Co-processing SPMD computation on CPUs and GPUs cluster

Hui Li, Geoffrey Fox, Gregor von Laszewski, Arun Chauhan
School of Informatics and Computing, Pervasive Technology Institute
Indiana University Bloomington
lihui@indiana.edu, gcf@indiana.edu, laszewski@gmail.com, achauhan@indiana.edu

***Abstract*— Heterogeneous parallel systems with multi processors and accelerators are becoming ubiquitous due to better cost-performance and energy-efficiency. These heterogeneous processor architectures have different instruction sets and are optimized for either task-latency or throughput purposes. Challenges occur in regard to programmability and performance when running SPMD tasks on heterogeneous devices. In order to meet these challenges, we implemented a parallel runtime system that used to co-process SPMD computation on CPUs and GPUs clusters. Furthermore, we are proposing an analytic model to automatically schedule SPMD tasks on heterogeneous clusters. Our analytic model is derived from the roofline model, and therefore it can be applied to a wider range of SPMD applications and hardware devices. The experimental results of the C-means, GMM, and GEMV show good speedup in practical heterogeneous cluster environments.**

Keywords: SPMD, GPU, CUDA, Multi-Level-Scheduler

I. INTRODUCTION

Heterogeneous parallel systems with multi-core, many-core processors and accelerators are becoming ubiquitous due to better cost-performance and energy-efficiency [1]. In mid-range HPC systems, a hybrid cluster with hundreds of GPU cards is capable of providing performance over one petaflops, while the same scale CPU-based cluster can provide one teraflops of peak performance. In high-end HPC systems, Titan, a hybrid cluster using CPUs provided by AMD, Inc and GPUs provided by NVIDIA, Inc became the fastest supercomputer in peak performance in 2012.

Two fundamental measures for processor performance are task latency and throughput [1]. The traditional CPU is optimized for a lower latency of operations in clock cycles. However this usage pattern of using single core has been replaced by new system using multiple cores available to the overall system. This includes architecture such as Intel Xeon Phi, AMD Opteron. These multi-core and many-core CPUs can exploit modest parallel workloads for multiple tasks. These parallel tasks can have different instructions and work on different types of data sets, or MIMD. The current generation of graphical processing units (GPUs) contain large number of simple processing cores that are optimized for computation that contain single-instruction, multiple threads, or SIMT. GPUs sacrifice single thread execution speed in order to achieve aggregated high throughput across all of the threads. More recently, the CPUs system is augmented with other processing engines such as GPU, which we call this system as fat node. The purpose of fat nodes is to keep more processing on the local node. The AMD take a more aggressive strategy based on this idea and it has merged GPU/CPU with Fusion APU.

The NVIDIA's CUDA [2] and Khronos Group OpenCL [3] are the current and most widely used GPU programming tools. Both CUDA and OpenCL can compile source code into binaries to run on CPUs and GPUs, respectively. However, this process cannot be done automatically, and requires programming efforts from programmers. If these heterogeneous resources are assigned to users, then the CPU cores are idle while conducting computations on GPUs, or vice versa. In order to solve this problem, one should first meet the programmability challenge of how to map the SPMD computation to the CPUs and GPUs. NVIDIA uses the terminology SIMT, "Single Instruction, Multiple Threads" to present the programming model on the GPU. SIMT can be considered a hybrid between vector processing and hardware threads. The difficulty of writing CUDA program is that developers need to organize the kernel threads and carefully arrange the memory access patterns. For CPUs, the SPMD style computations are already presented on CPUs by using many programming tools such as Pthreads, OpenMP, and MPI. Other programmability difficulties are that the developer needs to explicitly split the input data, and to handle the communications across the cluster.

The MapReduce [4] programming model was popularized at Google, and has been successfully applied to various SPMD applications on shared memory and distributed memory systems. Developing programs with the MapReduce program is easy because MapReduce runtime hides the implementation details such as data movement, task scheduling and work load balance issues from the developers. Research has proven that executing MapReduce computations on GPUs is not only feasible but also practical. Recently, some MapReduce like runtime frameworks have been used to run tasks on CPUs or GPUs simultaneously [5] [6]. However, these works usually are constrained when it comes to SPMD applications, or introduced extra overhead during computation, which is not general and desirable.

The Roofline model [7] provides researchers with a graphical aid that provides realistic expectations of performance and productivity for a given application on a specific hardware device. It models the performance as the function of the arithmetic intensity of the target application and some performance related parameters of the hardware including DRAM, PCI-E bandwidth, and the peak performance of the CPU or the GPU. Generally, for applications that have low arithmetic intensity, such as log analysis and GEMV, the performance bottleneck lies in the disk I/O. For applications with moderate arithmetic intensity, such as FFT, and Kmeans, the performance bottleneck lies in the DRAM, and PCI-E bandwidth. For applications with high arithmetic intensity, such as DGEMM, the performance bottleneck lies in the L2 cache and the peak performance of computation unites. All of these types of information are critical in regard to making scheduling decisions and therefore they can be used as parameters for the mathematic modeling of task scheduling on GPUs and CPUs.

We implemented a parallel runtime system in order to co-process the SPMD computation on modern NVIDIA GPUs and Intel Xeon CPUs on distributed memory systems. We provided a MapReduce like programming interface for developers. More importantly, we leveraged the roofline model to derive an analytic model that is used to create automatic scheduling plan for placing SPMD computations on the GPUs and CPUs cluster. In order to evaluate this scheduling model, we implemented C-means, GMM and GEMV applications and conducted comprehensive performance studies.

The rest of the paper is organized as follows. We give a brief overview of the related work in section 2. We illustrate the design and implementation of the runtime environment in Section 3. In Section 4, we introduce three applications and evaluate their performance and we conclude the paper in Section 5.

II. RELATED WORK

A. High-Level Interface on Heterogeneous Devices

Other high level programming interface for GPUs include the following research projects: The Mars MapReduce framework [8] was developed for a single NVIDIA G80 GPU and the authors reported up to a 16x speedup over the 4-core CPU-based implementation for six common web mining applications. However, Mars cannot run on multiple GPUs and is not capable of running computation on GPUs and CPUs simultaneously. StarPU [9] provides a unified execution model to be used to parallelize numerical kernels on a multi-core CPU equipped with one GPU. OpenACC [10] is a current framework that provides OpenMP style syntax and can translate C or Fortran source code into a low-level codes, such as CUDA, or OpenCL. A growing number of vendors support the OpenACC standard. However, OpenACC cannot automatically run tasks on GPUs and CPUs simultaneously, which requires programmers' extra effort to make this happen.

Existing technologies for high-level programming interfaces for accelerators fall into two categories 1) using a domain specific language (DSL) library such as Mars, Qilin, or MAGMA [11] to compose low-level GPU and CPU codes and 2) compiling a sub-set of a high-level programming language into a low-level code run on GPU and CPU devices such as OpenACC, Accelerate [12], or Harlan [13]. The second group supports a richer control flow and significantly simplifies the programming development on different accelerator devices. However, this approach usually incurs extra overhead during compile and runtime, and makes it more difficult for developers to use low-level CUDA/OpenCL code in order to optimize application performance. To allow better access to optimization, we chose to use the DSL library technology in this paper.

B. Task Scheduling on Heterogeneous Devices

A number of studies exist that focus on task scheduling on distributed heterogeneous computing resources. The Grid community has developed various solutions among which the CoG Kits [36] provides a very flexible and simple workflow solution with its Karajan workflow scheduling engine and Coaster, a follow up to provide sustained jobs management facilities. GridWay [14] can split entire job into several sub-jobs, and schedule the sub-jobs to geographically distributed, heterogeneous resources. Falkon [15] uses a multi-level scheduling strategy in order to schedule massive, independent tasks on the HPC system. The first level is used to allocate resources, while the second level dispatches the tasks to their assigned resources.

Recently, several runtime systems have been created to schedule and execute SPMD jobs on GPUs and CPUs. The Qilin system can map SPMD computations onto GPUs and CPUs, and they reported good results of the DGEMM using an adaptive mapping strategy. Their activity is similar to ours in terms of scheduling SPMD tasks on GPUs and CPUs simultaneously; and their auto tuning scheduler needs to maintain a database in order to build a performance profiling model for the target application. MPC [16] uses a multi-granularities scheduling strategy in order to schedule inhomogeneous tasks on GPUs and CPUs. The Uintah [17] system implements the CPU and GPU tasks as C++ methods and models hybrid GPU and CPU tasks as DAG. The hybrid CPU-GPU scheduler assigns tasks to CPUs for processing when all of the GPU nodes are busy and there are CPU cores idle. They report a good speedup for radiation modeling applications on the GPU cluster. MAGMA is a collection of linear algebra libraries used for heterogeneous architectures. It models the linear algebra computation tasks as a DAG. The scheduler then schedules small, non-parallelizable linear algebra computations on the CPU, and larger, more parallelized ones, often Level 3 BLAS, on GPU.

Two main problems exist in regard to the above related work for task scheduling on GPUs and CPUs. One is the lack of generality which requires the adaptation of the approaches for specific domains. For example, it requires identifying regular memory access patterns, such as array based input data format; or it requires regular computation patterns, such as having iterative computation steps; or it focuses on a subset of parallel programs, such as linear algebra applications. The other problem is the extra performance overhead [18][19] when leveraging their proposed solutions. Some papers needed to run a set of small tests jobs on the heterogeneous devices, while some others need to maintain a database in order to store the performance profiling information for the target applications. The second problem may be not very serious because these papers claim that the benefit usually outweighs overhead their approaches introduced.

One of the main contributions of this paper is to propose an analytic model to be used for scheduling general SPMD computations on heterogeneous computation resources. Specifically, it can be used to calculate the work-load balance of SPMD tasks on heterogeneous resources and determine the task granularity for target applications. Our analytic model is derived from the roofline model, and it can be applied to a wide range of SPMD applications and hardware devices because the roofline model provides realistic expectations of performance and productivity for various given applications on many hardware devices. In addition, our model does not introduce extra performance overhead as there is no need to run test jobs.

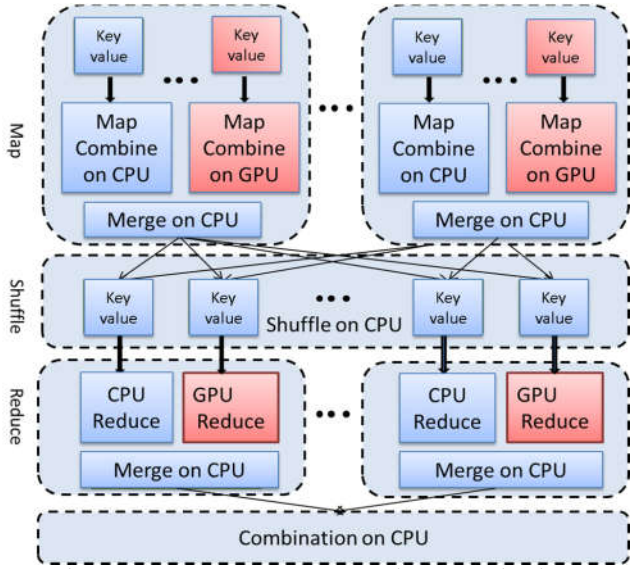


Figure 1: Heterogeneous MapReduce Based Scheme

III. RUNTIME ARCHITECTURE

Programmability and performance are two challenges faced when designing and implementing a parallel runtime system (PRS) on heterogeneous devices. In this section we will illustrate our design idea and the implementation details of our solution.

A. Design

1) Programming Model on Heterogeneous Resources

Figure 1 illustrates the heterogeneous MapReduce based scheme for co-processing SPMD computation on CPUs and GPUs. Map and Reduce are two user-implemented functions that present two main computation steps of the SPMD applications run on our PRS. Some SPMD applications only have the Map stage. Further, the heterogeneous MapReduce based interface supports both GPU and CPU implementations. Therefore, the developers can choose between the GPU or CPU versions, or both versions for the Map and Reduce functions. This design decision is based on a well-known agreement that applications should deliver different performance on different types of hardware resources. The process of tuning the application performance is specific to hardware resource and requires domain based knowledge when programming. Therefore, we allow developers implement either GPU or CPU versions of MapReduce functions.

By providing a high-level MapReduce based programming interface, we hide the implementation and optimization details of the underlying system from developers including the data movement across the levels of memory hierarchy, communication among the distributed nodes and scheduling tasks on heterogeneous devices. However, we leave developers the flexibility to write optimized MapReduce code for different devices if needed.

2) Work flow of Tasks

From the developer perspective, the workflow of a typical job run on our system consists of three main stages: job configuration stage, map stage, and reduce stage. In the job configuration stage, users specify the parameters for scheduling the tasks and sub-tasks. These parameters include the arithmetic intensity and performance parameters of hardware devices, such as DRAM and PCI-E bandwidth.

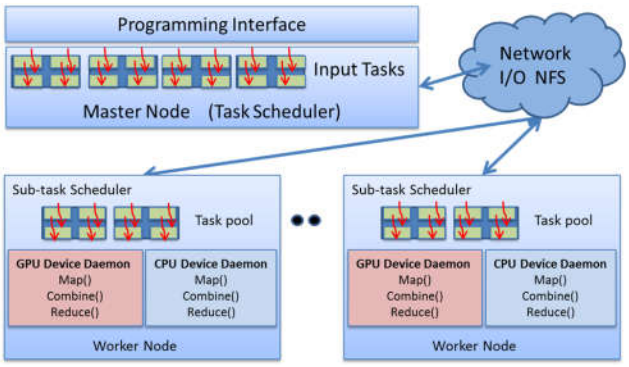


Figure 2: Parallel Runtime System Framework

In the *map stage*, the GPU and CPU backend utilities receive a set of input key/value pairs from the PRS scheduler, and invoke the user-implemented map() function in order to process the assigned key/value pairs. The map() function generates a set of intermediate key/value pairs in GPU and CPU memory, separately. The intermediate data located in GPU memory will be copied/sorted to/in CPU memory after all map tasks on local node are done. Then the PRS scheduler shuffles all intermediate key/value pairs across the cluster so that the pairs with the same key are stored consecutively in a bucket on the same node.

In the *reduce stage*, the PRS scheduler splits the data in buckets into some blocks, each of which is assigned to a Reduce task run on GPU or CPU. After the reduce computation is completed, the PRS merges the output of all of the Reduce tasks into the CPU memory so that these results can be further processed/viewed by the users.

B. Implementation

Figure 2 shows the PRS framework, which consists of the programming interfaces, task scheduler on master node, sub-task scheduler on worker node, the GPU/CPU device daemons on worker node and communication utility across the network.

1) API

The programming interfaces consist of the PRS provided API and user implemented API. Table 1 summarizes the user-implemented MapReduce based API. It provides three types of implementations, including gpu_host_mapreduce, gpu_device_mapreduce, and cpu_mapreduce.

cpu_mapreduce() is the C/C++ version of user-implemented MapReduce functions that run on CPUs.

gpu_host_mapreduce() is the user-implemented CUDA __host__ function for MapReduce functions that run on CPU, and it can invoke the CUDA __global__ function or other CUDA libraries to run on GPUs, such as cuBLAS.

gpu_device_mapreduce() is the user-implemented CUDA __device__ function of MapReduce functions that run on GPU directly, and it can invoke other __device__ functions to run on GPU as well.

Developers need implement at least one type of the map(), reduce(), and compare() functions; while the combiner() function is optional. For some applications, the source codes of cpu_mapreduce and gpu_device_mapreduce are same or similar

to each other. Paper [12][13] discuss their work on automatically transferring C++ code to CUDA code for different accelerator devices. However, this topic is not the focus of this paper.

2) Scheduler

We take the two-level scheduling strategy [15][22] consisting of the task scheduler in the master node and the sub-task scheduler on the worker node. The task scheduler first splits the input data into partitions, whose default number is twice that of the fat nodes. Then, the task scheduler sends out these partitions to sub-task schedulers on worker nodes. The sub-task scheduler further split the partition into blocks, which will be assigned to GPU and CPU device daemons for the further processing.

Table 1 User-implemented MapReduce based API

Type	Function
C/C++	void cpu_map(KEY *key, VAL *val, int keySize, …)
	void cpu_reduce(KEY *key, VAL *val, …)
	void cpu_combiner(KEY *KEY, VAL_Arr *val, …)
	Int cpu_comare(KEY *key1, VAL *val1, .., KEY …)
CUDA Device Func.	__device__ void gpu_device_map(KEY *key, …)
	__device__ void gpu__device_reduce(KEY *key, …)
	__device__ void gpu_device_combiner(KEY *key, ..)
	__device__ Int gpu_device_compare(KEY *key, …)
CUDA Host Func.	__host__ void gpu_host_map(KEY *key, …)
	__host__ void gpu_host_reduce(KEY *key, …)
	__host__ void gpu_host_combiner(KEY *key, …)
	__host__ void gpu_host_compare(KEY *key, …)

Table 2: Parameters of the Roofline Model

Parameters	Description
F_c/F_g	flop per second for target application on CPU/GPU, respectively
A_c/A_g	flops per byte for target application on CPU/GPU, respectively
B_dram	present the bandwidth of DRAM,
B_pcie	present the bandwidth of PCI-E,
P_c/P_g	present peak performance of CPU/GPU

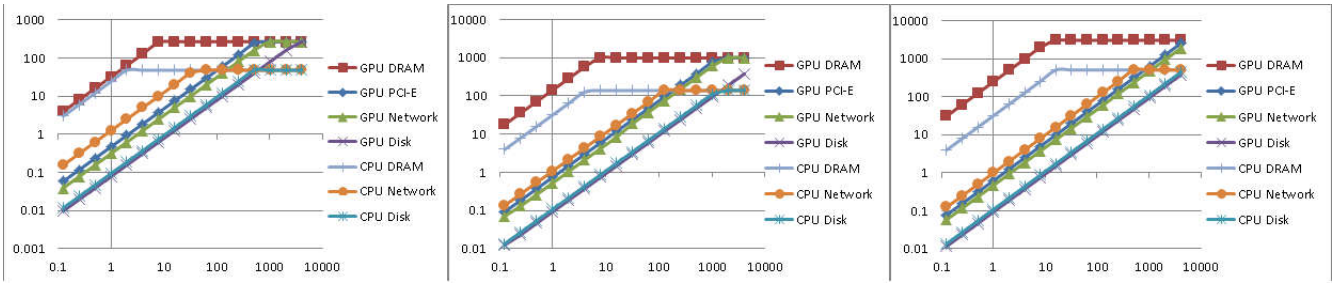
One problem for the scheduler is to determine how to schedule the tasks using the appropriate granularities on the GPUs and CPUs. There are two options to solve this problem. The first option is to have the sub-task scheduler split the partition into fixed size blocks, and then have the GPU and CPU devices daemons dynamically poll available blocks from sub-task scheduler when GPU or CPU resources are idle. This is called dynamically scheduling, however it is non-trivial work to find out the appropriate blocks sizes for both the GPUs and CPUs. Previous researches have proposed some solutions, but they usually introduce extra performance overhead or put some constrains on the target applications. The other option is to have the sub-task scheduler split the assigned partition into two parts to be assigned to the GPU and CPU device daemons. The workload distribution among GPU and CPU is calculated by our proposed analytical model. Then, the GPU and CPU device daemons would split assigned sub-partitions into sub-tasks with heterogeneous granularities suitable to run on the GPUs and CPUs, respectively. This process is the static scheduling approach. Our PRS provides for both scheduling strategies. We will make comparisons in following sections.

3) Roofline Model Based Scheduling

One highlighted feature of our PRS is the analytical model for guiding scheduling of the SPMD computations on the GPUs and CPUs clusters. We leverage the Roofline model in order to derive the analytical task scheduling model. We studied two parameters that affect performance of the task scheduling. One is the work load distribution fraction between the CPU and GPU, while the other is the task granularities on the CPU and GPU. The required system information for leveraging the Roofline model is summarized in Table 2.

a) Workload Distribution

We first analyses the workload distribution between the CPU and GPU for the SPMD computation on each node. Let T_c represent the overall run time on each node when only the CPUs are engaged in the computation. Let T_g represent the overall run time on each node when only the GPUs are used for the computation. Let T_{gc} represent job runtime when both CPUs and GPUs are engaged in the computation, the job run time is defined in Equation (1). The T_{c_p} is the CPU's time to process the fraction p of all of the input data, T_{g_p} is the GPU's processing time and data movement time for processing the remaining fraction, (1-p), of the input data. Let F_c and F_g represent the flop per second for the target application on the CPU and GPU. Let A_c and A_g represent the flops per byte for the target application on the CPU and GPU. Usually $A_c \sim = A_g$, but they could be different due to different algorithm implementations on the CPUs and GPUs. Let M represent the



(1) GPU:GT430, CPU:i5-2400 (2) GPU:C2050, CPU:x5660 (3) GPU:K20, CPU: Opteron 6212

Figure 3: Roofline model for fat nodes consist of different GPUs and CPUs. Y axis represents flop per second, X axis represents arithmetic intensity. The slope of left curve of ridge point represents the peak bandwidth when data go through DRAM, PCI-E, Network, and Disk. The right curve of ridge point represents the peak computation of GPU and CPU.

size of the input data for the target application. The parameter, p, to be tuned is the fraction of the input data that has to be processed by the CPU cores. We formulate equations to derive T_{gc} using the above defined variables.

$$T_{gc} = \max(T_{g\_p}, T_{c\_p}) \quad (1)$$

$$T_{g\_p} = (1-p) * M * A_g / F_g \quad (2)$$

$$T_{c\_p} = p * M * A_c / F_c \quad (3)$$

$$(1-p) * M * A_g / F_g = p * M * A_c / F_c \quad (4)$$

$$p = \frac{F_c}{F_g + F_c} \quad (if\ A_g \cong A_c) \quad (5)$$

According to the linear programming theory in math, when $T_{g\_p} \sim= T_{c\_p}$, T_{gc} gets the minimal value, and therefore we get the Equation (2) and Equation (3). Since F_c and F_g have to do with the other parameters defined in Table 1, we have the consequent deductions of Equation (4) and Equation (5) for F_c and F_g. Let B_dram represent the bandwidth of DRAM, and B_pcie represent the bandwidth of PCI-E. Let P_c represent the peak performance of the CPU and P_g represent the peak performance of the GPU. Let S represent the size of the input data. As shown in Figure 3, usually the GPU and CPU have drastically different ridge points. Let R_c represent the value of the Y axis of the ridge point for the CPU cores, and R_g represent the value of Y axis of the ridge point for the GPU. The slope of left part of the ridge point for the CPU cores is equal to DRAM bandwidth which is the peak performance divide by arithmetic intensity of application; the slope of right part of ridge point for GPU is equal to aggregated DRAM and PCI-E bandwidth, which is equal to the peak performance divide by arithmetic intensity of target application. When the arithmetic intensity of target application is less than the value of the X axis of the ridge point (as shown in Figure 3) and when the computation of application achieve the dynamic balance (data transfer rate equal to computation rate), then we get the first part of Equation (6) and (7) for the CPU and GPU, respectively. When the arithmetic intensity of the target application is larger than the value of the X axis of the ridge point, we get second part of Equation (6) and (7).

$$\begin{cases} \frac{A_c * S}{F_c} = \frac{S}{B\_dram} & (if\ A_c < A_{cr}) \\ F_c = P_c & (if\ A_c \geq A_{cr}) \end{cases} \quad (6)$$

$$\begin{cases} \frac{A_g * S}{F_g} = \frac{S}{B\_dram} + \frac{S}{B\_pcie} & (if\ A_g < A_{gr}) \\ F_g = P_g & (if\ A_g \geq A_{gr}) \end{cases} \quad (7)$$

Equations (6) and (7) can be used to derive the F_g and F_c values in different situations. Let A_{cr} and A_{gr} represent the values of the X axis of ridge points for CPU and GPU, respectively. Assuming all of the input data is located in CPU memory, program needs to load data from CPU to GPU memory through PCI-E bus. For this case, A_{cr} is usually smaller than A_{gr}, as shown in Figure 3. Thus, the arithmetic intensity of the target application can lie between three scopes: $A<A_{cr}$, $A_{cr}<A<A_{gr}$, and $A_{gr}<A$. When using Equation (6) and (7) to replace the F_g and F_c in Equation (5), we get Equation (8), which is used to derive the optimal work load distribution between the CPU and GPU for the target application.

Equation (8) is the core result of the proposed analytical model, and can be used to explain the work-load distribution among the CPUs and GPUs for various SPMD applications. When the target applications have low arithmetic intensity, the performance bottleneck is probably the bandwidth of the disk, network or DRAM. For these applications, such as word count, the CPU may provide better performance than the GPU. When the target applications have high arithmetic intensity, the performance bottleneck is the peak performance of the CPU and GPU, or the L2 cache. For these applications, such as DGEMM, the GPU has a better performance than the CPU. The similar observations have been reported in other papers [5][11]. However, our analytic model is the first mathematical model to precisely calculate the work load balance between the CPU and GPU, while it can be applied to applications with wide range of arithmetic intensities as shown in Figure 4.

Figure 4: the arithmetic intensity of different applications

$$\begin{cases} p = \frac{A_c * B_{dram}}{A_g * (\frac{1}{B_{pcie}} + \frac{1}{B_{dram}}) + A_c * B_{dram}} & if(A < A_{cr}) \\ p = \frac{P_c}{A_g * (\frac{1}{B_{dram}} + \frac{1}{B_{pcie}}) + P_c} & if(A_{cr} \leq A < A_{gr}) \\ p = \frac{P_c}{P_g + P_c} & if(A_{gr} \leq A) \end{cases} \quad (8)$$

The task scheduler on master node can use Equation (8) in order to split the input data among homogeneous or inhomogeneous fat nodes in cluster. The sub-task scheduler on the worker node can also use Equation (8) to split the data partition between the CPU and GPU. Equation (8) can also be extended by considering the bandwidth of the network in order to schedule communication intensive tasks. In this paper, we study the case where the fat nodes are of homogeneous computation capability; and we do not discuss communication intensive applications in the paper.

b) Task Granularity

The sub-task scheduler on the worker node can use the Equation (8) in order to indicate the splitting of the data partition between the GPU and CPU. However, the sub-partition of the data may be too large causing the CPU and GPU daemons to cause further splits. Intuitively, a small block size for the CPU can achieve good load balancing among multiple CPU cores; while a large block size for the GPU can minimize the impact of the data transfer latency on the execution time.

Paper [5][16][20] discuss their solutions for the task granularity issue. They use the parameter sweeping in order to discover the suitable task granularity, which is associated with extra performance overhead or they introduce some constrains on the target applications. These studies split the input partition into blocks whose numbers are several times those of the CPU cores. This splitting pattern can provide desirable results in both

the balanced workload distribution and low sub-task scheduling overhead. We adopt the same splitting pattern for scheduling sub-tasks on CPU cores in this paper.

It becomes complex to decide the task granularity for the GPUs. Serialized data transfers and GPU computations can either be PCI-E bus idle or GPU idle. The CUDA stream can simultaneously execute a kernel, while performing data transferring between the device and host memory. The Fermi architecture support only one hardware work queue; while the Kepler Hyper-Q model supports multiple hardware work queues. In addition, the stream approach can only improve application performance whose data transferring overhead is similar to computation overhead. Otherwise there will not be much overlap to hide the overhead.

For an application whose arithmetic intensity is a function of the input size, such as BLAS3, whose arithmetic intensity is O(N), we should increase the arithmetic intensity by increasing the input data size so as to saturate the peak performance of GPU. By using the Roofline model, we can calculate the minimal task block size necessary to achieve peak performance. Then, one task should be split into several sub-tasks and run on GPUs concurrently by launching multiple streams.

Let B_S represent the block size of the target application on the GPU. The overlap percentage between data transfer overhead and computation overhead can be deduced by using the Roofline model as show in Equation (9).

$$op = \frac{\left(\frac{B_S}{B_{dram}} + \frac{B_S}{B_{pcie}}\right)}{\left(\frac{B_S}{B_{dram}} + \frac{B_S}{B_{pcie}}\right) + \frac{B_S * A_g}{P_g}} \quad (9)$$

$$A_g = Fag(B_s) \quad (10)$$

$$MinBs = Fag^{-1}(A_{gr}) \quad (11)$$

Let A_{gr} represent the value of the X axis of the ridge point on the GPU. Let F_{ag} represent the arithmetic intensity function of a target application on the GPU. Then, the minimal block size necessary to cause the target application to achieve peak performance is the result of the inverse function F_{ag}^{-1} of A_{gr}.

As shown in Equation (11), $MinB_S$ is the theoretical minimal block size that should be used to saturate the peak performance of the GPU. One should notice that having a block size larger than the $MinB_S$ won't further increase the flops performance. Therefore, there are two requirements for leveraging multiple streams in CUDA: 1) the overlap percentage calculated by Equation (9) is larger than a certain threshold. 2) The data block size is larger than $MinB_S$ calculated by Equation (11).

C. Other Implementation Details

1) Threading Model

The PRS leverages the Pthreads in order to create CPU and GPU device daemons for managing tasks. It spawns one daemon thread for each GPU card and one daemon thread for all assigned CPU cores in the host. For example, if there are two GPUs and 12 CPU cores on one machine, then the PRS will spawn two daemon threads to be used for scheduling tasks on the GPUs and another daemon thread for scheduling tasks on the 12 CPU cores. The PRS also makes use of Pthreads to schedule tasks on CPU cores. Each thread runs one mapper or reducer on each CPU core. For gpu_device_mapreduce function, the PRS leverages the CUDA kernel threads to schedule tasks on GPU cores. It runs one mapper or reducer task per CUDA kernel thread. The default number of mappers and reducers in the PRS is several times larger than GPU cores in order to keep physical cores busy and hide latencies of the context switch. The gpu_host_mapreduce function is invoked by the GPU daemon thread, where the grid and block configuration of kernel threads is determined by programmer.

2) Region-based Memory Management

Region-based memory management [23] is a type of memory management in which each allocated object is assigned to a region, which, typically, is a single contiguous range of memory space. Two advantages exist to adopting this technology in a runtime framework. First, although the latest CUDA supports dynamically allocating the buffer in the GPU global memory using the malloc operation, the aggregated overhead of the malloc operations can degrade the performance if many small memory allocation requests exist. Instead of allocating many small memory buffers, the runtime library allocates a block of memory for each CPU or GPU thread, whose size should be big enough to serve many small memory allocations. When the block is filled, the runtime library will increase the buffer and copy the data to new buffer. The second advantage is that the collection of allocated objects in the region can be deallocated all at once.

3) Iterative Support

A set of iterative applications, such as Cmeans, exist that have loop invariant data during the iterations. It is expensive for the GPU program to copy these loop invariant data between the CPU and GPU memories over the iterations.

Paper [24][25][26] discuss the work of caching loop invariant data in the CPU memory over iterations. However, it will be difficult to do so because GPU need maintain the GPU context between iterations [27][35]. Therefore, instead of having every MapReduce tasks creating its own GPU context, we make GPU device daemon to be the only thread that communicate to GPU device. The GPU device daemon take in charge of read/write input/output data on behalf of MapReduce tasks. In addition that, GPU context switch is expensive. Such overhead is magnified when a large number of MapReduce tasks create their own GPU context. We adopt same strategy for funneled MapReduce tasks onto CPU cores.

IV. APPLICATIONS AND EVALUATION

This section evaluates the execution time using three sample applications on different experimental environments. Table 4 illustrates the configuration of the GPU and CPU devices used in the experiments. All of the NVIDIA GPU cards listed in Table 3 support computation capability at 2.x or above. The user implemented API are written in CUDA and C/C++, and compiled by using nvcc 4.2 and gcc 4.4.6, respectively.

A. Applications

1) C-means

The computational demands of the multivariate clustering grow rapidly; therefore clustering for large data sets is very time consuming on a single CPU. Fuzzy K- means (also called as C-means) [28][29] is an algorithm of clustering that allows one element to belong to two or more clusters with different probabilities. The C-means application is frequently used in multivariate clustering, such as flowcytometry clustering [30]. The algorithm is based on a minimization of the Equation 12. M is a real number greater than 1, while N is the number of elements. U_{ij} is the value of the membership of X_i in cluster C_j. $||X_i-C_j||$ is the norm expressing the similarity between the data point and the cluster center. The X_i is the ith data point, while C_j is the j^{th} cluster center. The fuzzy partitioning is performed using an iterative optimization of the objective function as shown above. Within each iteration, the algorithm updates the membership U_{ij} and the cluster centers the C_j using Equation 13 and Equation 14. The iteration will stop when $\max_{ij}\left\{\left|u_{ij}^{(k+1)}-u_{ij}^{(k)}\right|\right\}<\epsilon$ where 'e' is a termination criterion between 0 and 1, and 'k' is the iteration steps.

$$J_m = \sum_{i=1}^{n}\sum_{j=1}^{c} u_{ij}^{m}\left\|x_i - c_j\right\|^2 \qquad (12)$$

$$U_{ij} = \frac{1}{\sum_{k=1}^{c}\left(\frac{\|x_i-c_j\|}{\|x_i-c_k\|}\right)^{\frac{2}{m-1}}} \qquad (13)$$

$$C_i = \frac{\sum_{i=1}^{n} u_{ij}^{m} x_i}{\sum_{i=1}^{n} u_{ij}^{m}} \qquad (14)$$

We implemented a C-means MapReduce application using our PRS framework on GPU and CPU. The input matrices were copied into CPU and GPU memories in advance. The key object of the C-means MapReduce task contains the indices bound of input matrices, while the value object stores the pointers of input matrices in GPU or CPU memory. The event matrix is cached in GPU memory in order to avoid data staging overhead over iterations. The Map function calculates the distance and membership matrices, and then multiplies the distance matrix by the membership matrix in order to calculate the new cluster centers. The Reduce function aggregates partial cluster centers and calculates the final cluster centers.

We used one of Lymphocytes data set, which has 20054 points, 4 dimensions, and 5 clusters, to evaluate correctness of C-means implementation. The Lymphocytes data set has already been

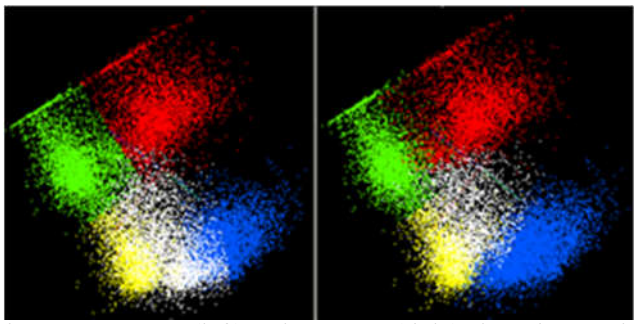
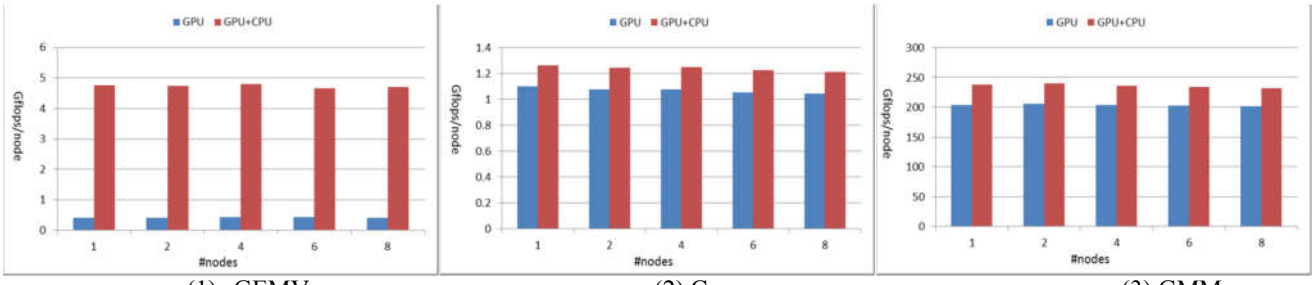


Figure 5: C-means (left) and K-means (right) clustering results of a Lymphocytes data set after a 3D projection.

studied in paper [30], and the clusters were calculated using Flame with finite mixture model. Figure 5 is the plot of C-means and K-means clustering results for Lymphocytes data set after project 4D data points into 3D data points by using algorithms[31][32]. The initial centers of C-means and K-means programs were picked up randomly, and we choose the best clustering results among several runs. We also compare results between C-means and K-means and DA[37][38] approaches [33] in terms of average width over clusters and points and clusters overlapping with standard Flame results. The DA approach provide the best quality of output results. The C-means results are a little better than Kmeans in the two metrics for the test data set. Table 3 shows the performance results in seconds of Cmeans using different runtime frameworks including MPI/GPU, PRS, and Mahout/CPU on 4 GPU nodes. The MPI/GPU and PRS use one GPU on each node. The MPI/CPU and Mahout/CPU use all CPU cores on each node, and they spawn two threads for each CPU core with hyper-threading enabled. The sample data set has 200k to 800k points, 100 dimensions, and 10 clusters. The results indicate that our PRS introduce some overhead during the computation as compared with MPI using one GPU per node solution, but it is faster than MPI using multiple CPUs per node and is two orders of magnitude faster than the Mahout (Apache Hadoop clustering) solution. We also have seen similar performance ratios for Kmeans application.

Table 3 Performance results of C-means with different runtimes

#points	200k	400k	800k
MPI/GPU	0.53 sec	0.945 sec	1.78 sec
PRS/GPU	2.31 sec	3.81 sec	5.31 sec
MPI/CPU	6.41 sec	12.58 sec	24.89 sec
Mahout/CPU	541.3 sec	563.1 sec	687.5 sec



(1) GEMV (2) C-means (3) GMM

Figure 6: weak scalability for GEMV, C-means, and GMM applications with up to 8 nodes on Delta. Y axis represents Gflops per node for each application. (1) GEMV, M=35000, N=10,000 per Node. (2) C-means, N=1000,000 per node , D=100, M=10. (3) GMM, N=100,000 per node, D=60, M=100. The red bard means only using GPUs as computation resources, while blue bar means using both GPUs and CPUs as computation resources.

2) GMM

The expectation maximization using a mixture model approach takes the data set as a sum of a mixture of multiple distinct events. Gaussians mixtures form probabilistic models composed of multiple distinct Gaussians distributions as clusters. Each cluster 'm' within a D dimensional data set can be characterized by the following parameters[28]:

N_m: the number of samples in the cluster
π_m : probability that a sample in data set belongs to the cluster
μ_m : a D dimensional mean
R_m: a DxD spectral covariance matrix

Assuming that there are N data points $y_1, y_2, \ldots, y_N$, then the probability that an event y_i belongs to a Gaussian distribution is given by the following equation

$$P(y_n|m,\theta) = \frac{exp\left\{-\frac{1}{2}(y_n-\mu_m)^t R_m^{-1}(y_n-\mu_m)\right\}}{(2\pi)^{D/2}|R_m|^{1/2}} \quad (15)$$

Neither the statistical parameters of the Gaussian Mixture Model, $\theta = (\pi, \mu, R)$, nor the membership of events to clusters are known. An algorithm must be employed to deal with this lack of information. The expectation maximization is a statistical method for performance likelihood estimation with incomplete data. The objective of the algorithm is to estimate θ, the parameters for each cluster.

3) GEMV

The BLAS are a set of basic linear algebra subprograms that perform vector-vector, matrix-vector, and matrix-matrix operations. The matrix-vector multiplication is embedded in many algorithms for solving a wide variety of problems. There are three straightforward ways to decompose a MxN matrix A: row wise block striping, column wise block striping and the checkerboard block decomposition. In this paper, we use row wise block-striped decomposition to parallel matrix-vector multiplication. We associate a primitive map task with each row of the matrix A. Vectors B and C are replicated among the map tasks so the memory can be allocated for the entire vectors on each compute node. It follows that the map task has all the elements required to compute. Once this is done, reduce task can concatenate the pieces of vector C into a complete vector.

For many programmers, the key to a good performance of numerical scientific applications is still linked to the availability of high-performance libraries available for GPUs and CPUs, e.g., Nvidia's cuBLAS [2], Intel MKL, and open source MAGMA library. In the experiment, we leveraged the CUDA cuBLAS and Intel MKL library to perform the GEMV computation on GPU and CPU on each node. This strategy simplified our programming work so that we could focus on evaluating the proposed scheduling strategy.

B. Performance Evaluation

Figure 6 show the weak scalability of GEMV, C-means, and GMM using our framework. In this case the problem size (workload) assigned to each node stays constant. The GPU version only uses one GPU per node during computation, while GPU+CPU version uses one GPU and all available CPU cores on same node during computation. The value X in Table 4 means X percentage of the work load is assigned to CPU, while the remain (1-X) percentage of work is assigned to GPU.

Table 4: Hardware Configuration

Machine Name	Future Grid Delta	IU BigRed2
GPU Type	C2070	K20
GPUs/Node	2	1
Memory/GPU	6 GB	5 GB
Cores/GPU	448 Cores	2496 Cores
CPU Type	Intel Xeon 5660	AMD Opteron 6212
Cores/CPU	12 Cores	32 Cores
Memory/CPU	192 GB	62 GB

Table 5: Work Load Distribution among GPU and CPU of Three Applications using Our Framework

Apps	GEMV	C-means	GMM
Arithmetic intensity	2	5*M (M = 100)	11*M*D (M=10,D= 60)
p calculated by Equation (8)	97.3%	11.2%	11.2%
p calculated by app profiling	90.8%	11.9%	13.1%

For GEMV, it shows the Gflops/node performance gap between GPU and CPU is large, i.e., CPU+GPU version is 10 times faster than GPU only version. This is because GEMV has low arithmetic density. The data staging overhead between GPU and CPU cost more than 90% of its overall overhead. We calculate the work load distribution proportion of GEMV among GPU and CPU on Delta node by using equation (8) of analytical model. For C-means, it shows the linear scaling is achieved as the Gflops per node stays constant while the workload is increased in direct proportion to the number of nodes. In addition, the GPU+CPU version is 1.3 times faster than GPU only version. The peak performance per node decrease by 5.5% when using 8 compute nodes, which is due to the increasing overhead in global reduction stage of the parallel C-means algorithm. For GMM, it shows similar linear weak scaling when number of points per node is fixed. But peak performance of GMM is much larger than that of C-means, as it has larger arithmetic intensity O(M*D), as compared with O(M) for C-means. Given C-means and GMM are of iterative computationsteps, we didn't timing the data staging overhead between GPU and CPU at the beginning step and end step of computation. This is because these overhead are one-off overhead[34], which will be amortized when number of iterations is large. In other words, the average arithmetic intensity of C-means and GMM depend on the bandwidth of DRAM and peak performance of GPU, rather than bandwidth of PCI-E bus.

We also study the work load balance issue of our PRS implementation on GPUs and CPUs clusters. Table 5 summarizes the work load distribution between GPU and CPU of three applications using our PRS framework on the Delta node illustrated in Table 4. The work load distribution proportions, p values, between GPU and CPU are calculated by using Equation (8). The parameters of bandwidth of DRAM, PCI-E bus, and peak performance of GPU and CPU are shown in Figure 3 (1). Another set of p values calculated by measuring

the real peak performance of the three applications using GPU version and GPU+CPU version, respectively. As it shown in Table 5, applications with low arithmetic intensity, such as GEMV, should assign more work load onto the CPU; while applications with high arithmetic intensity should assign more work load onto the GPU. The error between p values calculated by using Equation (8) and the ones by application profiling is less than 10% for the three applications in Table 5.

I. SUMMARY AND CONCLUSION

This paper introduced a PRS framework for running SPMD computation on GPU and CPU cluster. The paper is proposing an analytical model that is used to automatically scheduling SPMD computation on GPU and CPU cluster. The analytical model is derived from roofline model, and therefore, it can be applied to a wide range of SPMD applications and hardware devices. The significant contribution of analytic model is that it can precisely calculate the balanced work load distribution between the CPU and GPU, while be applied to applications with wide range of arithmetic intensities. Experimental results of GEMV, C-means, and GMM indicate that using all CPU cores increase the GPU performance by 1011.8%, 11.56%, and 15.4% respectively. The error between the real optimal work load distribution proportion and theoretical one is less than 10%.

For SPMD applications, such as PDEs, FFT whose arithmetic intensities are in the middle range as shown in Figure 4, using our PRS framework can increase resource utilization of heterogeneous devices, and decrease job running time because both GPU and CPU can make the non-trivial contribution to overall computation, and because the workload is evenly distributed between GPU and CPU by the PRS.

The future work of our PRS framework could be: a) Extend the proposed analytical model by considering the network bandwidth issue. b) Extend the framework to other backend or accelerators, such as OpenCL, MIC. c) Applying the analytical model to heterogeneous fat nodes.

Acknowledgements

The authors thank Andrew Pangborn for the original C-means and GMM CUDA programs. We also thank Jerome Mitchell and Adnan Ozsoy for the help about running experiments on Delta nodes. We also thank Judy Qiu for the suggestions on MapReduce design and implementation. At last we thank Jong Choi for plotting figure 5 in the paper. The work in this paper was supported by FutureGrid project funded by National Science Foundation (NSF) under Grant No. 0910812.

REFERENCES

[1] Michael Garland, David Kirk, Understanding throughput-oriented architectures, COMMUNICATIONS of the ACM 2010.

[2] NVIDIA Inc, CUDA C Programming Guide, http://www.nvidia.com/ October 2012.

[3] MUNSHI, A. "OpenCL Parallel Computing on the GPU and CPU", In ACM SIGGRAPH, Los Angeles, California, USA, August 2008.

[4] Dean, J. and S. Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters". Sixth Symposium on Operating Systems Design and Implementation: San Francisco, CA , 2004.

[5] Chi-Keung Luk, Sunpyo Hong, Hyesoon Kim, "Qilin: Exploting Parallelism on Heterogeneous Mulitprocessors with Adaptive Mapping", MICRO'09, New York, NY, 2009.

[6] Chun-Yu Shei, Pushkar Ratnalikar and Arun Chauhan. "Automating GPU Computing in MATLAB". In Proceedings of the International Conference on Supercomputing (ICS), pages 245–254, 2011.

[7] Samuel Williams, Andrew Waterman, David Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architecture", Communications of the ACM , Volume 52 Issue 4, New York, NY, USA , April 2009.

[8] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. "Mars: A MapReduce Framework on Graphics Processors". PACT 2008, Toronto, CANADA, 2008.

[9] Ludovic Courtes and Nathalie Furmento, "StarPU: Hybrid CPU/GPU Task Programming, C Extensions and MPI Support", ComPAS, Grenoble, January 2013.

[10] OpenACC www.openacc-standard.org

[11] ICL Innovative Computing Laboratory, "MAGMA: Matrix Algebra on GPU and Multicore Architectures", SC12, Salt Lake City, Utah, 2012

[12] Manuel M. T. Chakravartyy Gabriele Kellery Sean Leezy Trevor L. McDonelly Vinod Groverz, "Accelerating Haskell Array Codes with Multicore GPUs". DAMP'11 Austin, Texas, USA, 2011.

[13] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. "Declarative Parallel Programming for GPUs. In Proceedings of the International Conference on Parallel Computing" (ParCo), September 2011.

[14] GridWay, Metascheduling Technologies for the Grid, www.GridWay.org, September 2009.

[15] Ioan Raicu, Yong Zhao, "Falkon: a Fast and Light-weight tasK executiON framework for Grid Environments", IEEE/ACM SuperComputing 2007, November 15th, Reno, Nevada, USA, 2007.

[16] Patrick Carribault, Marc Pérache and Hervé Jourdren "Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC", IWOMP 2010, Aprajon France, 2010.

[17] Alan Humphrey, Qingyu Meng, Martin Berzins, Todd Harman, "Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System", XSEDE'12, Chicago Illinois, USA, July 2012.

[18] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, Toshitsugu Yuba, "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment", VECPAR'06, Rio de Janeiro, Brazil, 2006.

[19] Linchuan Chen, Xin Huo, Gagan Agrawal, "Accelerating MapReduce on a Coupled CPU-GPU Architecture", SC12, Salt Lake City, Utah, USA, Nov, 2012.

[20] Z Guo, M Pierce, G Fox, M Zhou, Automatic Task Re-organization in MapReduce , CLUSTER2011, Austin Texas , September 2011.

[21] T.R.Vignesh, M. Wenjing "Compiler and runtime support for enabling generalized reduction computation on heterogenesou paralle configuration" ICS'10; Proceedings of the 24ACM International Conference on Supercomputing. New Orleans, Louisiana, 2010

[22] Li Hui, Yu Huashan, Li Xiaoming. A lightweight execution framework for massive independent tasks. Many-Task Computing on Grids and Supercomputers. MTAGS 2008. Austin, Texas. Nov 2008.

[23] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, SOFTWARE-PRACTICE AND EXPERIENCE, Jan, 1990.

[24] J.Ekanayake, H.Li, et al. (2010). Twister: A Runtime for iterative MapReduce. Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference. Chicago, Illinois, ACM. June, 2010.

[25] Bingjing Zhang, Yang Ruan, Tak-Lon Wu, Judy Qiu, Adam Hughes, Geoffrey Fox, Applying Twister to Scientific Applications, in Proceedings of IEEE CloudCom 2010 Conference (CloudCom 2010), Indianapolis, November 30-December 3, 2010, ISBN: 978-0-7695-4302-4, pp. 25-32

[26] Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu, Geoffrey Fox, "Design Patterns for Scientific Applications in DryadLINQ CTP", DataCloud-SC11, Nov 2011

[27] Thilina Gunarathne, Bimalee Salpitikorala, Arun Chauhan and Geoffrey Fox. Optimizing OpenCL Kernels for Iterative Statistical Algorithms on GPUs. In Proceedings of the Second International Workshop on GPUs and Scientific Applications (GPUScA), Galveston Island, TX. 2011.

[28] Andrew Pangborn, Gregor von Laszewski, James Cavenaugh, Muhammad Shaaban, Roy Melton, Scalable Data Clustering using GPUs cluster, Thesis. Computer Engineering, Rochester Institue of Technology, 2009.

[29] Andrew Pangborn, Scalable Data Clustering with GPUs, Thesis, Computer Engineering, Rochester Institue of Technology, 2010.

[30] FLAME DataSet, Gene Pattern, http://www.broadinstitute.org/cancer/software/genepattern/modules/FLAME/published_data, 2009.

[31] J. Choi, S. Bae, X. Qiu, and G. Fox, "High Performance Dimension Reduction and Visualization for Large High-dimensional Data Analysis," proceedings of CCGRID, 2010.

[32] S.-H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox, "Dimension reduction and visualization of large high-dimensional data via interpolation," in HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, (New York, NY, USA), pp. 203–214, ACM, 2010.

[33] Y Ruan, S Ekanayake, M Rho, H Tang, SH Bae, J Qiu , DACIDR: deterministic annealed clustering with interpolative dimension reduction using a large collection of 16S rRNA sequences, Proceedings of the ACM Conference on Bioinformatics, 2012.

[34] Hui Li, Geoffrey Fox, Judy Qiu, "Performance Model for Parallel Matrix Multiplication with Dryad: Dataflow Graph Runtime", BigDataMR-12, Nov 2012

[35] Jon Currey, Simon Baker, and Christopher J. Rossbach, Supporting Iteration in a Heterogeneous Dataflow Engine, in SFMA 2013, The 3rd Workshop on Systems for Future Multicore Architectures, 14 April 2013

[36] G. von Laszewski, Workflow Concepts of the Java CoG Kit, in Journal of Grid Computing in Vol 3, Issue 3-4, pp. 239-258, 2005, http://dx.doi.org/10.1007/s10723-005-9013-5,, http://cyberaide.googlecode.com/svn/trunk/papers/anl/vonLaszewski-workflow-taylor-anl.pdf

[37] Geoffrey Fox , D. R. Mani, Saumyadipta Pyne Parallel Deterministic Annealing Clustering and its Application to LC-MS Data Analysis Proceedings of 2013 IEEE International Conference on Big Data October 6-9, 2013, Santa Clara, CA, USA

[38] Geoffrey Fox, Robust Scalable Visualized Clustering in Vector and non Vector Semimetric Spaces, To be published in Parallel Processing Letters 2013