2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes

Vignesh T. Ravi [#1], Michela Becchi [*2], Wei Jiang [#3], Gagan Agrawal [#4], Srimat Chakradhar [+5]

[#]*Department of Computer Science and Engineering, The Ohio State University*
Columbus OH - 43210
[1]raviv@cse.ohio-state.edu
[3]jiangwei@cse.ohio-state.edu
[4]agrawal@cse.ohio-state.edu

[*]*Department of Electrical and Computer Engineering, University of Missouri*
Columbia, MO - 65211
[2]becchim@missouri.edu

[+]*NEC Laboratories America*
Princeton, New Jersey
[5]chak@nec-labs.com

***Abstract*—Heterogeneous architectures comprising a multi-core CPU and many-core GPU(s) are increasingly being used within cluster and cloud environments. In this paper, we study the problem of optimizing the overall throughput of a set of applications deployed on a cluster of such heterogeneous nodes. We consider two different scheduling formulations. In the first formulation, we consider jobs that can be executed on either the GPU or the CPU of a single node. In the second formulation, we consider jobs that can be executed on the CPU, GPU, or both, of any number of nodes in the system. We have developed scheduling schemes addressing both of the problems. In our evaluation, we first show that the schemes proposed for first formulation outperform a blind round-robin scheduler and approximate the performances of an ideal scheduler that involves an impractical exhaustive exploration of all possible schedules. Next, we show that the scheme proposed for the second formulation outperforms the best of existing schemes for heterogeneous clusters, *TORQUE* and *MCT*, by up to 42%.**

I. Introduction

A. Motivation

The recent trends in computer architecture, specifically, the emergence of multi-core CPUs and many-core GPUs, and their integration within a single machine, have given rise to a new class of heterogeneous architectures. Such heterogeneity can be seen within popular desktops, as well as in each node of several of the world's fastest supercomputers. As an example of the former, AMD's recently released Fusion Accelerated Processing Unit integrates CPU and accelerator on a single die, and is targeting the desktop and notebook market. As examples of the latter, in the top 500 list released in November 2011, three of the top five fastest supercomputers in the world are CPU/GPU clusters. Apart from being used in high-end clusters, heterogeneous architectures have been recently introduced by cloud providers like Amazon and Nimbix.

Heterogeneous systems have received much attention from the parallel software community, though mostly from the programmability and application portability view-point. NVIDIA's CUDA [1], an extension to C, has been popular for programming general purpose computations, although there is a growing interest in OpenCL [2], because of its portability. Various ongoing research efforts have considered more high-level interfaces for GPU and/or multi-core programming [3], [4]. There have been a limited number of efforts in utilizing a combination of CPUs and GPUs to further accelerate application performance [5], [6], [7], [8], [9].

B. Problem Definition and Approach

Despite much interest in heterogeneous systems, key scheduling challenges associated with them have not received much attention. Particularly, with highly shared clusters (such as those at supercomputer centers) and cloud resources having heterogeneous CPU-GPU nodes, new application scheduling problems are arising. In such shared clusters or cloud environments, high utilization of resources and overall system throughput are important considerations, as opposed to the need for scaling a single application.

In this paper, we consider a cluster where each node has a multi-core CPU and a GPU. Our goal is to accelerate a set of applications using the aggregate set of resources in the cluster. We focus on two distinct scheduling problems.

Single Node Job Scheduling: In the first problem, we assume that an application can either execute on a single GPU *or* a multi-core CPU, on any one node in the cluster. This is consistent with some of the most recent trends in application development for multi-core and many-core architectures. For instance, OpenCL [2] is becoming a popular programming model for heterogeneous architectures, and is device agnostic. Thus, a kernel can be written once and compiled for many devices to produce different binaries. Furthermore, recent research has proposed compilation techniques to dynamically transform CPU into GPU code and vice-versa. For example, MCUDA [10] performs basic transformations of CUDA code into C, Ocelot [4] and PGI CUDA-x86 [11] transform CUDA into optimized x86 compatible code, SWAN [12] performs a CUDA-to-OpenCL transformation, whereas OpenMPC [13] allows OpenMP-to-CUDA translation. In comparison, however, the available support for developing applications for clusters of GPUs is much limited, and the existing GPU benchmarks like Rodinia [14] and Parboil [15] comprise only single-GPU applications.

978-0-7695-4691-9/12 $26.00 © 2012 IEEE
DOI 10.1109/CCGrid.2012.78

Our first scheduling formulation, therefore, views a cluster as having two sets of independent resources (CPU and GPU), and each job can be mapped to any one resource. Such mapping needs to be performed with the goal of maximizing overall throughput, while keeping the latency of each application at a reasonable level.

Multi-Node Job Scheduling: The second scheduling problem we consider assumes that applications can be developed to flexibly utilize available resources in a heterogeneous cluster. Thus, an application can either use the GPU, the multi-core CPU, or both, on any number of nodes in the cluster. Thus, the same job can be executed on a cluster of GPUs, or a cluster of CPUs, or both. While there is very limited amount of work on developing applications with these properties today, it is clearly desirable to have such flexibility in the future.

The scheduling problem now involves determining the number of nodes for execution of each application, and whether the application should just use the GPU on each node, the multi-core CPU on each node, or both. Furthermore, we assume that a job using a set of GPUs and a job using CPUs on the same set of nodes can share interprocess communication resources.

The two scheduling problems we have defined above are challenging for a number of reasons. The first factor is that CPUs and GPUs have different processing capabilities, and as a result, each workload performs differently on multi-core CPUs and GPUs [16]. Moreover, the relative performance of a workload on CPU and GPU can vary with the problem size. Similarly, different applications may scale differently with increasing number of nodes. Thus, there are very interesting trade-offs with respect to scheduling an application on the resource where it will perform better, as compared to scheduling it on the resource that becomes available sooner. Additionally, while overall system throughput is important, one cannot ignore the latency of each application from the point of submission to its completion. Earlier research in scheduling has addressed concurrent parallel job scheduling on supercomputers and clusters [17], [18], [19], [20], [21], and on chip multi-processors with different computing cores [22], [23]. However, these works were proposed either for homogeneous distributed systems or did not involve heterogeneity introduced by the presence of CPUs and GPUs within a machine.

In this work we propose a number of novel scheduling schemes to enable job scheduling on CPU-GPU clusters. For the single node jobs scheduling (first scheduling problem), our proposed heuristics include Relative Speedup based policy with Aggressive assignment (`RSA`), Relative Speedup based policy with Conservative assignment (`RSC`), and Adaptive Shortest Job First policy (`ASJF`). These policies vary with respect to whether they focus on using the resource where an application's performance will be better, or on using the resource that becomes available sooner. These policies also vary with respect to the amount of information about the application that a user needs to provide to the scheduler. For the multi-node jobs, i.e., the *second scheduling problem*, we have developed a Flexible Moldable Scheduling scheme (FMS) that considers molding of jobs along two dimensions: resource type and the number of requested nodes.

We evaluate our scheduling schemes for a number of different combinations of workloads, including workloads with a larger number of CPU or GPU-oriented jobs. The main observations from our experiments are as follows. First, our proposed schemes for the first scheduling problem perform significantly better than a naive scheme, which we refer to as the *Blind Round Robin* scheme. Furthermore, the resulting system throughput is within 10 - 20% compared to a *static optimal* performance obtained from an impractical exhaustive search-based scheduling algorithm. Furthermore, our proposed algorithm for the second scheduling problem improves the overall throughput by up to 42% when compared to the best of two existing schemes in the literature, *TORQUE* and *MCT*.

II. System Model and Challenges

In this section, we describe our *system model*, i.e., the nature of the environment and applications we consider. Then, we make several observations about possible approaches and challenges.

A. System Model

In this paper, we target a cluster of nodes, where each node contains a multi-core CPU and a GPU. Note that with current technologies, there could be multiple GPUs on a single node, or there could be up to 4 GPUs in a box (like the S2050/S2070 boxes from Nvidia) that are accessible from 2 nodes in the cluster. For simplicity, we assume 1 GPU per node, directly accessible only from the multi-core CPU located on the same node. The CPU and the GPU are connected through a PCI-Express interconnect. Each node communicates with the other nodes in the cluster using an interconnection network.

Any given application can either use all cores on the multi-core CPU, or can primarily use the GPU. It should be noted that GPU intensive applications still require one CPU thread. Typically, such thread is used to issue GPU calls and does not consume many computing cycles. Therefore, we can assume that all cores in the CPU can be effectively used by the other application scheduled on the same node to use the CPU. In practice, it is possible that some applications can be executed only on the multi-core CPU or primarily on the GPU, but for simplicity, we assume that each application can be executed on either platform. As mentioned in the Introduction, recent research has focused on compilation techniques to dynamically transform CPU into GPU code, and vice-versa.

We also assume that there is a shared file system across all the nodes in the cluster. Thus, binaries of the workloads and the corresponding data files need not be communicated across nodes. This is a reasonable assumption for most clusters today. In our scheduling problem, we further assume that no job preemption is performed, and the scheduler operates on batches of jobs. However, batches of jobs enter the system stochastically and are not known *a priori*.

Existing techniques for scheduling across multiple sites [20], [21] and widely used resource managers like SLURM [24] make scheduling decisions based on user inputs regarding execution times and speedups. Similarly to these approaches, we assume that information about relative speedups/execution times are provided by the user or are available through analytical models [16] or profiling [25].

B. Problem Formulation and Challenges

Our goal is to address two scheduling problems: one targeting *single-node* and the other targeting *multi-node* applications. First, given a CPU-GPU cluster and a set of *single-node* jobs, we want to effectively schedule each job on either one of the multi-core CPUs or one of the GPUs available in the cluster. Second, given a CPU-GPU cluster and a set of *multi-node* jobs, we want to effectively schedule each job on either a

set of CPUs, or a set of GPUs, or a set of combined CPU-GPU resources. In both cases, the goal is to maximize the overall system throughput while keeping the average execution latency at a reasonable level.

To understand the main challenges for our scheduler, we make the following observations. CPUs and GPUs differ not only in their computational capabilities, but also in the memory capacities and latencies. On one hand, a multi-core CPU has a small number of cores, where each core is quite powerful, and MIMD parallelism can be applied across the cores. On the other hand, a GPU has a large number of simple cores, each of them working in an in-order SIMD fashion. Moreover, on CPUs, the best performance can generally be achieved when the number of running threads is at most twice the number of cores. However, on GPUs, a much higher number of threads can help to hide the memory latencies. Finally, GPU programs are hosted by the CPU threads and need to be launched from the CPU. As a result, data are transferred back and forth between CPU and GPU. This cost can sometimes be significant and could reduce the benefits from using a GPU.

CPUs generally perform better than GPUs on small problem sizes and short jobs. As mentioned earlier, GPUs have many cores and typically thousands of threads need to be launched to achieve good performance. Therefore, small problem sizes lead to low GPU utilization. On the other hand, larger problem sizes can allow an effective use of the GPU resources, leading to higher speedups. Besides the problem size, the analysis of the computation to memory ratio and of the computation patterns can help identify the suitability of the workload to a resource. Workloads performing frequent GPU device memory accesses are likely to report lower speedups on the GPU. Moreover, the ratio of data copy between CPU main memory and GPU device memory to the computation is also a key factor in deciding whether a workload is suitable for CPU or GPU. In general, compute-intensive applications can benefit more from the GPU, while memory intensive applications are better suited to multi-core CPUs.

The main challenge for the scheduler is to decide to which resource(s) (multi-core CPU, GPU, or possibly both) a particular workload should be assigned. Since we want to support a high overall throughput, and since different jobs have different performances on different resources, the scheduler will privilege the resources which represent the best match for each task. In addition, the scheduler may perform scheduling decisions that maximize the aggregate throughput of the workload at the expenses of the execution time of each single application. To this end, the scheduler may perform molding, that is, it may assign to some jobs fewer resources than optimally required when such jobs are run on a dedicated cluster.

III. Scheduling Policies

We now present scheduling policies for both the problems we have formulated.

A. Scheduling of Single Node Jobs

The policies we have proposed and evaluated for this case fall into two groups, based on the user input that they require. The first group (Relative Speedup based policies) require the relative multi-core (MP) as well as the GPU (GP) speedups compared to the sequential CPU version. The second category (Adaptive Shortest Job First policy), in addition to MP and GP, also requires the sequential CPU execution time (SQ) of the jobs being submitted.

Relative Speedup Based - Aggressive or Conservative: This scheme operates on batches of N jobs, and, as mentioned, takes the relative multi-core (MP) and the GPU (GP) speedups of each job as input. It first creates two job queues (`CJQ`) and (`GJQ`), which contain the jobs that report better speedup on the multi-core CPU and on the GPU, respectively. Each of these queues is sorted in descending order with respect to the difference in performance of the jobs on the two resources. This difference in performance is used as an indicator as to where non-optimal placement would incur less penalty.

Whenever a resource becomes free, the scheduler assigns to it the job at the top of the corresponding queue. This decision is straightforward if such queue is not empty. However, the non-optimal case arises when the corresponding job queue is empty, e.g. when a CPU (GPU) becomes available for execution, but there are no pending jobs that would prefer to execute on the CPU (GPU). We propose two schemes that handle this case differently.

In the *Aggressive* scheme, a job available for scheduling is assigned to the non-preferred resource. This is done so as to avoid overall system throughput degradation due to waiting (queuing delay) for the optimal resource. In particular, if a CPU (GPU) is idle, the aggressive scheme dequeues the task from the bottom of `GJQ` (`CJQ`), and schedules it on the CPU (GPU). In the *Conservative* scheme, the jobs in the queue are not scheduled until the preferred resource becomes available. In this case, idle times are preferred over slower execution of any task due to non-optimal assignment.

Note that the Relative Speedup based schemes operate oblivious of the actual execution times of the tasks. Without this information, it is not possible to determine the amount of idle time or the amount of slowdown expected by scheduling a task on a non-preferred resource.

Adaptive Shortest Job First: This scheme, in addition to the inputs taken by the previous policies, requires the sequential CPU execution time (SQ) of each job being submitted. In particular, this additional information makes the Adaptive Shortest Job First scheme an enhanced version of the Relative Speedup based schemes. This scheme improves over the previous one in two ways.

First, the knowledge of the expected execution times allows to easily determine the penalty for non-optimal resource assignments. Thus, when faced with a non-optimal assignment, the scheduler can choose between the aggressive or conservative decision, based on the calculation of the penalty of each. Specifically, the scheduler compares the minimum penalty for scheduling a job on the non-optimal resource to the wait time for the optimal resource to become free. Second, this policy schedules shorter jobs before longer ones. This reduces the latency perceived by the user.

Overall, this policy will get the best of the aggressive and conservative policies above, and further reduce the latency caused by queuing delays.

B. Scheduling of Multi-Node Jobs

We now describe a scheme we have developed specifically for multi-node jobs. As compared to the scheduling in the previous subsection, we assume that jobs are flexible in two other ways: 1) A job can simultaneously utilize both the multi-core CPU and the GPU within a node, i.e. in addition to the

possibility of using either only the multi-core CPU or the GPU within a node, 2) A job can request multiple nodes in the cluster as specified by the user. As we had stated earlier, our work is assuming a programming model where such flexibility is possible. For example, MATE-CG [26] is a map-reduce like system which can execute applications to use either multi-core CPU (CPU-only), a GPU (GPU-only), or both (CPU+GPU), on any number of nodes of a cluster, starting from the specification of reduction functions. We expect further developments in this area in the near future, allowing a larger class of applications to be developed for such flexible execution.

Before presenting our scheduling scheme, we present the main insight for our scheduling decisions. When run in isolation, a job may achieve the best performance by simultaneously utilizing both the multi-core CPU and the GPU within each node. However, when we are trying to concurrently schedule a set of jobs, always allocating both the multi-core CPU and the GPU to one job may not lead to the best overall throughput. This is because jobs can differ in their relative performance between the CPU and the GPU. If we want to schedule two jobs, one of which uses the CPU very well and another uses the GPU very well, it may be desirable to give a set of CPUs to one job, and the set of GPUs (from the same nodes) to another job.

Similarly, a job may achieve the best performance when the allocation is equal to the number of nodes requested by the user. However, there are two factors that need to be considered before allocating as many nodes as requested by the user. First, the jobs may not have a perfect linear scalability on the requested number of nodes. Next, the job may have to wait for a much longer time in the queue for requested number of nodes. However, a smaller number nodes (say half of requested) may be available much sooner. Thus, in many cases, allocations with small number of nodes as requested may lead to effective global throughput.

To enable such scheduling decisions, we assume that the execution times of the jobs are available when utilizing both CPUs and GPUs, only the multi-core CPUs, or only the GPUs. Since our scheduling algorithm consider the possibility of flexibly molding the number of nodes, we also take the execution times of the jobs on half and quarter of the number of nodes originally requested for the job. We assume that this information can be provided by the user, or is available through profiling and/or modeling.

Flexible Moldable Scheduling Scheme (FMS): As the jobs arrive in batches, we first group and sort the jobs in the following way. The submitted jobs are grouped based on the resources requested for each job (in our case, the number of nodes). Thus, jobs with similar resource requirements are grouped. Now, within each group, the jobs will be sorted in the ascending order of their execution times corresponding to the $CPU + GPU$ version. Grouping the jobs based on the resource requirement (number of nodes) improves the minimization of resource fragmentation. For instance, in most cases, since i^{th} and $i + 1^{st}$ jobs request the same (or similar) number of nodes, it reduces the need to wait for additional resources due to disparity in the request. Sorting the jobs within a group increases the chance of bringing two jobs with similar execution times next to each other during scheduling considerations. This is a key step that help our decision making process which is discussed below.

In our scheme, we consider the scheduling options for a *pair* of jobs at every scheduling event, rather than one job at a time. This is done to obtain a good global view for scheduling and to make more informed decision for coscheduling jobs on the same set of nodes, without having a very high cost that would arise from considering all possible options.

We earlier mentioned that three types of resource allocations (CPU-only, GPU-only, and CPU+GPU) are possible. Let us assume that the execution times (for three versions) of a given job J_i on a requested number of N nodes (N is less than the number of nodes in the cluster) are $T(i, N, C)$, $T(i, N, G)$, and $T(i, N, CG)$, denoting the CPU, GPU, and CPU+GPU executions, respectively, on N nodes. Now, for a number of jobs that are already executing or submitted for execution, we maintain the *Current Completion Time (CCT)* for these jobs in the system. When scheduling two new jobs J_1 and J_2, both of which are requesting N nodes, we have the following possibilities.

First, both the jobs can be allocated CPU+GPU on N nodes sequentially. Second, they can be given N nodes each in parallel (if 2 $timesN$ nodes are currently available). Third possibility is to map one job to only CPUs and another to only GPUs, both on the same set of N nodes, at the same time. For each of these possibilities, we predict the completion times for jobs in the system, and choose the option that leads to the lowest completion time. Note that while second option will result in lowest completion time for two jobs under consideration, it will not allow other jobs in the queue to be scheduled for a longer time. However, if the system is lightly loaded, it may be the best option overall.

Another possibility we need to consider is as follows. When the requested number of nodes, N, are not available at the time of scheduling, or when the system is very highly loaded, we consider the possibility of molding the number of requested nodes. Specifically, we consider the possibility of molding the number of nodes to $N/2$ and $N/4$. These possibilities are considered in addition to the possibilities listed above. Among all these options, we choose the one that results in fastest global completion time.

It is easy to see that this scheme is not optimal, as it considers sharing only among a pair of jobs, and does not consider the impact on jobs that may arrive in the future. However, as we stated above, by comparing options across only two similar jobs, we keep the scheduling overheads low. Our scheme can also be extended to consider more than two jobs for possible resource sharing.

IV. Experimental Results

A. Experimental Setup

Our experiments are conducted on a cluster consisting of sixteen heterogeneous nodes. Specifically, each machine is equipped with two Intel Quad-core Xeon E5520 CPUs (8 cores each running at 2.27 GHz), 48 GB main memory and an Nvidia Tesla C2050 (Fermi) GPU card. Each Tesla C2050 card has 14 streaming multiprocessors (SMs), each containing 32 cores (for a total of 448 cores) running at 1.15 GHz, and 3 GB device memory. The machines are connected through an Infiniband network.

B. Benchmarks

In our evaluation, we use two sets of benchmarks: one consisting of *single-node*, and the other consisting of *multi-node* applications.

***Single-node* applications:** Our *single-node* applications workload includes the ten applications listed in Figure 1. As can be seen, this benchmark covers different application domains: Scientific (PDE Solver and PCA), computational finance (Black Scholes and Binomial Options), physics (FDTD, Monte Carlo and Molecular Dynamics), image processing (Image Processing), and machine learning (Kmeans and KNN). For each of these applications, we consider three execution configurations: small dataset, large dataset, and large number of iterations (on large dataset). In Figure 1, we show the results obtained by running each benchmark program with the large dataset configuration. Specifically, we report the sequential execution time (on CPU), as well as the GPU and CPU speedups relative to sequential execution. Due to space limitations, we only show the results obtained from large dataset, however, we discuss the trends from all the three configurations below.

The significance of using three execution configurations is the following. When large datasets are used, the majority of the benchmarks (6 out of 10) run faster on the GPU as compared to the multi-core CPU. The trend changes when small datasets are used; in this case, the majority of the benchmarks (7 out of 10) perform better on the multi-core CPU. This is expected, since with small datasets the utilization of the 448-core GPU is limited, and the initial overheads on the GPU can limit the speedup. Similarly, when the number of iterations increases, the ratio of data transfer overhead to the computation time decreases, leading to increased performance improvement on the GPU.

Having considered 10 applications and 3 configurations, we have created a pool of 30 jobs that perform differently on CPU and GPU. In the experiments described in the next section, we create workload mixes by randomly selecting jobs from the described pool.

***Multi-node* applications:** Our *multi-node* applications workload includes three benchmark applications: Gridding Kernel (GK), Expectation Maximization (EM), and PageRank (PR). These applications arise from scientific data processing, data mining, and web search, respectively. These applications were developed using a middleware MATE-CG [26], which allow map-reduce like applications to be developed to use a multi-core CPU, a GPU, and/or both, on any number of nodes in a system.

We execute these applications using two datasets, three numbers of nodes (2, 4 and 8), and three kinds of resources (only multi-core CPU, primarily GPU, and hybrid CPU-GPU). The results for all execution configurations are reported in Figure 2. Note that the CPU+GPU configuration exhibits good scalability for GK, but does not noticeably scale in case of EM and PR. Moreover, all three applications scale well with the number of nodes. Thus, by considering three applications, two datasets and three resource requirements (i.e., number of nodes), we create a pool of 18 jobs that we will use in our multi-node experiments.

C. Experiments on Single-Node Applications

In our *single-node* applications experiments, we compare our proposed scheduling policies - Relative Speedup Based with Aggressive Option (`RSA`), Relative Speedup Based with Conservative Option (`RSC`), and Adaptive Shortest Job First policy (`ASJF`) - with two baseline schemes: `Blind Round Robin (BRR)` and `Manual Optimal`. The `Blind Round Robin (BRR)` scheme is oblivious to

Benchmarks	Seq. CPU Exec. (sec)	GPU Speedup (GP)	Multicore Speedup (MP)	Data set Characteristics
PDE Solver	7.3	4.7	6.8	14336*14336
Image Processing	33.8	5.1	7.8	14336*14336
FDTD	8.4	2.2	7.6	14336*14336
BlackScholes	2.6	2.1	7.2	10 mil options
Binomial Options	11.8	5.6	4.2	1024 options
MonteCarlo	45.4	38.4	7.9	1024 options
Kmeans	330.0	12.1	7.8	1.6 * 10 ^ 9 points
KNN	67.3	7.8	6.2	67108864 points
PCA	142.0	9.7	5.6	262144*80
Molecular Dynamics	46.6	12.9	7.9	256000 nodes, 31744000 edges

Fig. 1. *Single-node* Benchmark Applications Characteristics - Large Datasets

Application	Dataset Size	No. of Nodes	CPU Time (s)	GPU Time (s)	CPU+GPU Time (s)
Gridding Kernel (GK)	680 MB	2	314	311	221
		4	179	173	123
		8	103	99	65
	2.7 GB	2	1280	1152	703
		4	638	591	370
		8	338	301	188
Expectation Maximization (EM)	450 MB	2	109	39	37
		4	55	19	18
		8	28	9	8
	900 MB	2	215	74	73
		4	106	36	33
		8	54	18	16
Page Rank (PR)	900 MB	2	6.5	1.1	0.9
		4	3.3	0.6	0.5
		8	1.8	0.3	0.38
	3.6 GB	2	26	4.1	3.4
		4	13	2.3	1.8
		8	8.2	1.3	1.2

Fig. 2. *Multi-node* Benchmark Applications Characteristics

the fact that jobs perform differently on different resources; the `Manual Optimal` scheme assumes that the performances on CPU and GPU of all jobs in the workload mix are known a priori and an exhaustive search can be applied to find the schedule leading to the best overall system throughput.

Our evaluation is based on four metrics. First, we use the total completion time for the entire workload mix (Comp. Time or `CT`) as an indicator of the system throughput. The next two metrics focus on the scheduling related delays incurred in the execution of each job. If there is only one job to be scheduled, it will be executed on the resource on which it will perform better, and its completion time will equal its execution time on this resource. In practice, competition for resources may make the completion time of each job higher. In fact, competing jobs may be scheduled on a non optimal resource, or may incur queuing delays. The second metric we consider is the average delay introduced due to the non-optimal assignment for all the jobs in the workload mix (Ave. NOA. Lat or `ANL`). The third metric is the average queuing delay (Ave. QD Lat. or `AQL`). Finally, our last metric is the maximum idle time (Max. Idle Time or `MIT`), or the amount of time the least occupied resource in the entire system is not used. This metric can be seen as an indicator of resource utilization.

Results from Different Workload Mix The first experiment

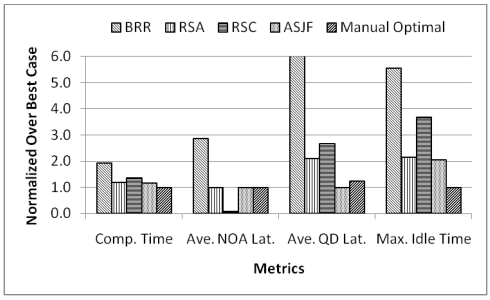

Fig. 3. System Throughput and Latency for Uniform CPU/GPU Workload

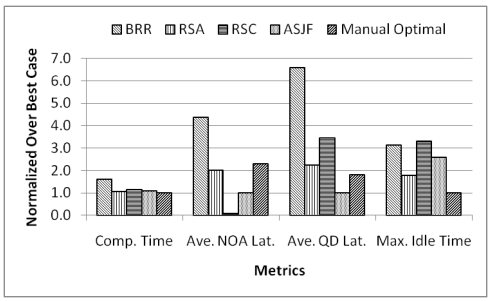

Fig. 4. System Throughput and Latency for CPU Biased Workload

is conducted on a set of two nodes from the cluster described in Section IV. Three workload mixes, each consisting of 24 randomly selected jobs, are considered. Workload mixes are generated as follows. We categorize the jobs as CPU- and GPU-friendly (depending on their performances on the different computing resources). The first workload mix is characterized by an equal number of CPU-friendly and GPU-friendly jobs; in the second and third workload mixes, 75% of the jobs are CPU- and GPU-friendly, respectively.

The results reported on all considered workload mixes are shown in Figures 3, 4, and 5. In the charts, the performance of each policy is normalized with respect to the the best reported result. When averaging the results across the three considered workload mixes, we can observe that our proposed schemes (`RSA`, `RSC` and `ASJF`) exhibit a 108% lower completion time over `BRR`. In addition, our policies show significantly better results with respect to the two latency factors `ANL` and `AQL`, as well as the utilization metric `MIT`. When compared to the ideal `Manual Optimal` policy and considering the completion time, our proposed schemes perform on the average within 20.6% and the best of the three proposed schemes performs

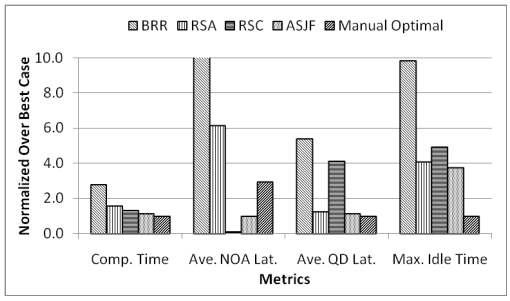

Fig. 5. System Throughput and Latency for GPU Biased Workload

within 12%.

We next analyze the trade-offs between `RSA`, `RSC`, and `ASJF`. `RSA`, as expected, incurs a higher `ANL` and a low `AQL` across all workload mixes. In the case of `RSC`, `ANL` is always 0 (by default), and correspondingly, `AQL` is higher. Thus, the relative total completion time between `RSA` and `RSC` is dependent on the dominant factor between `ANL` and `AQL`. In the case of `ASJF`, `AQL` is very low. Since the `RSC` policy waits until the optimal resource is available, it results in a high `MIT` as compared to `RSA` and `ASJF`. Overall, we can see that `ASJF` is able to almost always achieve the best results in completion time as well as application latency. However, it achieves these benefits at the cost of needing an additional input value from the user.

While our charts have reported normalized values for all metrics, it is worth comparing the absolute values of maximum idle time and the completion time. For the balanced workload, these values are 226.6 and 705.2 seconds, respectively. This shows that our schemes are quite successful in keeping the resources utilized, i.e, each resource is occupied for 68% of the time or more.

D. Experimental Results on Multi-node Applications

In our *multi-node* applications experiments, we compare our scheduling schemes with two existing policies: a practical scheme used by TORQUE [27], a popular scheduler for today's heterogeneous clusters, and a published scheme, called Minimum Completion Time [28]. We refer to these two schemes as *TORQUE* and *MCT*. *TORQUE* always assigns the resources (type of resource as well as number of nodes) as requested by the user. Thus, for *TORQUE*, we assume that the user always requests the resource that provides the best performance for the specific job. On the other hand, *MCT* considers the option of molding the resource type (selecting either multi-core CPU only, or GPU-only, or CPU+GPU), and assigns a job to the resource where it will complete the earliest depending on the waiting times on each resource. However, our algorithm for molding the resource type is different from *MCT*. In particular, we consider the opportunity to co-schedule jobs on a (or set of) node(s) so to maximize the resource utilization. In addition, we consider downgrading the number of nodes requested by the user, which is not performed by *TORQUE* and *MCT*.

The experiments were performed using all 16 nodes in the cluster described in Section IV. Specifically, we run 32 jobs randomly selected from the pool of *multi-node* jobs described earlier, and we dispatch them in batches of 8. The inter-arrival times between batches are exponentially distributed [29].

Varying Execution Length of Jobs In the first experiment, we use three job mixes created by varying the ratio of long and short running jobs. The results are shown in Figure 6. On the *x*-axis, *x SJ/y LJ* stands for *x*% short and *y*% long running jobs.

As mentioned earlier, we compare our scheduling scheme with *TORQUE* and *MCT*. Our proposed FMS scheme (*Molding ResType+NumNodes*) performs molding on both resource type and number of nodes. To better understand how different factors impact the performances, we show the results reported when molding only along one dimension: either the resource type (*MoldingResType Only*), or the number of nodes (*MoldingNumNodes only*). In particular, we report the total completion time normalized over the completion time obtained using the (*Molding ResType+NumNodes*) scheme.

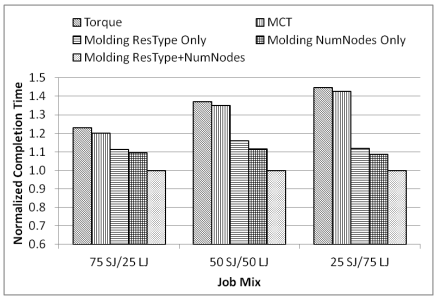

Fig. 6. System Throughput for *Multi-node* Jobs with Varying Execution Lengths

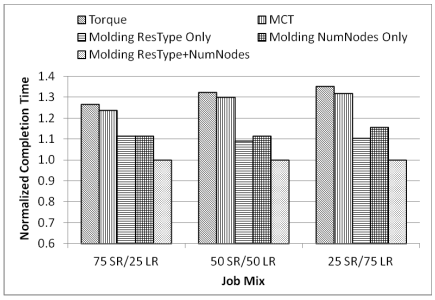

Fig. 7. System Throughput for *Multi-node* Jobs with Varying Resource Request Size

As can be observed, our FMS (Molding ResType+NumNodes) scheme achieves the least completion time and is up to 42% better than the faster of *TORQUE* and *MCT*. It is worth noting that each type of molding when applied in isolation leads to good performance improvement when compared to *TORQUE* and *MCT*. However, when both the molding techniques are applied together, we obtain an additional performance improvement of about 12%. In addition, the presence of long jobs creates an imbalance in the simultaneous availability of the nodes. Our schemes handle those imbalances much better than the considered baselines.

Varying Resource Request Size of Jobs Our second experiment studies the effect of varying the size of the resources requested by the jobs. In particular, in the results shown in Figure 7, we varied the ratio of the number of nodes requested by a job. On the *x*-axis, *x SR/y LR* stands for *x%/y%* of the jobs request a small/large number of nodes.

As can be seen, our scheme reduces the completion time by a factor up to 32% compared to the faster of *TORQUE* and *MCT*. Consistently with the previous experiment, we can observe that good performance improvements can be achieved by applying molding only in one dimension. Moreover, the combination of the two molding techniques leads to an additional 11.3% reduction in the completion time. It is interesting to note that, as we increase the ratio of large requests, the benefit from *MoldingResType Only* is higher than *MoldingNumNodes Only*. In other words, as the resource contention increases, resource sharing (by using CPU-only and GPU-only implementations) is an effective mechanism to guarantee good aggregate performances.

E. Additional Considerations

As anticipated, jobs using primarily the GPU (for brevity, GPU-only jobs) need a CPU-core to issue GPU calls. We evaluate the penalty due to sharing a CPU-core between a CPU-only and a GPU-only job deployed on the same set of nodes. We recall that CPU-only jobs are intended to use all cores on one or more multi-core CPUs. The results of experiments performed on three combinations of two *multi-node* applications are shown in Figure 8. In all cases, CPU-only and GPU-only jobs are deployed on the same set of nodes. As can be seen, the average sharing penalty is low (less than 7%) in all cases. This indicates that resource sharing is feasible and desirable in today's CPU/GPU heterogeneous clusters.

CPU-only App	GPU-only App	Average Sharing Penalty (%)
Expectation Maximization	GriddingKernel	4.2
Expectation Maximization	PageRank	6.8
PageRank	GriddingKernel	2.4

Fig. 8. Penalty from Simultaneous Execution of CPU- and GPU-only jobs on the Same Set of Nodes

V. Related Work

In this section, we compare our work against relevant work on parallel job scheduling.

Scheduling of parallel jobs in a cluster or a supercomputing center is a very widely studied problem [17]. Other related works that enable scheduling for parallel jobs in a distributed cluster are [18], and [19]. These efforts propose scheduling strategies that involve both static and dynamic mechanisms. However, the above works focus only on the homogeneous type of processors. Sabin *et al.* [20], and [21] proposed and evaluated job scheduling strategies multiple sites, i.e. separate clusters with different types of processing nodes. Our work consider a distinct type of heterogeneity, which is arising in the emerging environments.

Torque [27], an open-source resource manager, is being widely used to manage heterogeneous clusters comprising multi-core CPUs and GPUs. However, the mapping and scheduling strategy used by Torque (based on OpenPBS) is fairly simple. While waiting for a resource, it does not consider the possibility of assigning the job to another resource type. Thus, Torque cannot exploit the flexibility that emerging frameworks like OpenCL will provide. Other related efforts [30], [28] do consider the possibility of scheduling a job onto a different resource type, but are limited in the moldability they consider. Parallel job scheduling with moldability has also been studied [31], [32]. Our work is specific to moldability in heterogeneous environments. Ahmad *et al.* [33] developed a scheduling scheme for jobs with fixed priority on heterogeneous computing systems, where they dynamically compute the precedence for jobs on different resources to make scheduling decisions. However, unlike our work, they consider scheduling for dependent set of tasks represented as directed acyclic graphs (DAG).

Scheduling with consideration of resource heterogeneity has also been addressed in the context of grid computing [34], [35], [36], [37]. These efforts, however, differ both in targeting much longer running applications, and in the nature of systems they focus on.

VI. Conclusions

In this paper, we have considered the problem of optimizing the overall throughput of a set of applications deployed on a

cluster of heterogeneous nodes comprising multi-core CPUs as well as many-core GPUs. Specifically, we have addressed two scheduling problems: one targeting *single-node* (first problem) and the other targeting *multi-node* (second problem) applications.

For the first problem, we have proposed three scheduling heuristics - namely Relative Speedup based policy with Aggressive Assignment (`RSA`), Relative Speedup based policy with Conservative assignment (`RSC`), and Adaptive Shortest Job First policy (`ASJF`). Our proposed schemes are based on different amount of information expected from the user, and focus on different tradeoffs between application wait and completion times. We have shown that our scheduling policies outperform a blind round robin scheduler and approximate the performances of an ideal scheduler that involves an impractical exhaustive exploration of all possible schedules. For the second scheduling problem, we have developed a Flexible Moldable Scheduling scheme (FMS) that considers molding of jobs along two dimensions: resource type and the number of requested nodes. We also demonstrate that our FMS scheme improves the overall throughput by up to 42% when compared to the best of two existing schemes in the literature, *TORQUE* and *MCT*.

Acknowledgments

This work is supported by NSF grants CCF-0833101 and IIS-0916196

REFERENCES

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, pp. 40–53, March 2008.

[2] "OpenCL," http://www.khronos.org/opencl/.

[3] C. Hong*et al*, "MapCG: writing parallel program portable between cpu and gpu," in *PACT '10*, New York, NY, USA, 2010, pp. 217–226.

[4] G. F. Diamos *et al*, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT'10*, New York, NY, USA, 2010, pp. 353–364.

[5] G. Teodoro *et al*, "Coordinating the use of GPU and CPU for improving performance of compute intensive applications," in *CLUSTER '09*, 2009, pp. 1–10.

[6] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *HPDC '08*, New York, NY, USA, 2008, pp. 197–200.

[7] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO '09*, New York, NY, USA, 2009, pp. 45–55.

[8] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *ICS '10*, New York, NY, USA, 2010, pp. 137–146.

[9] M. Becchi *et al*, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *SPAA '10*, New York, NY, USA, 2010, pp. 82–91.

[10] J. Stratton, S. Stone, and W. mei Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," in *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, July 2008. [Online]. Available: http://www.gigascale.org/pubs/1328.html

[11] "PGI CUDA-X86 Compiler," http://www.pgroup.com/resources/cuda-x86.htm.

[12] M. Harvey and G. D. Fabritiis, "Swan: A tool for porting cuda programs to opencl," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093 – 1099, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465511000117

[13] S. Lee and R. Eigenmann, "OpenMPC: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.36

[14] S. Che *et al*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC '09*, Washington, DC, USA, 2009, pp. 44–54.

[15] "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php.

[16] A. Kerr *et al*, "Modeling GPU-CPU workloads and systems," in *GPGPU '10*, 2010, pp. 31–42.

[17] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling - a status report," in *JSSPP*, 2004, pp. 1–16.

[18] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, pp. 25–36, May 1996.

[19] V. K. Naik, M. S. Squillante, and S. K. Setia, "Performance analysis of job scheduling policies in parallel supercomputing environments," in *Supercomputing '93*, New York, NY, USA, 1993, pp. 824–833.

[20] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, "Scheduling of parallel jobs in a heterogeneous multi-site environment," in *in Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 87–104.

[21] G. Sabin, V. Sahasrabudhe, and P. Sadayappan, "Assessment and enhancement of meta-schedulers for multi-site job sharing," in *HPDC '05*, Washington, DC, USA, 2005, pp. 144–153.

[22] R. Kumar *et al.*, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 64–75.

[23] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *CF '06*, New York, NY, USA, 2006, pp. 29–40.

[24] "SLURM: A highly scalable resource manager," https://computing.llnl.gov/linux/slurm/slurm.html.

[25] K. Furlinger, N. J. Wright, and D. Skinner, "Comprehensive performance monitoring for gpu cluster systems," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1377–1386. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.289

[26] W. Jiang and G. Agrawal, "MATE-CG: A mapreduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *IPDPS '12 (to appear)*, 2012.

[27] "Torque Resource Manager," http://www.clusterresources.com/products/torque-resource-manager.php.

[28] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, ser. HCW '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 30–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795690.797893

[29] A. B. Downey and D. G. Feitelson, "The elusive goal of workload characterization," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, pp. 14–29, March 1999. [Online]. Available: http://doi.acm.org/10.1145/309746.309750

[30] R. F. t. Freund, "Scheduling resources in multi-user, heterogeneous, computing environments with smartnet," in *Proceedings of the Seventh Heterogeneous Computing Workshop*, ser. HCW '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 3–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795689.797878

[31] W. Cirne and F. Berman, "Using moldability to improve the performance of supercomputer jobs," *JPDC*, vol. 62, no. 10, pp. 1571–1601, 2002.

[32] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan, "Effective selection of partition sizes for moldable scheduling of parallel jobs," in *HiPC*, 2002, pp. 174–183.

[33] I. Ahmad, M. Dhodhi, and R. Ul-Mustafa, "DPS: Dynamic priority scheduling heuristic for heterogeneous computing systems," *IEE Proceedings - Computers and Digital Techniques*, vol. 145, no. 6, pp. 411–418, 1998. [Online]. Available: http://link.aip.org/link/?ICE/145/411/1

[34] Y. S. Kee, H. Casanova, and A. Chien, "Realistic modeling and synthesis of resources for computational grids," in *SC '04*, 2004, pp. 54–63.

[35] Y. S. Kee, K. Yocum, A. A. Chien, and H. Casanova, "Improving grid resource allocation via integrated selection and binding," in *SC '06*, 2006.

[36] F. Berman *et al*, "New grid scheduling and rescheduling methods in the grads project," *International Journal of Parallel Programming*, pp. 209–229, 2005.

[37] S. K. Garg, R. Buyya, and H. J. Siegel, "Scheduling parallel applications on utility grids: Time and cost trade-off management," in *ACSC '09*, 2009, pp. 139–147.