Co-processing SPMD Computation on GPUs and CPUs on Shared Memory System

School of Informatics and Computing, Pervasive Technology Institute
Indiana University Bloomington

Hui Li, Geoffrey Fox, Gregor Laszewski, Zhenhua Guo, Judy Qiu

***Abstract*— Heterogeneous parallel system 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 executing SPMD computations on heterogeneous architectures simultaneously. In order to meet these challenges, we implemented a MapReduce runtime system to co-process SPMD job on GPUs and CPUs on shared memory system. We are proposing a heterogeneous MapReduce programming interface for the developer and leverage the two-level scheduling approach in order to efficiently schedule tasks with heterogeneous granularities on the GPUs and CPUs. Experimental results of C-means clustering, matrix multiplication and word count indicate that using all CPU cores increase the GPU performance by 11.5%, 5.1%, and 41.9% respectively.**

Keyword: MapReduce, 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 low-end HPC systems, a small sized hybrid cluster with only tens of GPU cards can provide performance over one petaflops, while the same scale CPU cluster can provide one teraflops of peak performance. In high-end HPC system, Tianhe-1A, a hybrid cluster using Intel CPUs and NVIDA GPUs became the fastest supercomputer in 2010.

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. Now this pattern has stalled, as such CPUs are integrating more cores within the processor. 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) contains massively 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.

The NVIDIA's CUDA [2] and Khronos Group OpenCL [3] are the current and most widely used GPU programming tools. Both CUDA and OpenCL claim to translate source code into binaries run on CPUs and GPUs. However, these generated binary codes cannot run on CPUs and GPUs simultaneously. The CPU cores are idle while doing GPU computation, or vice versa. Figure 1 shows programmability challenges of how to map the SPMD computation to the CPUs and GPUs simultaneously. NVIDIA use the terminology SIMT, "Single Instruction, Multiple Threads", to present the programming model on GPU. The SIMT can be considered a hybrid between vector processing and hardware threads. To write SIMT codes, CUDA developers need calculate the thread indices and carefully arrange memory access pattern. Our work bridges this gap between SIMT and SPMD by providing a high-level MapReduce programming interface to developers and hides the implementation details from developers. The SPMD style computations are already presented on CPUs by using many programming tools such as Pthreads, and OpenMP. To co-process SPMD computation on GPUs and CPUs, we need find the intersection of GPU and CPU SPMD applications first. The applications should have enough computation so as to keep GPUs busy. In addition, their input data size should be fit in both GPU and CPU memory. Most important, these SPMD applications should be those whose major computation can be partitioned into parallel sub-tasks with arbitrary granularities because the proper task processing granularities on GPU and CPU are different. Some applications in linear algebra, data mining can meet above requirements, such as DGEMM, FFT, Kmeans, SVM, which have had the implementations for both CPU and GPU.

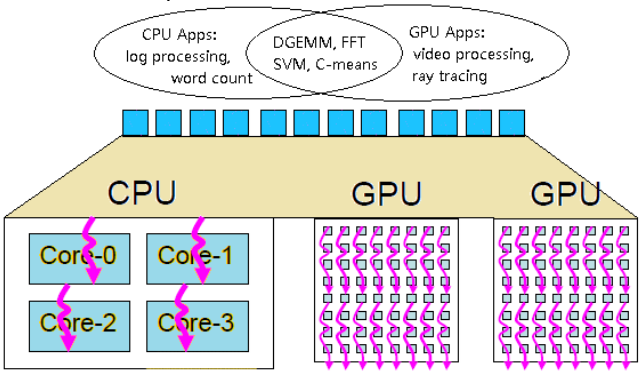

Figure 1: Co-processing SPMD on CPUs and GPUs

We also need a programming model that can present the SPMD computation on both GPUs and CPUs. The MapReduce [4] programming model originated at Google, and it has been successfully applied to large classes of

SPMD applications on shared memory, and distributed memory systems. What's more, the recent researches proved that executing MapReduce computation on GPU is not only feasible and but also practice. Developing the MapReduce program is easy because the MapReduce runtime hides the implementation details such as data movement, task scheduling and work load balance from the developers. However, most state-of-the-art MapReduce implementations are designed to run tasks solely on CPUs or GPUs, rather than on CPUs and GPUs simultaneously, which is one of the differences between our researches with other related work.

Performance is another challenge faced when running programs on GPUs and CPUs simultaneously as they have different types of instruction sets. In addition, CPUs prefer to process coarse granularity tasks rather than the massive fine grained tasks favored by GPU cards. These challenges require software to coordinate the parallelism properly in order to achieve good performance. Even if the SPMD jobs can be split into tasks with arbitrary granularity, workload balance, task scheduling overheads issues still need to be elaborately solved. In addition, the process of mapping the SPMD computation to GPUs and CPUs should be as automated as possible due to the various types of hardware configurations of the CPU and GPU devices.

We have implemented a parallel runtime system, with the code name Panda, to co-process SPMD computation on modern NVIDIA Fermi GPUs and Intel Xeon CPUs on shared memory system. We are proposing a heterogeneous GPU/CPU parallel programming model for mapping SPMD computation with heterogeneous task granularity on GPU and CPUs devices. We implemented the two-level-scheduler [5] to efficiently schedule tasks with heterogeneous granularity on GPUs and CPUs. We also implemented an auto tuning component to adaptively assign the proper computation workload to GPUs and CPUs devices. In order to evaluate Panda runtime system, we implemented three applications including C-means clustering, matrix-matrix multiplication and word count using Panda. We make the comprehensive performance comparison between Panda with CUDA, Mars [6], Phoenix [7], OpenACC [8] and MAGMA.

The rest of the paper is organized as follows. We gave a brief overview of the related work in section 2. We illustrate the design and implementation of Panda in section 3. In section 4, we introduce three Panda applications and evaluate their performance. We make the conclusion in section 5.

II. RELATED WORK

A. *High-Level Interface on GPUs*

The early MapReduce library for GPUs included the following research projects: In 2008, Bingsheng published the seminal research paper on Mars. The Mars MapReduce framework was developed for a single Nvidia G80 GPU and they reported up to 16x speedup over the 4-core CPU-based implementation for six common web mining applications. However, Mars cannot run on multiple GPUs and it does not support to run on GPU and CPU simultaneously. This project was the first to show the GPU potential for MapReduce.

OpenACC is a state-of-the-art framework that provides OpenMP style syntax and can translate C or Fortran source code into a low-level code, such as CUDA, or OpenCL. A growing number of vendors support OpenACC as developers can easily reuse their existing codes. However, OpenACC cannot run tasks on GPUs and CPUs simultaneously. In addition, if the parallel algorithm of application is complex, OpenACC may not perform well as expect.

Existing technologies for high-level programming interfaces for accelerators falling into two categories: 1) using a library such as Mars, Qilin [9] to compose low-level GPU and CPU codes. 2) compiling a sub-set of a high-level programming such as OpenACC [8], Accelerate [10], and Harlan [17] langue into a low-level code that is run on GPU and CPU devices. The second technology supports richer control flow and significantly simplifies the programming development on different accelerator devices. However, this approach usually incurs the extra overhead during compiling time and runtime, and it prevents developers from using low-level CUDA/OpenCL code to optimize application performance themselves. Panda follows the idea in the first category. However, instead of providing the unified API, Panda are proposing the heterogeneous GPU/CPU parallel programming API to compose low-level code that run on GPUs and CPUs.

B. *Scheduling on Heterogeneous Devices*

There are large numbers of studies about task scheduling on distributed heterogeneous computing resources. GridWay presented by the Globus Alliance can split entire job into several sub-jobs, and assign each sub-job to one distributed resource manager for the further processing. Falkon [11] use a multi-level scheduling strategy to schedule massive, independent tasks on the HPC system. The first level was used to allocate resources, while the second level dispatched the tasks to their assigned resources.

Recently, there are also several runtime systems can 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 DGEMM using adaptive mapping strategy. Their job is similar to us in terms of scheduling SPMD tasks on GPU and CPU simultaneously; however their auto tuning scheduler need maintains a database to build a performance model in order to schedule proper workload to GPUs and CPUs. This approach usually works well for applications which have regular computation and memory access pattern. Our auto tuning scheduler make the scheduling decision based on the performance results of a set of small testing jobs. It does not need build performance model and therefore works for the larger problem classes and more heterogeneous devices. The Uintah system [13] implements the CPU and GPU tasks as C++ methods and it models hybrid GPU and CPU tasks as DAG. The hybrid CPU-GPU scheduler assigns tasks to CPUs for processing when all GPU nodes are busy and there are CPU cores idle. They reported good speedup performance of radiation modeling applications on GPU cluster. The Panda framework leveraged two-level scheduling strategy where the CPU tasks scheduler and GPU

tasks scheduler are embedded within first level scheduler. The two-level scheduler assigns tasks to both GPUs and CPUs in order to increase resource utilization and decrease job run time. MAGMA [14][15] is a collection of linear algebra libraries for heterogeneous architectures. It models the linear algebra computation tasks as a DAG. The scheduler schedules small non-parallelizable linear algebra computation on the CPU, and schedules larger more parallelized ones, often Level 3 BLAS, on the GPU. The Panda scheduler is designed for scheduling flat MapReduce style computation on GPUs and CPUs.

III. PANDA ARCHITECTURE

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

A. Design

1) Heterogeneous MapReduceBased Scheme

Figure 2 illustrates the heterogeneous MapReduce based scheme for co-processing SPMD computation on CPUs and GPUs on shared memory system. The Map and Reduce functions present two computation steps of SPMD applications. Some SPMD applications only have the Map stage. Further, the heterogeneous MapReduce based scheme support both GPU and CPU interface. The developers need implement at least one of GPU and CPU versions, or both GPU and CPU versions for Map and Reduce functions. This design decision is based on two reasons. First, the proper granularities of SPMD tasks on GPU and CPU are different from each other. CPU prefers to process coarser granularity than GPU does [9][12]. Thus, developers may implement different MapReduce functions to efficiently process tasks with heterogeneous granularities. The second reason is based on a well-known agreement that different applications favor different type of hardware resources. Therefore, instead of providing the unique MapReduce programming interface, we allow developers implement either the GPU or CPU versions of MapReduce functions based on above two reasons.

By providing high-level MapReduce based programming interface, Panda hides runtime implementation and optimization details from developers such as data movement across levels of memory hierarchy, scheduling tasks over heterogeneous devices, and management of multiple GPU contexts. We also provide several optimization strategies in runtime level include auto tuning, region based memory management, iterative support, local combining. We expect to leave developers the flexibility to write optimized MapReduce code for different devices, while hiding the implementation details from them as much as possible.

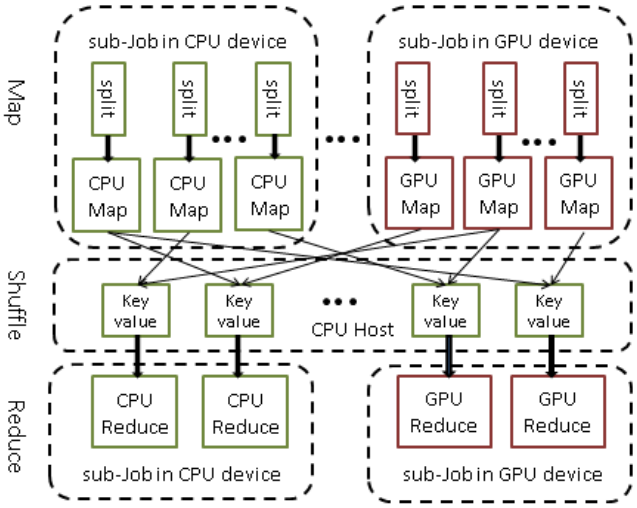

Figure 2: Panda Heterogeneous MapReduce Based Scheme

2) Multi-levele Scheduling

The innovation of our work is to run heterogeneous MapReduce tasks on GPUs and CPUs simultaneously. Panda enable above function by providing the two-level scheduling strategy [11][18]. The first-level scheduler splits the SPMD job into several sub-jobs, each of which was assigned to one GPU or multi-core CPUs. The second-level schedulers include the CPU tasks scheduler and GPU tasks scheduler, which are embedded within first-level scheduler. They further split the sub-jobs into many tasks to be run on the assigned GPU and CPU resources.

The second design challenge is to determine how to properly map SPMD computations onto GPUs and CPUs because they favor different task granularities. Several approaches exist by which to schedule MapReduce tasks on GPUs and CPUs. For example, one can construct the homogenous MapReduce tasks with same data block sizes and then assign a different number of MapReduce tasks to the CPUs and GPUs. The problem with this approach is that task granularity could be too fine for the CPUs or too coarse for GPUs, either case can lead to a workload imbalance issue during computations. Although one can adjust task granularity by group/split tasks after the initial task partitioning, such an action introduce extra programming efforts from developers and increase the performance overhead. With our two-level scheduling approach, because the CPU scheduler and GPU scheduler are independent from each other, they can split the sub-jobs into tasks with heterogeneous granularities that are suitable to run on GPU and CPU respectively. As compared with other approaches, this approach is simple and can be extended to other devices.

B. Implementation

The runtime framework consists of three components: programming interfaces, a job scheduler, and the backend utility as shown in Figure 3. The programming interfaces consist of the framework provided API and user implemented API. The job scheduler implemented the two-

level scheduling strategy discussed in Section 3.1.2. The backend utility is implemented with C/C++ and CUDA language and it is used to run the MapReduce tasks on GPUs and CPUs. Currently, it supports the CUDA based GPU device and multi core CPUs as the backend.

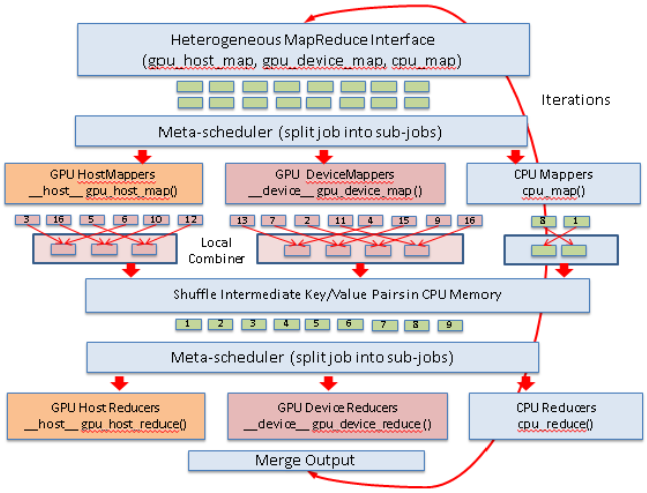

Figure 3: Runtime Framework

From developer perspective, the work flow of a typical Panda job consists of three main stages: job configuration, Map, and Reduce. In the job configuration stage, users can specify the parameters to be used to configure sub-jobs and tasks. These parameters include the number of CPU and GPU resources; the type and number of Map and Reduce tasks that used in the computation. The Panda framework will split the entire job into several sub-jobs whose number is equal to total number of GPU and CPU devices. The workload distribution ratio between these sub-jobs can be specified by the users or determined by an auto tuning mechanism provided by the Panda framework. Developers can further divide sub-jobs into MapReduce tasks with different granularities that run on different devices. The granularity of these tasks depends on several factors, including 1) the number of MapReduce tasks specified by the users; 2) the computation capability features of the devices, such as the number of cores, and memory size and 3) the computation features of the applications such as being computation or memory intensive.

In the Map stage, the GPU and CPU backend utilities get a set of input keyvalue pairs from Panda two-level-scheduler, and invoke the map() functions that implemented by developers to process assigned keyvalue pairs. The map() functions generate a set of intermediate keyvalue pairs in GPU and CPU memory separately, which will be copied to CPU memory after all map tasks are complete. Then Panda shuffle all these intermediate key/value pairs in CPU memory so that the pairs with the same key are stored consecutively. In the Reduce stage, the two-level-scheduler divides the shuffled intermediate key/value pairs into several chunks, each of which will be assigned to GPU or CPU backend utility. Then the backend utility invokes the reduce() functions to process assigned keyvalue pairs. After the reduce computation is complete, Panda copies the results of all of the Reduce tasks in the CPUs and GPUs into the memory of the CPUs so that these results can be further processed by the users.

C. API

Similar to the existing MapReduce framework such as Mars and Phoenix, Panda has two kinds of APIs: user-implemented APIs, which the users should implement; and system-provided APIs, which the users can use as library calls. Table 1 illustrates the three types of MapReduce based API supported by Panda runtime framework. The three types of MapReduce based APIs include: gpu_host_mapreduce, gpu_device_mapreduce, and cpu_mapreduce. For example, gpu_host_map(..), gpu_device_map() and cpu_map(..) are the three types of the map function. The gpu_host_map() is a user defined CUDA host function that invoke CUDA global function or other CUDA based libraries such as MAGMA. gpu_device_map() is a user defined CUDA kernel function that perform CUDA code or invoke other kernel functions. cpu_map() is the user defined C/C++ function. Users need implement at least one type of map(), reduce(), and compare() functions; the combiner() function is optional. For most applications, the source code for the CPU and GPU versions of the user-implemented APIs are similar due to two reasons. The first reason is that CUDA support the C/C++ syntax and grammar; one can compile C/C++ source code with nvcc. The second reason is that the Panda framework hides some GPU specific work for the users, such as data staging between the CPU and GPU memory; threads and blocks indices calculation.

Table 1: Panda MapReduce Based API:

Function Type	Function	Illustration
C/C++ Function	void cpu_map(KEY *key, VAL *val, int keySize, ..)	CPU Map function using C/C++
	void cpu_reduce(KEY *key, VAL *val, int keySize, …)	CPU Reduce function using C/C++
	void cpu_combiner(KEY *KEY, VAL_Arr *val, int keySize, int valSize)	CPU Combiner function using C/C++. Used for partial aggregation.
	Int cpu_comare(KEY *key1, VAL *val1, .., KEY *key2, VAL *val2, int KeySize1, int KeySize2, int valSize1,…)	CPU Compare function using C/C++. Used for shuffling key/value pairs.
CUDA Device Function	__device__ void gpu_device_map(KEY *key, …)	GPU Map using CUDA device function.
	__device__ void gpu__device_reduce(KEY *key, …)	GPU Reduce using CUDA device function.
	__device__ void gpu_device_combiner(KEY *key, …)	GPU Combiner using CUDA device function.
	__device__ Int gpu_device_compare(KEY *key, …)	GPU Compare using CUDA device function.
CUDA Host Function	__host__ void gpu_host_map(KEY *key, …)	GPU Map using CUDA host function.
	__host__ void gpu_host_reduce(KEY *key, …)	GPU Reduce CUDA host function.
	__host__ void gpu_host_combiner(KEY *key, …)	GPU Combiner CUDA host function.
	__host__ void gpu_host_compare(KEY *key, …)	GPU Compare CUDA host function.

```
A.
void cpu_reduce(void *KEY, val_t *VAL…){
int count = 0;
for (int i=0;i<valCount;i++){
        count += *(int *)(VAL[i].val);
}//calcualte word occurence
EmitCPUReduceOutput(KEY, &count, keySize, …);
}//cpu version of reduce function

B.
__device__ void gpu_device_reduce(void *KEY){
int count = 0;
for (int i=0;i<valCount;i++){
        count += *(int *)(VAL[i].val);
}// calcualte word occurence
EmitGPUDeviceReduceOutput(…);
}//gpu version of reduce function

C.
__host__ void gpu_host_map(…){
dgemm_kernel<<<grid, threads>>>(..);
EmitGPUHostMapOutput(KEY, keySize, …);
}// gpu host function invoke MAGMA code
```

Figure 4: User Implemented cpu_reduce and gpu_device_reduce functions for word count and gpu_host_map function for matrix multiplication.

Figure 4 show the user implemented cpu_reduce() and gpu_device_reduce() functions for the word count application and gpu_host_map() function for the matrix multiplication application, which invoke MAGMA as library. As shown in Figure 4, Panda simplified the development of the SPMD MapReduce program on the GPU and CPU devices. Nevertheless, Panda leave users the flexibility of either making advanced optimizations in the kernel function themselves, or leveraging third party highly tuned linear algebra library, such as MAGMA.

D. *Threading and Memory Models*

In Panda runtime, there are four steps to map the SPMD computation to CPU cores or GPU cores. These steps include 1) from job into sub-jobs; 2) from sub-job into map tasks; 3) from map tasks into CUDA and CPU threads; 4) from CUDA threads and CPU threads into GPU/CPU cores.

Panda runtime leverages the Pthreads to manage the GPU and CPU devices. It spawns one Pthread to manage each GPU device and one Pthread to manage all of the CPU cores in the same machine. For example, if there are two GPUs and 12 CPU cores on one machine, then Panda will spawn two threads to manage the two GPUs and one thread to manage the 12 CPU cores. Panda also uses Pthreads to manage the CPU cores; and leverages the CUDA kernel threads to manage the GPU cores. In order to increase the resource utilization of the CPU/GPU cores, Panda usually spawn multiple threads for each CPU/GPU core. For each CPU core, it can spawn between one and four Pthreads. For each GPU core, Panda can spawn between two and 16 CUDA threads. GPU need enough tasks to keep most cores busy with computation. In addition, without enough threads to switch between, the GPU won't be able to hide its high latencies. CPU has much less hardware threading than GPU, and it depends on other technologies such as cache hierarchy, speculative prefetching to keep CPU cores busy; too many small tasks will increase the overhead of context switch, which is expensive in CPU. Therefore, the resource utilization of both CPU and GPU cores are increased because Panda can assign proper number of Pthreads and CUDA threads to the OS kernel and CUDA runtime. In addition, in order to accommodate processing large numbers of map tasks, each CUDA thread or CPU Pthread is usually required to process multiple map tasks. This requirement is met by striding the total number of threads within the for loop iteration. Therefore the workload balance requirement is likely to be satisfied as well.

Equation 1 illustrates the threading, internal processes, and external processes running pattern for Panda GPU/CPU tasks. EP is always equal to 1, as we only discuss single

machine in this paper. For CPU, paper [33] reports that using more number of internal processes than CPU threads can deliver good performance. For GPU, GT is equal to 1 when running the gpu_host_mapreduce function, which means it uses one C++ function to invoke CUDA or MAGMA code. This approach is leveraged by Utah work[13]. However, when invoking the gpu_device_mapreduce function, the proper number of GT is equal to or several times bigger than number of GPU cores. This approach is used by Mars. Panda support both approaches, and we will prove that the former approach can give better performance for GPU applications.

$$(CTxIP + GTxG) \times EP \qquad \text{(Eq.1)}$$

G #GPU Cards
GT #GPU Threads
CT #CPU Threads
IP #Internal Processes
EP #External Processes

CPUs and GPUs have different levels of memory hierarchy; therefore, copying data between them is not trivial work and needs elaborate effort in order to achieve a good performance. Panda can achieve the data movement between the GPU and CPU memory spaces without the effort of developers. However, we assume that all of the input data are already in the CPU memory. In the Map stage, the input data for the CPU Map tasks are copied from the user memory space to the runtime memory space in CPU. The input data of GPU Map tasks are copied from the user memory in the CPU to the runtime memory space in GPU. After the Map stage, all of the intermediate key/value pairs will be copied to the runtime memory space in CPU in order to create shuffling. The Reduce stage has a similar memory management process.

E. Optimization

1) Auto Tuning

Workload balance is critical to performance when running SPMD tasks on heterogeneous resources. Difficulty occurs when determining the proper workload distribution among the heterogeneous devices. Panda provides an auto tuning utility that can be used to find the near optimal workload distribution for heterogeneous devices. Panda uses a straightforward, mature auto tuning technology that has already been adopted by other frameworks such as ATLAS [19]. Similarly, Panda picks up a set of parameters that affect job performance and genera a serial of small test jobs by sweeping the selected parameters. Then, it runs the generated test jobs and picks up the parameters that have the best performance. Some approaches attempt to solve the workload balance issue by making heuristic models to guide work load distribution based on CPU speed, cache size, and memory bandwidth. These approaches avoid the overhead of running many tests jobs, but usually only works well for certain classes of applications or hardware. Our approach requires extra overhead, but is more likely to be adaptive to various types of devices. In addition, it is only worthwhile to use the auto tuning facility if the application is compute-intensive. Thus, the overhead of seeking the best runtime parameters is not problematic when compared with the long job running time.

2) Region-based Memory Management

Region-based memory management [20] 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 the Panda framework. First, although the latest CUDA runtime supports dynamically allocating the buffer in the GPU global memory using the malloc operation, the aggregated overhead of the malloc operation can kill the performance if many small memory allocation requests exist. For example, the word count MapReduce job can generate a large number of intermediate key/value pairs in the Map stage. Instead of dynamically allocating memory for each generated key/value, Panda allocates a block of memory for each CUDA thread, whose size should be big enough to serve many small memory allocations. When the block is filled, the runtime will double the size of the block and copy the data to a new block. The old block will be deallocated. The second advantage is that the collection of allocated objects in the region can be deallocated all at once. For example, Panda can simply deallocate a block of GPU memory assigned to each CUDA thread after copying the intermediate key/value pairs to the CPU, in which case, there would be no need to traverse all of the key/value pairs.

3) Iterative Support

A set of iterative applications, such as Kmeans, exist that have loop invariant data during the iterations [21][29]. It is costive for the GPU program to copy these loop invariant data between the CPU and GPU memories over the iterations. In order to eliminate the data staging overhead, Panda enables the program to cache loop invariant data in the GPU memory over iterations. The performance results in the next section indicate that the caching loop invariant data causes an increase in performance. Currently, Panda support the iterative computation on only one GPU because of the difficulty to maintain multiple GPU contexts between iterations. We will support iterative computation on multiple GPUs in the next release.

4) Local Combiner

If the Reduce function is associative (commutatively is not necessary), then one can apply the partial aggregation operation to a subset of the Map output by using a local combiner function. In Panda, if the above requirement for the Reduce function is satisfied and local combiner is supplied, then Panda will perform a partial aggregation to the intermediate key/value pairs generated by each CUDA thread and CPU Pthread (each thread usually processes multiple Map tasks). The gpu_combiner() function are performed within GPU memory so that the file staging overhead between CPU and GPU is minimized.

IV. APPLICATIONS AND EVALUATION

This section evaluates the Panda runtime using three sample applications on different experimental environments. Table 2 illustrates configuration of GPU and CPU devices that used in this paper. All the NVIDIA GPU cards listed in Table 2 support computation capability 2.x. The numbers of cores on Keeneland [22] and Delta [23] machine are 12 and 24 with hyper thread enabled. The user implemented API are written in CUDA and C/C++, and compiled by nvcc 4.2 and gcc 4.4.6, respectively.

Table 2: Hardware Description

Machine Name	Keeneland	Delta	Basalt
GPU Type	M2050	C2070	T430
GPUs/Node	3	2	1
GPU Memory	6 GB	6 GB	1 GB
Cores/GPU	512	448	96
CPU Type	Intel Xeon 5660	Intel Xeon 5660	Intel I5-2400
CPU Speed	2.80 GHz	2.80 GHz	3.10 GHz
Cores/CPU	12 Cores	24 Cores	4 Cores
CPU Memory	24 GB	16 GB	4 GB
Operating System	Red Hat Enterprise	Red Hat Enterprise	Red Hat Enterprise
CUDA	4.2	4.2	4.2
GCC	4.4.6	4.4.6	4.4.6

A. *C-means Clustering using Panda*

The computational demands of the multivariate clustering grow rapidly; therefore clustering for large data sets is very time consuming on a single CPU. Fuzzy C-means is an algorithm of clustering that allows one element to belong to two or more clusters with different probabilities. The C-means application [24][25] is frequently used in multivariate clustering, such as flowcytometry clustering [24]. The algorithm is based on a minimization of the following objective function:

$$J_m = \sum_{i=1}^{N}\sum_{j=1}^{C} u_{ij}^{m}\left\|x_i - c_j\right\|^2 \qquad \text{(Eq. 1)}$$

M is a real number greater than 1, while N is the number of elements. Uij is the value of the membership of Xi in cluster Cj. ||Xi-Cj|| is the norm expressing the similarity between the measured and the center. The Xi is the ith of the d-dimensional measured data; Cj is the d-dimension center of the cluster. The fuzzy partitioning is performed using an iterative optimization of the objective function as shown above. Within each iteration, the algorithm updates the membership Uij and the cluster centers the Cj using the following functions:

$$u_{ij} = \frac{1}{\sum_{k=1}^{C}\left(\frac{\left\|x_i - c_j\right\|}{\left\|x_i - c_k\right\|}\right)^{\frac{2}{m-1}}} \qquad \text{(Eq. 2)}$$

$$c_j = \frac{\sum_{i=1}^{N} u_{ij}^{m} \cdot x_i}{\sum_{i=1}^{N} u_{ij}^{m}} \qquad \text{(Eq. 3)}$$

C-means MapReduce Algorithm:
Configure:
1) Copy data from the CPU to GPU memory
Map function:
2) Calculate the distance matrix
3) Calculate the membership matrix
4) Update the centers kernel
Reduce function:
5) Aggregate the partial cluster centers and compute final cluster centers.
6) Compute the difference between the current cluster centers and previous iteration.
Main program:
7) The iteration will stop when the difference is smaller than predefined threshold or it will go to next iteration.
8) Compute the cluster distance and memberships using final centers.

Figure 5: C-means MapReduce Algorithm

The iteration will stop when $\max_{ij}\left\{\left|u_{ij}^{(k+1)} - u_{ij}^{(k)}\right|\right\} < \varepsilon$, where 'e' is a termination criterion between 0 and 1, and 'k' is the iteration steps. Figure 5 shows the alogirhtm of C-means MapReduce application.

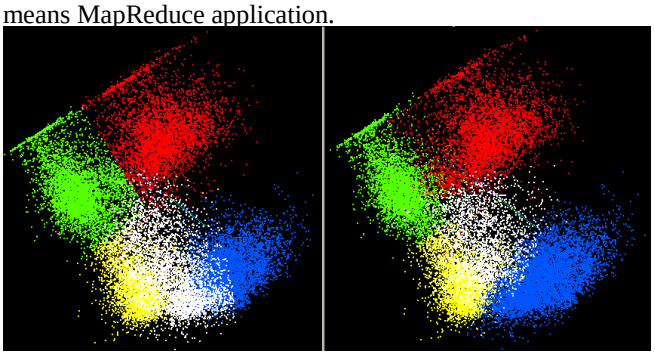

Figure 6: a. Cmeans (left) and Kmeans (right) clustering results for Lymphocytes data set after project 4D into 3D using MDS SMACOF. Lymphocytes data set [35] (22014 points, 4 dimensions, 5 clusters)

Table 3: Average Width Over Clusters and Points using Different Clustering Approaches.

Clustering Approaches	Average Width Over Clusters and Points

Kmeans	2.1479
Cmeans (hard classes)	2.1789
Cmeans (soft classes)	1.175019
Flame (finit mixture model)	2.1754
Determinstic Annealing	2.1478

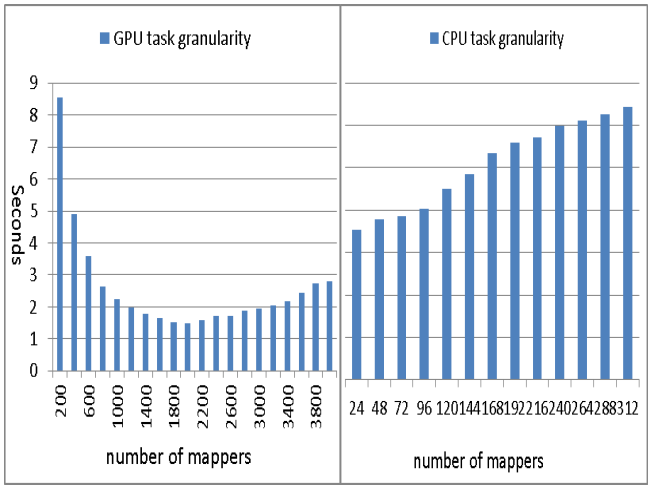

Figure 7: Performance of Different Task Granularity of Panda C-means Jobs on GPU and CPU.

We implemented C-means MapReduce application using Panda on GPU and CPU. The input matrices listed in Figure 5 were copied into CPU and GPU memory in advance. The 'key' object of Panda 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 calcuate and distence and membership matrices, and then multiply the distence matrix with mebership matrix to calcuate new cluster centers. The Reduce function aggregate partial cluster centers and calcuate the final cluster centers. Figure 7 shows peformance of Panda C-means jobs with different number of mappers using GPU and CPU on Delta machine. The parameters of C-means job are 1 million events, 100 dimensions, 10 clusers, 1 iteration. The number of GPU cores and CPU cores on Delta machine are 448 and 24 (with hyper-thread eanbled). The results indicate that the optimal number of mappers of C-means job using GPU or CPU are 2000 and 24 respectively. It is obvious that C-means GPU implementation prefered finer task grunaulitry as compared to CPU implmentation.

We also study the work load balance issue of Panda C-means job on GPUs and CPUs by mapping different ratio of workload to GPUs and CPUs. Figure 8 shows the time of Panda C-means job using GPU only, CPU only, and GPU+CPU with different workload distribution ratios. The cross point, point 0.1 in x-axis, of two lines plotted in Figure 9 is the optimal workload distribution among GPU and CPU. The time of C-means job is largely determined by calculating distance and membership matrix, which are computation steps 2)~4) in Figure 5. The computation of update centers can be considered as the matrix-vector multiplication of distance matrix and membership matrix. Therefor the C-means computation can be partitioned into some parallel sub-tasks that run on CPU and GPU, and the optimal workload distribution between CPU and GPU is at the point when the tasks on CPU and GPU get completed at the same time. The similar conclusions are also reported in papers study workload distribution issue of the matrix-matrix multiplication on GPU and CPU [26]

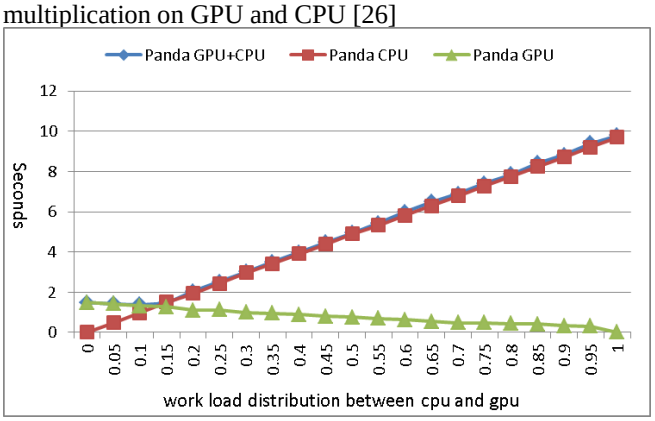

Figure 8: Effect of workload distribution on Panda C-means on GPU and CPU on Delta machine. For job only using CPU, the value X on x-axis means X percentage of workload is mapped to CPU. For job only using GPU, the value X means (1-X) percentage of work is mapped to GPU. For job using both GPU+CPU, the value X means X percentage of work was mapped to CPU, and the remain (1-X) percentage of work was mapped to GPU.

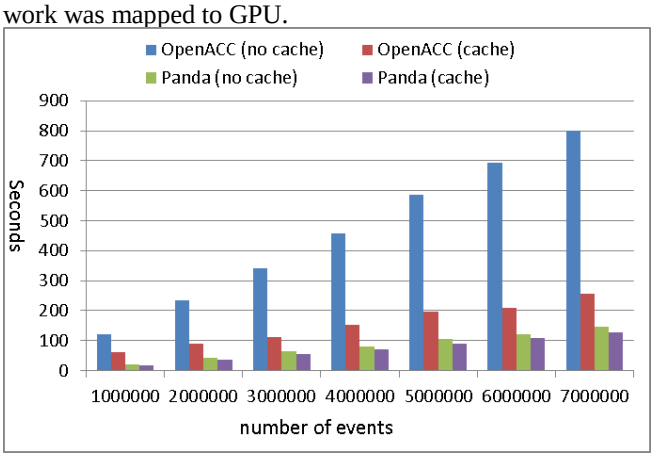

Figure 9: Performance of C-means jobs using Panda and OpenACC on 1 GPU with/without cache loop invariant data.

The C-means algorithm is of iterative computation steps, however, elements of event matrix are not changed during iterations. It is costive to copy events matrix from GPU memory to CPU memory over iterations. One can avoid this overhead by caching loop invariant data in GPU memory. In OpenACC, developers can add "#pragma acc cache (list)" directive at the top of a loop. The elements or sub-arrays in the list are cached in software-managed data cache. In Panda, developers can specify the "iterative_support" option when configuration GPU sub-jobs to indicate the runtime to copy cache loop invariant data in GPU memory once, and reuse it over iterations. Figure 9 shows the performance of

C-means jobs using Panda and OpenACC with/without using caching. The OpenACC and Panda can achieve up to the speedup of 3.14x and 1.15x when using cache for large input data.

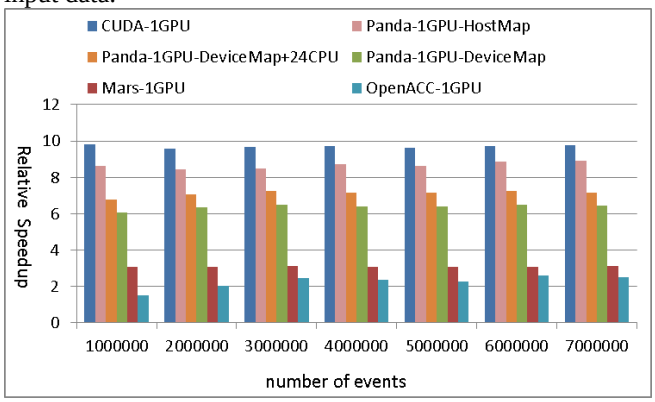

Figure 10: Relative Speedup of C-means Jobs on Delta Machine Using Panda-1GPU-DeviceMap, Panda-1GPU-DeviceMap+24CPU, Panda-1GPU-HostMap, CUDA 1GPU, Mars-1GPU, and OpenACC-1GPU, Using OpenMP 24CPU as the Baseline Performance.

Figure 10 shows the relative speedup performances of C-means jobs using different runtime environments as compared to the performance of using OpenMP on 24 CPU cores. The parameters of C-means jobs are 100 dimensions, 10 clusters, 10 iterations, and number of events range from 1 million to 7 million. The results indicate that Panda-1GPU-DeviceMap is up to 1.97x and 2.46 x faster than Mars-1GPU and OpenACC-1GPU implementation for large input dataset. The CUDA-1GPU is 1.66x, 1.51x and, 1.12x faster than Panda-1GPU-DeviceMap, Panda-1GPU-DeviceMap+24CPU, and Panda-1GPU-HostMap implementations respectively. Actually, the Panda-1GPU-HostMap implementation invoked the CUDA Cmeans code directly, and the performance gap between them is mainly due to the Panda runtime overhead. For multiple GPU results, the Panda-2GPU-DeviceMap and Panda-1GPU-DeviceMap+24CPU improve the performance by 1.88x and 1.115x as compared with Panda-1GPU-DeviceMap. CUDA-2GPU is 1.8x faster than Panda-2GPU-2DeviceMap because it leveraged benefit of coalescing memory access and no need to care about scheduling and synchronization overhead on GPUs and CPUs. However, developing Panda C-means MapReduce program requires less programming effort. The number of code lines of CUDA C-means source code file is more than 850, while the user-implemented code lines of Panda C-means is 270. Table 3 shows the size of the source code of all the three applications using CUDA and Panda.

Table 4: number of code lines using CUDA and Panda

Apps	Other	Panda
Cmeans	CUDA 850+	gpu_device_map 230+ gpu_device_reduce 40 gpu_host_map 800+ gpu_host_reduce 60 cpu_map 190+ cpu_reduce 40
Dgemm	CUDA 310+ MAGMA 30+	gpu_device_map 110+ gpu_device_reduce 0 gpu_host_map 20+ gpu_host_reduce 0 cpu_map 70+ cpu_reduce 0
Word count	Mars 110+ Phoenix 80+	gpu_device_map 25 gpu_device_reduce 5 gpu_device_combine 5 cpu_map 25 cpu_reduce 5 cpu_combin 5

B. *Matrix Multiplication using Panda*

The matrix-matrix multiplication is a fundamental kernel [27][28] widely applicable in scientific computing and data mining. The computation can be partitioned into parallel subtasks with arbitrary granularity, which makes it another good sample application by which to evaluate Panda framework on GPUs and CPUs. The matrix-matrix multiplication is defined as A * B = C (Eq. 4).

$$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj} \qquad \text{(Eq. 4)}$$

We implemented dense matrix-matrix multiplication application using Panda on CPU and GPU. The computation only consists of map stage, no shuffle or reduce stage is included. Both implementations for CPU and GPU utilize the blocking algorithm in order to enhance cache and shared memory utilization. In order to achieve better overall flops performance on GPU, we optimized gpu_map() function by coalescing the memory access of reading matrix blocks. If one block is too big to fit in GPU shared memory, we split that big block into some sub-blocks and process these sub-blocks sequentially. The computation of each sub-block is performed in parallel by CUDA threads within same block. In order to achieve better flops performance on CPU, the cpu_map() function is compiled with g++ with O3 enabled.

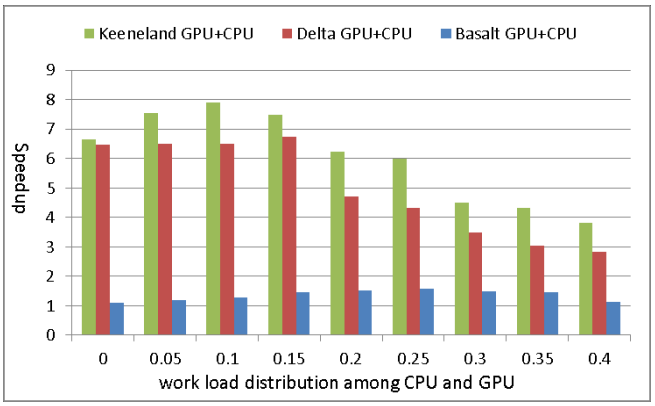

Figure 11: Speedup Performance of Matrix Multiplicaiton Jobs with Different Workload Distribution among CPU and GPU on Keeneland, Delta, and Basalt machines. Value X on x-axis presents X% of workload of job is assigned to CPU

for the processing, and the remain 1-X% of workload of job is assigned to GPU.

Figure 11 shows the speedup performance of 5000x5000 matrix multiplication job with different workload distriubion among CPU and GPU. The optimal workload distribution among CPU and GPU are 25%, 15%, and 10% when running the same job on Basalt, Delta, and Keenleand machines. Similar to the workload distribution analysis in Figure 8, the optimal workload distribution among GPU and CPU should be in proportion to the computation capability of CPU and GPU as shown in Table 1. As shown in Figure 11, machine with faster GPU card prefer to assign more workload to GPU.

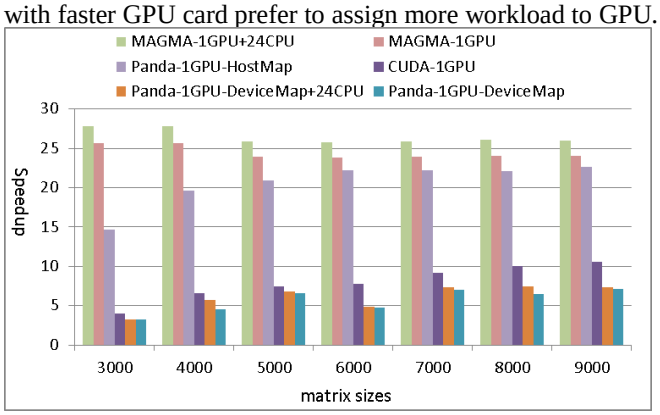

Figure 12: Speedup Performance of Matrix Multiplication Jobs using Panda-1GPU-HostMap, Panda-1GPU-DeviceMap, Panda-1GPU-DeviceMap+24CPU, MAGAMA-1GPU, MAGMA-1GPU+24CPU, and CUDA-1GPU implementations on Delta machine.

Figure 12 shows the speedup performance of matrix multiplication jobs using Panda-1GPU-DeviceMap, Panda-1GPU-HostMap, Panda-24CPU, Panda-1GPU-DeviceMap+24CPU, MAGMA-1GPU, MAGMA-1GPU+24CPU, CUDA-1GPU, Mars-1GPU, and Phoenix-24CPU. The CUDA-1GPU implementation is around 1.52~1.94x faster than Panda-1GPU-DeviceMap for large matrices sizes. The Mars and Phoenix crashed when the matrices sizes larger than 5000 and 3000 respectively. For 3000x3000 matrix multiplication job, Panda-1GPU-DeviceMap achieves the speedup of 15.86x, and 7.68x over Phoenix and Mars respectively. Panda-1GPU-HostMap is only a little slower than CUDA-1GPU for large matrices. Panda 1GPU-DeviceMap+24CPU improve the performance by 5.1% over Panda-1GPU on average. The workload distribution among GPU and CPU is 90/10 as calculated by auto tuning utility. MAGMA-1GPU+24CPU increase the performance by 7.2% over MAGMA-1GPU, where the workload distribution among GPU and CPU is determined by its auto tuning utility.

C. Word Count using Panda

Word Count computes statistics about word occurrences in text documents. Its input data is a collection of text, which can be partitioned into arbitrary granularity tasks. Therefore, word count is another typical SPMD application to be used to evaluate performance of Panda framework on GPUs and CPUs. In this section, we study the task granularity and workload balance issue of the Panda word Count on GPUs and CPUs. In addition, we compare the results with those implemented using Mars and Phoenix.

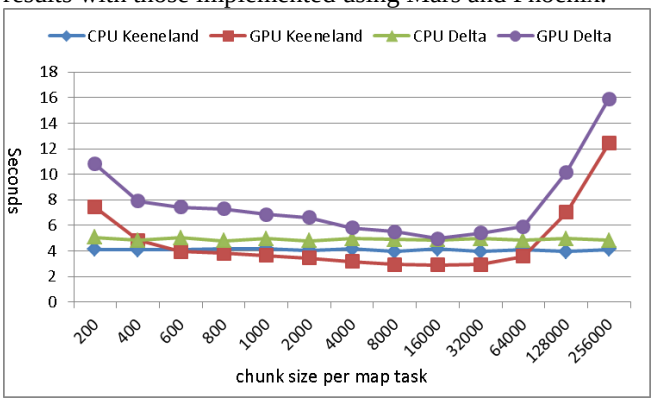

Figure 13: Performance of Panda Word Count with Different Chunk Sizes on Keeneland and Delta Machines.

For our test cases, we used randomly generated text of 100 words whose length was between 5 and 10. The size of the input text files ranged from 10MB to 200MB. In one Panda Word Count job, the input text file was loaded into memory and then it was split into several in-memory sub-files, each of which is assigned to one GPU or multi-core CPU for processing. And the assigned sub-file will be further split into some chunks, each of which presents input record of one Map task. The chunk size is specified by developers in the job configuration stage. Figure 13 shows the job time of Panda word count with different chunk sizes using only the CPU or GPU on the Keeneland and Delta machines. The results indicate that the optimal chunk size of Panda word count on GPUs is 16KB. Small chunk sizes can generate too many small map tasks which can increase scheduling overhead, while large chunk sizes may not generate enough map tasks to fully utilize all the GPU cores. In addition, few larger tasks are more likely to lead to workload imbalance issue. We also noticed that the performance of Panda word count using CPUs is not sensitive to the changes of chuck sizes because CPU has the sophisticated memory caching mechanism.

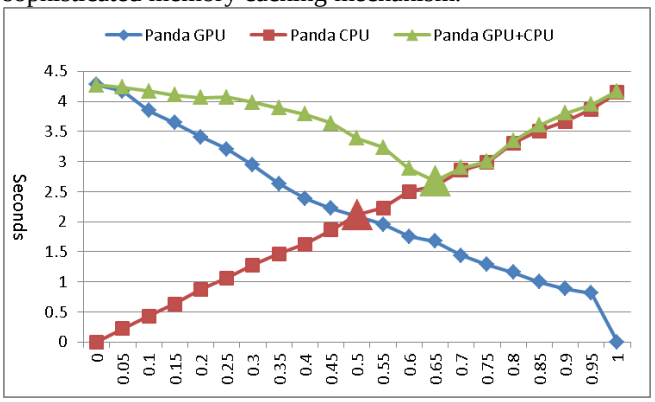

Figure 14: Effect of workload distribution on Panda wordcount jobs using 150MB text file on Keeneland machine. For Panda job only using CPU, the value X on x-axis means X percentage of work load is mapped to CPU. For Panda job only using GPU, the value X means (1-X) percentage of work mapped to GPU. For Panda job using both GPU and CPU, the value X means X percentage of work was mapped to CPU, and the remain (1-X) percentage of work is mapped to GPU.

We also studied the workload balance issue of the Panda Word Count on the GPUs and CPUs by mapping the different ratios of workload to the GPUs and CPUs. Figure 14 shows the job running time of word count job using Panda-1GPU, Panda-24CPU, and Panda-1GPU+24CPU with different workload distribution ratios. In Figure 14, the cross point, 0.5 in the x-axis, of the two lines plotted by Panda-1GPU and Panda-24CPU jobs is the theory optimal workload distribution among GPU and CPU. However, the practical optimal workload distribution ratio is at point 0.65 in x-axis. This means the workload distribution rule worked for the cases showed in Figure 8 and 10 could not be applied in the case showed in Figure 14 because Panda word count shuffles many small intermediate key/value pairs after Map stage, which incurs significant synchronization overhead among GPU and CPU. Therefore the Panda word count job prefers to assign more workload to run on the CPU device in order to decrease the synchronization overhead.

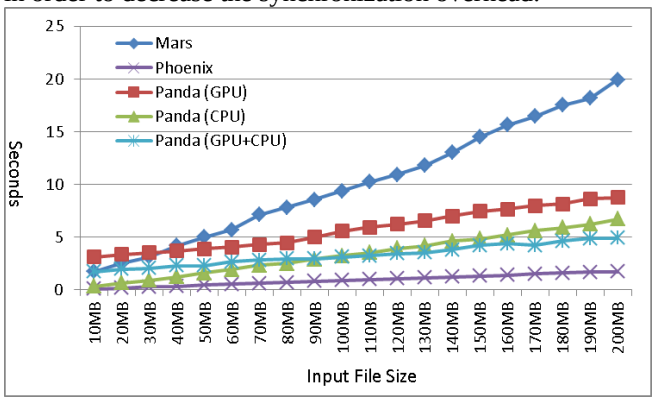

Figure 15: Performance of Word Count Job Using Panda-1GPU, Panda-24CPU, Panda-1GPU+24CPU, Mars-1GPU, Phoenix-24CPU on Delta Machine.

Figure 15 shows the performance of word count jobs using Panda-1GPU, Panda-24CPU, Panda-1GPU+24CPU, Mars-1GPU, Phoenix-24CPU on the Delta machine. The results indicate that Panda has 2.28x speedup over Mars, as Mars needs an extra run to calculate the buffer indices, which double the job running time. Another reason is that Panda supports the local combining on GPU and CPU device, which contributes around 40% performance improvement for the test input data set. We also noticed that Phoenix outperform Panda with a speedup of 4.63x due to the better CPU memory management. Another reason is that our cpu_map() function are compiled using nvcc, and while Phoenix using g++. A benchmark test of sequential word count program shown that the g++ generated binary code is around 2 times faster nvcc generated binary code. In the C-means, and matrix multiplicaion applications, the cpu_map() functions were compiled with g++ with O3 enabled. This result proved our argument that map functions should be implemented and optimzed for different devices sperattely.

I. Summary and Conclusion

The heterogeneous accelerator devices are becoming ubiquitous. However, programmability and performance challenges exist when developers want to make good utilization of these heterogeneous devices. In order to meet the challenges, we are proposing heterogeneous MapReduce model and a two-level scheduling strategy. A general purpose runtime with MapReduce interface for running SPMD computation on GPUs and CPUs is given. Experimental results of C-means clustering, matrix multiplication, and word count, indicate that using all CPU cores increase the GPU performance by 11.5%, 5.1%, and 41.9% respectively. For some application scenarios, using Panda to run SPMD job can increase device utilization, and decrease job running time.

We also found that for some application scenarios, Panda gpu_device_mapreduce functions may not performance as well as other runtime technologies. For example, Phoenix give better performance for word count, CUDA, and MAGMA are faster for Matrix Multiplication. The point is that if there are many threading code exist in SPMD application, it should be processed by tools such as Pthreads and OpenMP on CPUs, if there are many vector code exist in SPMD application, it should be processed by tools such as cuBLAS and MAGMA. Simply using threading code to process matrix algebra applications , such as Matrix Multiplication and C-means will not give good performance as it does not leveraged the vector processing instruction set. Therefore, we found that using Panda gpu_host_mapreduce functions can give 90% performance as compared with CUDA Cmeans and MAGMA applications. However, utilizing these vector processing programming model such as CUDA and Intel vector processing language will also increase the programmability for developers. Panda leave the flexibility to developers to choose whether implement the vector optimization code in the Map and Reduce function themselves. We made the design goal on trade-off between programmability and performance, which is in between OpenACC and CUDA.

The lessons we learned in Panda work include that the source code optimized for one device architecture properly won't perform well in others. Sometimes, it is necessary for developers to implement different codes or algorithm for different device architectures in order to get better performance. Second, adaptively scheduling tasks with heterogeneous granularities on GPU and CPU is another issue affects overall job performance. For applications that

have regular pattern of memory access, computation, and synchronization; the heuristic workload distribution model studied in paper[26] is the feasible solution. However, our proposed auto tuning approach showed the applicability to wider class of applications and devices at the cost of running some small tests jobs.

The future work of Panda could be:

1. Extend the framework to multiple nodes.
2. Extend the framework to other backend or accelerators, such as OpenCL, MIC.
3. Run MPMD computation on heterogeneous devices.

Acknowledgements

The authors thank Andrew Pangborn for the original C-means CUDA code and anonymous reviewers for theirs insightful suggestions. This work was partially supported by the CReSIS project funded by NASA.

REFERENCES

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

[2] NVIDA, CUDA SDK, http://www.nvidia.com/ojbect/cuda.get.html

[3] MUNSHI, A. OpenCL Parallel Computing on the GPU and CPU, In ACM SIGGRAPH (2008)

[4] Dean, J. and S. Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters." Sixth Symposium on Operating Systems Design and Implementation: 137-150.

[5] An Overview of the GridWay Metascheduler, 2009

[6] Bingshen He, Wenbin Fang, Qiong Luo, Mars: A MapReduce Framework on Graphics Processors

[7] Justin Talbot, Richar Yoo, Phoenix++: Modular MapReduce for Shared-Memory Systems.

[8] OpenACC www.openacc-standard.org

[9] Chi-Keung Luk, Sunpyo Hong, Qilin: Exploting Parallelism on Heterogenous Mulitprocessors with Adaptive Mapping

[10] Manuel Chakravarty, Gabriele Keller, Sean Lee, Accelerating Haskell Array Codes with Multicore GPUs. DAMP 2011

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

[12] 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.

[13] Alan Humphrey, Qingyu Meng, Martin Berzins, Todd Harman, Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System,2012

[14] Divide and Conquer on Hybrid GPU-Accelerated Multicore Systems, Christof Vömel, Stanimire Tomov, and Jack Dongarra, SIAM J. Sci. Comput. Volume 34, pp. C70-C82, 2012.

[15] Multithreading in the PLASMA Library, Jakub Kurzak, Piotr Luszczek, Asim YarKhan, Mathieu Faverge, Julien Langou, Henricus Bouwmeester, and Jack Dongarra in Mult and Many‐Core Processing: Architecture, Programming, Algorithms, & Applications, Edited by Mohamed Ahmed, Reda A. Ammar, Sanguthevar Rajasekaran Series: Chapman & Hall/CRC Computer & Information Science Series, published by Taylor & Francis, 2013.

[16] Kamesh Madduri, Khaled Ibrahim, Gyrokinetic Toroidal Simulations on Leading Multi-and Manycore HPC System. SC11

[17] Eric HOLK, William BYRD "Declarative Parallel Programming for GPUs" , PADL, 2011.

[18] Li, H., Y. Huashan, et al. (2008). A lightweight execution framework for massive independent tasks. Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Austin, Texas.

[19] Automatically Tuned Linear Algebra Software (ATLAS) http://math-atlas.sourceforge.net/

[20] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, SOFTWARE 2006.

[21] 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 June 20-25, 2010. Chicago, Illinois, ACM.

[22] Keeneland http://keeneland.gatech.edu/overview

[23] FutureGrid https://portal.futuregrid.org/

[24] Andrew Pangborn, Gregor von Laszewski Scalable Data Clustering using GPUs Paper Draft. 2009.

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

[26] Satoshi Ohshima, Kenji Kise, Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment

[27] G. Fox, A. Hey, and Otto, S (1987). Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing, 4:17-31

[28] Hui Li, Geoffrey Fox, Judy Qiu, Performance Model for Parallel Matrix Multiplication with Dryad: Dataflow Graph Runtime, BigDataMR, 2012

[29] Zhanquan, S., and G. Fox, Study on Parallel SVM Based on MapReduce, Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications

[30] 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.

[31] Jong Youl Choi, Judy Qiu, Marlon Pierce,Geoffrey Fox, Generative topographic mapping by deterministic annealing, International Conference on Computational Science, ICCS 2010

[32] FLAME DataSet: http://www.broadinstitute.org/cancer/software/genepattern/modules/FLAME/published_data.

[33] Judy Qiu and Seung-Hee Bae, "Performance of Windows Multicore Systems on Threading and. MPI," Concurrency and Computation: Practice and Experience

[34] Yang Ruan, Saliya Ekanayake, Mina Rho, Haixu Tang, Seung-Hee Bae, Judy Qiu, Geoffrey Fox DACIDR: Deterministic Annealed Clustering with Interpolative Dimension Reduction using a Large Collection of 16S rRNA Sequences ACM Conference on Bioinformatics, Computational Biology and Biomedicine (ACM BCB) Orlando Florida October 7-10 2012

[35] http://www.broadinstitute.org/cancer/software/genepattern/modules/FLAME/published_data.