64 GPUs. 64 CPUs is considered a small-to-medium cluster, but very few existing cluster installations worldwide feature more than 64 GPUs. Each node in this cluster runs RHEL 4 with the 2.6.27 Linux kernel, and uses the NVIDIA 195.36 driver and the CUDA 3.0 toolkit. We compiled our library and all test software using GCC 4.3.2 and MVAPICH2.

We ran the following benchmarks: Matrix Multiplication (MM, multiplies two large square matrices); Sparse Integer Occurrence (SIO, counts the number of times each integer appears in a large dataset); Word Occurrence (WO, counts the number of times each word occurs in a text corpus); Linear Regression (LR, computes a linear model of a set of data), and K-Means Clustering (KMC, partitions a set of data points into clusters). These are all typical benchmarks for testing new MapReduce libraries.

5.2. Benchmark Guidelines

Perhaps the most important trait of any MapReduce application is its scalability. MapReduce libraries do quite well at parallelizing and scaling algorithms that are already scalable. GPMR can do just that: MM has near-perfect scalability even to 64 GPUs. More important is understanding the performance of GPMR in the presence of algorithms that are not perfectly scalable. We thus choose 4 benchmarks that fail to scale for a diverse set of reasons and analyze their performance.

Besides scalability, there were other potential performance bottlenecks in GPMR. To analyze the impact of these bottlenecks, we needed to test many aspects of the library. We list these aspects below and include both a description of why they are important and which benchmarks stressed that aspect. Afterward, we explain the typical CPU implementation of that benchmark, and then how we modify it to better fit into the GPU world and onto our GPMR application.

- Multiple emits per thread—Many applications emit more than one key per map item. This is easy for a CPU, but takes extra planning with a GPU. LR, KMC, and WO all test this aspect.
- Non-uniform number of emits per thread—Different map items from the same chunk may emit a different number of key-value pairs. This feature is tested by WO.

- Sparsity of keys—An ideal world would give us a *compact set* of integer keys: if the maximum key is X, all keys in the range $[0,X]$ are present. This yields efficient partitioning (assuming all keys have an equal amount of reduction work) and efficient sorting. Of course, many non-contrived MapReduce jobs don't have such a key set, thus we need more efficient means to partition and sort. SIO tests this aspect.
- Accumulation—Some MapReduce jobs require only a small number of unique keys that are combined via an associative and commutative operator (e.g., addition or multiplication). In these situations, *Accumulation* mitigates the cost of PCI-e transfers. LR and KMC both use accumulation.
- Many key-value pairs—Some mapping tasks emit a lot of pairs, which imposes a high demand on memory, the PCI-e bus, and the network interconnect. Such tasks might suffer from limited scalability if the cost of transferring pairs cannot be sufficiently overlapped with *Mapping*. SIO especially stresses this aspect of GPMR.
- Compute-bound Scalability—One of the biggest advantages to MapReduce is that a compute-bound task should scale very well. However, if the library imposes any overhead or does not handle its internals well, this affects scalability. We wanted to make sure that the internals of GPMR did not interfere with scalability. MM is very much GPU-compute bound.

5.3. Benchmark Implementations

5.3.1. Matrix Multiplication. MM is the only application we chose that is, in itself, highly scalable on the GPU. We implemented a straightforward square matrix multiply in two phases. From the very beginning, we had to craft the algorithm carefully.

The common CPU MapReduce MM algorithm for multiplying square matrices of dimension M uses M^2 vector-vector multiplications in *Map*, one for each element of the result. There is no *Sort* or *Reduce*. This falls short on the GPU in two key ways. First, vector-vector multiply works well on a GPU only when reading row vectors of a matrix (coalescing rules). Second, the GPU has scratchpad memory but not nearly enough to contain both a row vector and a column vector if M is sufficiently large.

Due to the limitations of the above algorithm, we used a hierarchical approach from typical cache-oblivious algorithms [20]. We tile each matrix into smaller and smaller pieces until each block in the GPU can fit in shared memory. Then the entire block performs a small matrix multiplication where each thread computes an element of the result as an inner product. Each submatrix of the result is also a tile of inner products. We thus transform the original formulation into N^3 uniform-sized—and at least 1024^2—tile multiplications. Each *Map* tiles this small matrix into a set of 256^3 matrices, which are further divided into 16^2 matrices. Each block performs a full inner product of tiles by performing many 16^2 tile multiplications. We stop the division here because a block of 256 threads can read 16^2 values in a coalesced manner and perform enough computation to keep throughput on the GPU very high. To add all the partial sums of each submatrix and form the result, we bypass *Sort* and *Reduce* and implement another *Map* in a separate MapReduce[2].

The result is a very scalable, out-of-core implementation of MM that uses the GPU much more efficiently (and runs several orders of magnitude faster) than the direct port of the typical CPU implementation, even on small matrices. Another point that is well illustrated by our algorithm is that GPMR allows for a natural use of the GPU by not forcing any specific thread-to-task mapping (e.g. one GPU thread per map task).

5.3.2. Sparse Integer Occurrence (SIO). SIO counts the number of occurrences of each integer in a sequence with a random distribution. The GPMR implementation is similar to a CPU implementation and is straightforward. We chunk as many integers as possible in a tightly packed array. In the typical CPU implementation, each thread reads one integer I per map and emits $\langle I,1\rangle$. Our *Mapper* reads two integers, I_1 and I_2, to efficiently access GPU memory and then emits $\langle I_1,1\rangle$ and $\langle I_2,1\rangle$. We forego *Partial Reduction* and *Accumulation* as they yield no speedup with our intermediate data, and we skip *Combine* as it causes slowdown. We use the default *Sort*. For *Reduce*, we knew we could skip the value summation and just output the values count, but this goes against the spirit of the benchmark and is not robust to pipeline changes. We tried assigning each key set to a block instead of a thread, and then performing an iterative reduction over the key values. This yielded poor performance because the data set was sparse and many keys had less than

2. This is due to memory constraints. We built a scalable MM algorithm using GPMR, and to do so, we had to take into account that while *Maps* are broken into chunks, a single-key reduction must be entirely in-core. Once too many tiles of a matrix are necessary, this in-core requirement would not be met and thus we had to split this into two separate GPMR tasks.

five values. Our final and best implementation of the reducer is the same as the CPU approach: one key per thread, where the thread sums all values.

5.3.3. Word Occurrence. Like SIO, WO counts all occurrences of a unique object. Unlike SIO, the objects are words, not integers. Also, the output set for WO is much smaller, leading to a different configuration of the pipeline and drastically different scaling. The input is a collection of text, with words taken from a predetermined corpus, separated at line boundaries. For our test cases, we used randomly generated text from a forty-three thousand word dictionary. Each chunk contains millions of bytes.

The typical CPU *Mapper* implementation has each *Map* task scan one line of text and emit $\langle W, 1\rangle$ for every found word W. This does not work on a GPU. Sending one line of text to a *Mapper* is fine, but we should not use strings as keys; strings cannot be read in a single instruction as their length varies, and forcing all strings into a fixed-size area yields poor storage performance in many cases—storing a four-character word in sixteen bytes wastes 75% of the key space. Using variable-sized keys either requires more space for a key directory or more time as GPMR must use atomics to emit keys. Instead, we used a minimal perfect hash [21] to assign each key a unique, four-byte integer value. Thus, the GPU *Map* kernel gives each thread one line of text and scans the text for words, then finishes by hashing W and emitting $\langle hash(W), 1\rangle$.

As our dictionary is small (43k integer-integer pairs requires less than 350 kB), we chose to use *Accumulation*. An initial *Map* task emits all keys with the value 0. Afterward, whenever we emit a key-value pair, we simply index into the emit space and use a fire-and-forget atomic instruction to increment the associated value. By using *Accumulation* we mitigate PCI-e and network transfer and almost completely remove communication, the bottleneck of the CPU implementation.

We do not use a *Partitioner* because we send all key-value pairs to one node. This is fast on a small number of GPUs since a single *Reduce* kernel can handle the task. However, once the number of GPUs crosses a certain threshold, key-value pair communication bottlenecks the job, so we enable the default round-robin *Partitioner* after the crossover point. For *Sorting*, we also use the provided sorter in GPMR.

We started with the CPU *Reduce* task; each thread gets one key and performs a linear summation of all the values. But we noted that once we had more than a few GPUs, the reduction time, while still quite small, was significant. The are two reasons for this: the reads are not coalesced, and each thread has to wait a (relatively) long time for each read to finish to complete the sum. We also noted that if we emit all 43k keys, even when not all words were found by a GPU, the work per key would be balanced. Thus, we changed our implementation to assign each key to a warp (not a block). Each warp iterates over its value set, with the threads in the warp reading and summing in a coalesced fashion until it reads all values. Then, the kernel does a warp-wide reduction to finish the summation. The overall effect was that our reduction times were reduced (by an order of magnitude in some cases) down to less than 3 ms.

5.3.4. K-Means Clustering. KMC is used in machine learning. The basic task takes a set of points in space and determines clusters that can best approximate the space. For both the CPU and GPU benchmark, we use a fixed-size random set of cluster centers at job startup. The typical CPU implementation of the *Map* kernel reads one point P, finds the index of the closest center C, and emits $\langle index(C), P\rangle$. We implemented this in GPMR and saw poor results for three reasons: each thread loads its own point (thus not guaranteeing coalesced reads), we see far too many intermediate key-value pairs, and the size of the emitted pair causes us to issue non-coalesced writes.

The biggest change we made was to use persistent threads [22] within the *Map* stage to process many elements per thread and to use atomic-free *Accumulation* (discussed below). The entire block reads points in a coalesced manner and each thread finds the closest center C. The block performs a series of reductions of all points belonging to C. As each reduction completes, the block's master thread accumulates the reduction value to global memory. Because the GPUs we used do not have floating-point atomics, we were forced to use a per-block global-memory pool. After the primary kernel completes, another kernel reduces the values in each block's pool and emits the final values. The CPU implementation emits the $\langle index(clustercenter), point\rangle$. The GPU emits $\langle C, P_{dim}\rangle$ for each dimension, as well as an extra key per center (the number of influencing points). This allows for coalesced writes with negligible overhead for a small number of dimensions. If the number of dimensions and centers are large, the typical CPU approach may prove more efficient as it uses less storage. We also optimized by using *Accumulation*. These optimizations

reduced *Map* times by almost 8×.

For our *Partitioner*, we sent all keys of a center to one GPU. Our key-value pairs allow us to use the default *Sort* provided by GPMR. In *Reduce*, each thread sums the values for a single key. Even with many GPUs, we used sufficiently few centers and dimensions to make the full *Reduce* time negligible. KMC is an iterative process; the MapReduce results are new cluster centers, and a full KMC implementation repeats a fixed number of times or until convergence. Our benchmark simply runs one iteration.

5.3.5. Linear Regression. LR models the relationship between two parameters influenced by unknown variables. We store chunks in much the same way as KMC, grouping many points together and tightly-packing them in arrays. In fact, LR is similar to KMC in many ways and the same optimizations work well. We use persistent threads to compute the relationship as well as our own internal *Accumulation*. As with KMC, we achieve an almost order-of-magnitude speedup over a direct port of the typical CPU implementation that we modeled after the straightforward CPU implementation. The *Mapper* emits only six keys upon completion, and thus we do not use *Partitioning* (the network overhead is minimal in both cases). We use the default sort and perform reductions in a key-per-thread manner (reduction time is virtually nil).

6. Results

Of our five benchmarks, we expected only MM to scale well to 64 GPUs and beyond. While our optimizations greatly aid the scalability of other benchmarks, we expect either communication or the GPMR internals to inhibit scalability. Nonetheless, our analysis of the reasons these applications do not scale well are illuminating, and we emphasize that the scale of our benchmarks is something previously unattainable with GPU MapReduce libraries.

We ran each GPMR benchmark against two datasets. One tests strong scalability with four different-sized inputs; the other tests weak scalability. We only ran Phoenix against the former, as Phoenix only runs on a single node. Table 1 describes dataset sizes.

Figure 2 shows runtime breakdowns for our benchmarks on various cluster configurations. This figure shows how our applications exhibit different characteristics as they execute at scale. MM is very compute-bound and exhibits strong scaling. KMC is also mostly compute-bound in *Map*, but communication requirements affect performance at scale. LR requires, per element, very little time for its *Maps*, but required communication impacts runtime at scale. SIO shows changes in characteristics as scale is increased because the bottleneck changes with the scale: the bottleneck changes from *Sort* with a small number of GPUs to transmitting data with many GPUs. WO, like SIO, changes as GPU count increases, primarily because communication is less cumbersome with the the introduction of a partitioner once the GPU count crosses a certain threshold. For all but MM, it's difficult to further improve the application because the communication simply is a bottleneck; we believe we have mitigated those factors as much as is feasible with our implementation.

We show parallel-efficiency, where we use the standard definition of efficiency of $\textit{Efficiency} = \frac{\textit{Speedup}}{\#GPUs}$, graphs for each benchmark in Figure 3. Note the importance of *Accumulation*. We saw dramatically worse performance in KMC, LR, and especially WO before implementing *Accumulation*; before this addition, all three had similar characteristics to SIO (which cannot compact intermediate data well).

We also compared our results to both Mars[3] and Phoenix, an optimized CPU-based MapReduce library. Phoenix runs on a single node, so we compare against GPMR with one GPU and with four GPUs (both on one node). Table 2 summarizes speedup results over Phoenix, while Table 3 gives speedup results of GPMR over Mars. Note that GPMR with one GPU is faster on all benchmarks than either Phoenix or Mars, but also shows good scalability to four GPUs.

Source code size is another important metric. One significant benefit of MapReduce is the high level of abstraction, which reduces code size and development time, since MapReduce handles the low-level details (communication, scheduling, etc.). Table 4 shows the number of lines required for the three benchmarks in Phoenix, Mars, and GPMR. We would also like to show developer time for each benchmark and platform, but Mars and Phoenix didn't published this (we used the applications provided so as not to introduce bias in Mars's or Phoenix's runtimes). As a frame of reference, the lead author of this paper implemented and tested MM in GPMR in three hours, SIO in half an hour, KMC in two hours, LR in two hours, and WO in four hours. KMC, LR, and WO were then later modified in about half an hour each to add *Accumulation*.

3. We would have liked to compare our results against MapCG as it advertised better results, but the software was not freely available.

	MM	SIO	WO	KMC	LR
Input Element Size	—	4 bytes	1 byte	16 bytes	8 bytes
# Elems in first set ($\times 10^6$)	1024^2, 2048^2, 4096^2, 16384^2	1, 8, 32, 128	1, 16, 64, 512	1, 8, 32, 512	1, 16, 64, 512
# Elems in second set ($\times 10^6$/GPU)	—	1, 2, 4, 8, 16, 32	1, 2, 4, 8, 16, 32, 64, 128, 256	1, 2, 4, 8, 16, 32	1, 2, 4, 8, 16 32, 64

Table 1: Dataset Sizes for all benchmarks. We tested Phoenix against the first input set for SIO, KMC, LR, and the second set for WO. We tested Mars against the first input set for KMC, MM, and WO (these were included in the distribution). We tested GPMR against all available input sets.

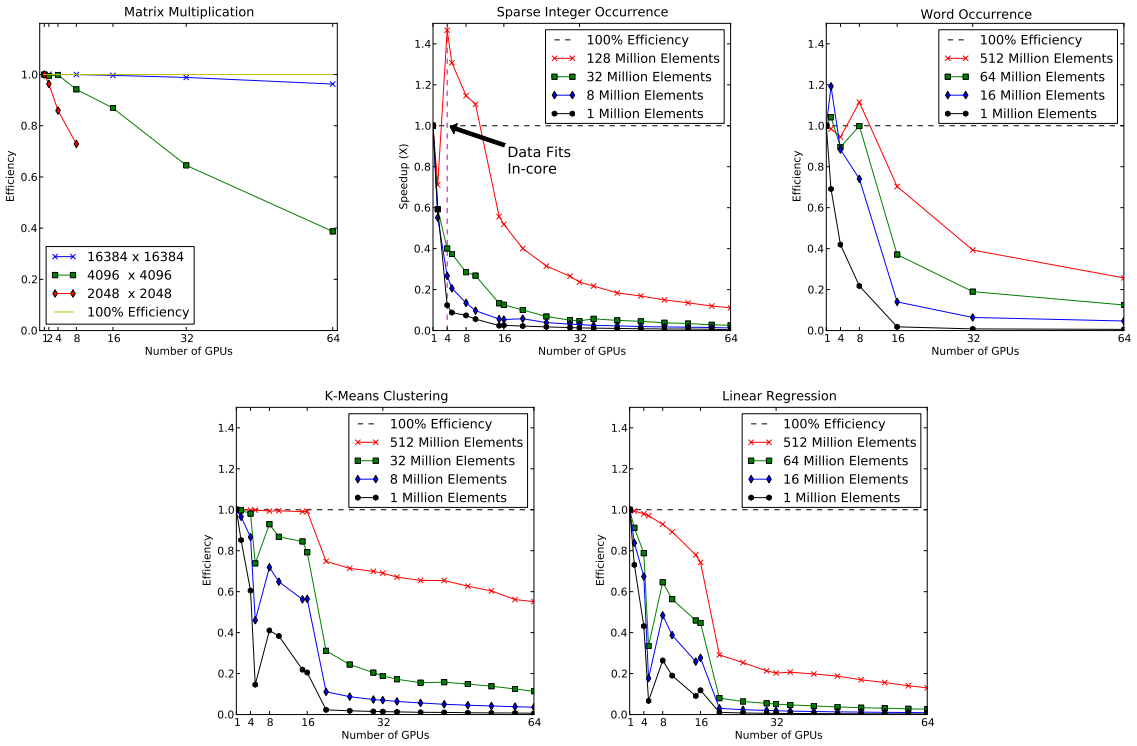

Figure 3: GPMR Parallel Efficiency for (top left to bottom right) MM, SIO, WO, KMC, and LR. MM is the only GPU-compute bound application. SIO exhibits super-linear speedup at four GPUs because the data fits in core. Scalability drops as node count increases due to overhead from the network. WO exhibits similar behavior. K-Means stops exhibiting strong scaling beyond 20 GPUs, but still has more than 60% efficiency at 64 GPUs. LR exhibits poor scaling beyond four GPUs (the dip in the graph is when the job becomes multi-node, but with an imbalance in used GPUs per node). Little time is required to perform the map, and communication affects scalability even though it's quite light.

	MM	KMC	LR	SIO	WO
1-GPU	162.712	2.991	1.296	1.450	11.080
4-GPU	559.209	11.726	4.085	2.322	18.441

Table 2: Speedup for GPMR over Phoenix on our large (second-biggest) input data from our first set. The exception is MM, for which we use our small input set (Phoenix required almost twenty seconds to multiply two 1024×1024 matrices). The 4 GPU test uses 4 GPUs on a single node.

	MM	KMC	WO
1-GPU Speedup	2.695	37.344	3.098
4-GPU Speedup	10.760	129.425	11.709

Table 3: Speedup for GPMR over Mars on 4096×4096 Matrix Multiplication, an 8M-point K-Means Clustering, and a 512 MB Word Occurrence. These sizes represent the largest problems that can meet the in-core memory requirements of Mars. The 4 GPU test uses 4 GPUs on a single node.

	MM	KMC	WO
Phoenix	317	345	231
Mars	235	152	140
GPMR	214	129	397

Table 4: Lines of source code for the three common benchmarks in Phoenix, Mars, and GPMR. We exclude setup as it was about equal for all benchmarks and had little to do with the actual MapReduce code. For GPMR we included boilerplate code (ḣ files, and CUDA-kernel wrapper functions). GPMR's WO is large because of the hashing required.

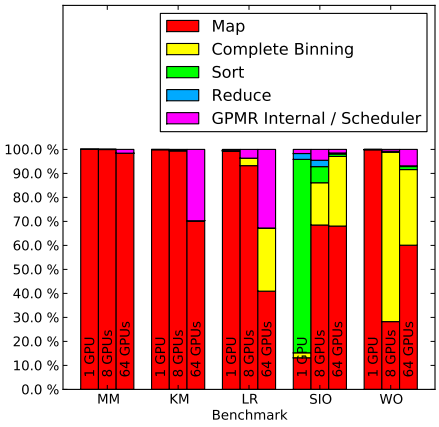

Figure 2: GPMR runtime breakdowns on our the largest datasets. This figure shows how each application exhibits different runtime characteristics, and also how exhibited characteristics change as we increase the number of GPUs.

7. Conclusion

GPMR offers many benefits to MapReduce programmers, the most important of which are new capabilities, scalability, and ease-of-use. While it is unrealistic to expect perfect scalability from all but the most compute-bound tasks, the minimal overhead and transfer costs of GPMR place it well in comparison to other MapReduce libraries. GPMR also gives developers flexibility in several areas, especially compared to Mars. GPMR allows flexible mappings between threads and keys and customization of the MapReduce pipeline with communication-reducing stages (both PCI-e and network), while still providing sensible default implementations. Our results show that even difficult applications that typically are not addressed by GPUs still show moderate scalability with GPMR.

Like other MapReduce libraries, GPMR does not scale a communication-bound job well. These jobs are gated by PCI-e and network/disk throughput. Because the advantage of the GPU over the CPU is primarily computation, MapReduce tasks need substantial computation to reap the benefits via GPMR.

We would like to draw a broad conclusion about the proper configuration of a GPU cluster for MapReduce. Unfortunately, our results show this is dependent on the characteristics of the task at hand, including its input sizes, intermediate data sizes, and the computational complexity of the *Map* and *Reduce*. This implies that each job, and to a certain degree each input set for a job, may require a different configuration to run at maximum potential. Thus, we conclude that GPMR users should devote at least some time to deciding what stages of the pipeline are suitable for their jobs.

Another hardware question for any GPMR user is whether to use a cluster configuration with enough GPU internal memory to avoid out-of-core computation. Many modern GPUs still have 512 MB of RAM or less, and thus it is quite hard to fit a problem of significant size in core. Our experience shows that with the proper chunk arrangement, and careful use of *Accumulation* or *Partial Reduction* when available, out-of-core work does not have a strong effect on GPMR jobs, and thus using a small set of GPUs and executing an out-of-core job can show the same scalability as a large, many-GPU in-core job.

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

8. Acknowledgements

Thanks to our funding agencies, the SciDAC Institute for Ultrascale Visualization and the National Science Foundation (Awards OCI-1032859 and CCF-1017399), to NVIDIA for equipment donations, and to NCSA and Wen-Mei Hwu for allowing us access to their GPU cluster. We would also like to thank Jeff Dean and Heshan Lin for their valuable feedback.

References

[1] M. Ekman, F. Warg, and J. Nilsson, "An in-depth look at computer performance growth," *ACM SIGARCH*

Computer Architecture News, vol. 33, no. 1, pp. 144–147, Mar. 2005.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, pp. 40–53, Mar./Apr. 2008.

[4] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

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

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

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

[8] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *International Symposium on High-Performance Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 13–24.

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

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

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

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

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

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

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

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

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

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

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

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

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

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