A Waterfall Model to Achieve Energy Efficient Tasks Mapping for Large Scale GPU Clusters

Wenjie Liu[1], Zhihui Du[1*], Yu Xiao[2], David A. Bader[3], and Chen Xu[2]

[1]*Tsinghua National Laboratory for Information Science and Technology*
Department of Computer Science and Technology, Tsinghua University, 100084, Beijing, China
[*]*Corresponding Author's Email: duzh@tsinghua.edu.cn*
[2] *Beijing University of Posts and Telecommunications, China*
[3]*College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA*

Abstract—**High energy consumption has become a critical problem for supercomputer systems. GPU clusters are becoming an increasingly popular architecture for building supercomputers because of its great improvement in performance. In this paper, we first formulate the tasks mapping problem as a minimal energy consumption problem with deadline constraint. Its optimizing object is very different from the traditional mapping problem which often aims at minimizing makespan or minimizing response time. Then a Waterfall Energy Consumption Model, which abstracts the energy consumption of one GPU cluster system into several levels from high to low, is proposed to achieve an energy efficient tasks mapping for large scale GPU clusters. Based on our Waterfall Model, a new task mapping algorithm is developed which tries to apply different energy saving strategies to keep the system remaining at lower energy levels. Our mapping algorithm adopts the Dynamic Voltage Scaling, Dynamic Resource Scaling and *β*-migration for GPU sub-task to significantly reduce the energy consumption and achieve a better load balance for GPU clusters. A task generator based on the real task traces is developed and the simulation results show that our mapping algorithm based on the Waterfall Model can reduce nearly 50% energy consumption compared with traditional approaches which can only run at a high energy level. Not only the task deadline can be satisfied, but also the task execution time of our mapping algorithm can be reduced.**

Keywords—mapping algorithm, Dynamic Voltage Scaling, GPU cluster

I. INTRODUCTION

Because of the lower cost, massively parallel hardware architecture and higher computing capability, GPUs gradually are becoming the mainstream accelerators in the heterogeneous supercomputing environment. And the high performance of floating point arithmetic and memory operations on GPUs make them particularly well-suited to many of the scientific and engineering workloads that occupy GPU clusters [22] [23] [24].

Beside the cost-efficiency of GPU cluster, it also has the potential to significantly reduce space, power, and cooling demands. Nvidia's commercial "Tesla" GPU with four high performance GPU cores is particularly tailored for GPU clusters. In addition, many modern supercomputers in the Top 500 list are highly-tuned clusters using commodity processors combined with heterogeneous accelerators. Given all of the above, we can expect that GPU clusters will become the main stream of the future HPC cluster and supercomputer deployment.

Over the past decade, many efforts are made to find efficient task mapping algorithms for heterogeneous computing environments, such as in supercomputers or large scale HPC clusters. Following the terminology of the scheduling related research, the *matching* tasks to cluster nodes and *scheduling* the execution order (inside node) of these tasks are combined referred to as *mapping*. As we know, the task scheduling problem with data dependencies or without data dependencies has been proven to be NP-complete [27]. Therefore, many existing and well-known heuristics algorithms [8][15] [19][20][25] are proposed to achieve the near optimal results for some scheduling objectives, such as *makespan* of the schedulable tasks set (the overall completion time of these tasks), Quality of Service (such as task deadlines or task priorities) , resource utilization and energy consumption.

Recently, more research has focused on the mapping algorithm in the hybrid system deploying several GPU cards. *Qilin* [6] features adaptive mapping technique in the hybrid system with only one CPU and one GPU by which the computation-to-processor mapping is dynamically determined by the runtime according to the problem size of the current task and the processor computing capability, choosing either the available CPU or GPU implementation. In [4][5], the authors designed a scheduling algorithm based on task speculation on multi-GPU systems and this scheduler can dynamically map GPU tasks to accelerators with potentially heterogeneous architectures. This scheduler does not consider the mapping problem combined with CPU applications or general tasks with both CPU and GPU implementations. In [14], the authors proposed a novel predictive scheduler based on past performance history for heterogeneous system with multiple GPUs. Besides, some works in [8], StarPU [11] also focus on the scheduling problem in multi-GPU systems.

Those algorithms are efficient for their target platform; however, when the hardware system contains much more nodes and each with many CPUs and many GPUs such as GPU cluster deployments, new algorithms must be developed to take advantage of the new hardware features. The scheduling problem in GPU clusters, and even in multi-GPU systems in the same machine, shows more difficulties and challenges than in the traditional heterogeneous system and we have not found published papers which pay attention to the scheduling problems in the large scale GPU cluster with many computing nodes. Some algorithms [6] mentioned the energy efficiency, but they did not consider the energy saving from the overall system prospective. In fact, the huge energy consumption has become an outstanding and serious problem in the high performance computing environment [25] and continues to plague the users and researchers in GPU clusters [24].

In this paper, we provide the Waterfall Energy Consumption Model (simply called Waterfall Model) which divides the energy consumption of overall system into three different levels according to different energy saving strategies deployed. Compared with traditional systems without any energy saving mechanism, our model can achieve energy savings by employing both the coarse-grained method and fined-grained method together. The key idea of our mapping algorithm based on the Waterfall Model for GPU clusters is trying to apply different energy saving strategies to keep the system remaining at lower energy levels. More specifically, we make the following contributions.

- We put forward and formulate the energy efficient mapping problem for independent tasks with deadline constraints on large scale GPU cluster systems.
- We develop the Waterfall Model which abstracts the system energy consumption into several energy levels and develop an efficient energy saving mapping algorithm for large scale GPU cluster systems based on this model
- We take advantage of some coarse-grained and fine-grained strategies together to reduce the energy consumption of GPU clusters: DRS (Dynamic Resource Scaling) for computing nodes, DVS (Dynamic Voltage Scaling) for CPUs, and β−migration for GPU sub-tasks.

As our simulation results show, our mapping algorithms with these strategies can provide much more benefits (nearly 50% energy saving) in term of energy consumption.

The rest of this paper is organized as follows. In section 2 we introduce some related works on scheduling algorithms. We give the overview of the task mapping problem on GPU cluster in section 3. The Waterfall Model is described in section 4. In section 5, we propose our scheduling algorithm in detail in a GPU cluster. Our simulation results are provided in section 6 and we conclude the paper in section 7.

II. Related work

A. Mapping Algorithms in Heterogeneous Systems

The task mapping (defined as matching and scheduling) problem in heterogeneous systems is a classic problem and has been studied for many years. Because of its intractable nature, much of the research focuses on heuristics. For the case of tasks without data dependencies (called meta-task in some publications), in [25], the authors summarize eleven static heuristics for mapping a class of tasks onto heterogeneous systems, such as Opportunistic Load Balancing (OLB, which tries to keep every machine as busy as possible), Minimum Execution Time (MET, select the best machine for each task), Minimum Completion Time (MCT, combines the benefits of OLB and MET), Min-min heuristic (similar to Shortest Task First scheduling), Max-min heuristic (Longest Task First), etc. Many later mapping algorithms make some modification and improvement based on these heuristics. In [15], the authors compare the mapping algorithms between dynamic mapping and static mapping, online mode and batch mode. They introduce several typical algorithms for both dynamic and static, immediate mode and batch mode. In addition, they introduce some applications of these mapping algorithms. All of these discussions inspire us to solve the mapping problems for the GPU cluster.

The Directed Acyclic Graph (DAG) tasks (tasks with data dependency) mapping problem is another significant topic in the heterogeneous computing. There are also many well-known heuristics proposed. In [20], the authors present two scheduling algorithms for a bounded number of heterogeneous nodes, which are called the Heterogeneous Earliest-Finish-Time (HEFT) algorithm and the Critical-Path-on-a-Processor (CPOP) algorithm. According to the evaluation results in [20], these two algorithms surpass other works in several metrics such as schedule length ratio, speedup, frequency of best results, and average scheduling time metrics. These two algorithms have already been applied to many systems and have significant impact in the research community [8][19][20].

B. Scheduling Algorithms on Multi-GPU systems

Because of the importance of task scheduling algorithms, many scheduling algorithms are proposed in order to enhance the system performance at multiple GPUs platform. Below, we will introduce some related works.

1) *Qilin system* : A DAG graph of kernels is created by a runtime component as the program executes. *Qilin* features an adaptive mapping technique by which the computation-to-processor mapping is dynamically determined by the runtime, choosing either the available CPU or GPU implementation. However, *Qilin* only provides its results on one *CPU-GPU* pair system and we focus on large-scale GPU clusters where each node may have many *CPU-GPU* pairs.
2) *Scheduling based on Speculation:* Harmony [4][5] provide an algorithm based on speculation on multi-GPU systems and this scheduler can dynamically map kernels to accelerators with potentially heterogeneous architectures. To achieve this, the software branch prediction coupled with the memory renaming mechanism is used to distinguish between the speculative and non-speculative state to enhance the program concurrency. However, this method focuses on multi-GPU system and it only considers the scheduling of GPU applications (called GPU computing kernels and no CPU instructions).
3) *Scheduling based on Performance History*: A novel predictive scheduler based on past performance history on the CPU/GPU-like systems is proposed in [14]. It is stated that this approach can fully utilize all available processing resources and achieve speed improvements ranging from 30% to 40% compared to the usage of the GPU alone in a single application mode. Like the Harmony, it only considers the GPU applications in some very simple scheduling situations, such as First Come First Served (FCFS), and scheduling based on past performance.

III. Overview of the Mapping Problem

A. GPU cluster model

Our mapping algorithm mainly focuses on the GPU cluster. Next we discuss a more accurate abstraction of the GPU cluster.

Many GPU clusters have been deployed in the past few years, such as the 160-node "DQ" GPU cluster at LANL [22] and the 16-node "QP"GPU cluster at NCSA [23]. In [24], the authors introduced two GPU cluster deployments in detail: the 192-node cluster "Lincoln" and the 32-node cluster "AC". This paper shows that a one-to-one ratio of CPU processors to GPUs , a much important feature in GPU deployment, may be desirable from the software perspective , as it greatly simplifies the development of MPI-based applications and also greatly simplifies the mapping algorithm. A one-to-one ratio of CPU processors to GPUs means that when a given task is pushed into a compute node, we can assign a CPU-GPU pair to execute this task to achieve high performance and much less energy consumption. Therefore, we assume that the number of CPUs (meaning CPU processor and one CPU may have multiple cores) and GPUs are equal inside a compute node.

In most of the GPU clusters mentioned above, every computing node is homogeneous and has the same configuration. A k node GPU cluster can be formulated as a set

$$\{n_1, n_2, n_3, \cdots, n_k\}$$

Here, for each node n_h ($1 \le h \le k$) in a GPU cluster which consists of m CPUs (homogeneous) and m GPUs (homogeneous), the m CPUs are formulated as a CPU set for node n_h

$$\{C_{h,1}, C_{h,2}, \cdots, C_{h,m}\}$$

and the m GPUs for node n_h are formulated as

$$\{G_{h,1}, G_{h,2}, \cdots, G_{h,m}\}$$

We have in total m *CPU-GPU* pairs. All of the nodes are connected in a fully connected topology using a high speed interconnect.

We add two energy efficient properties for the GPU clusters in our Waterfall Energy Consumption Model to further reduce the energy consumption. Dynamic Voltage Scaling (DVS) for CPU in a computing node and Dynamic Resource Scaling (DRS) for different volumes of tasks in a GPU cluster.

- *DVS enabled GPU Clusters*: the DVS scheme is an efficient way to reduce dynamic power consumption by adjusting the supply voltage in an appropriate manner. Much research has been done to power-aware cluster computing by using the DVS scheme [1][2][3][4]. We adopt the DVS feature of CPU (unavailable for GPU because currently none of such product is available) in the GPU clusters and will discuss the usage of above strategies for energy reduction in later part.
- *DRS enabled GPU Clusters*: Each node in a GPU cluster can be transitioned among three states, respectively *busy*, *spare*, and *sleep*. When all of CPUs or GPUs inside this node are executing some tasks, this node is in the *busy* state. When at least one of the *CPU-GPU* pair is free (but ready to execute a task immediately), this node is in the *spare* state. When a node in *spare* state does not have any task to run on any *CPU-CPU* pair, this node will be powered down and changes its state from *spare* to *sleep*. The node in the *sleep* state can significantly reduce the energy consumption. When more tasks arrive, the sleeping node can be woken up and its state will be changed from *sleep* to *spare* or *busy*. The DRS (Dynamic Resource Scaling) mechanism can dynamically adjust the number of working nodes to save energy.

B. Tasks Model

We assume that all the incoming tasks are independent with each other without data communication. All the tasks may follow different distributions. For each task t, it has two sub-tasks, CPU-subtask t_{CPU} and GPU-subtask t_{GPU}. The CPU subtask is the part which is suitable for running on CPU and the GPU subtask is the part which is suitable for running on GPU. If we let w be the total processing volume of t and α be the percentage of the CPU subtask, then the processing volume of the CPU subtask should be $w \cdot \alpha$ and the processing volume of the GPU subtask should be $w \cdot (1\text{-}\alpha)$. Let r be the release time of task t, d be the required deadline of t. Then a task can be formulated as a tuple

$$t = <\alpha, w, r, d>$$

All the μ tasks that are to be executed can be formulated as a set

$$\{t_1, t_2, \cdots, t_\mu\}$$

All the tasks can be freely assigned to any *CPU-GPU* pair in a given node as a whole. The CPU subtask and the GPU subtask of the same task will not be assigned to different nodes for simplicity and reduced overhead. For the same reason, we adopt the non-preemptive scheduling mechanism which means that once a task is running on one selected *CPU-GPU* pair, it will keep running to the end of this task, and even high priority tasks cannot replace it to run on this *CPU-GPU* pair.

C. Energy Consumption Model for GPU cluster

With the appearance of large scale GPU clusters, there are increasing and even critical requirements on lowering the total system energy consumption. The main power consumption in CMOS circuits is composed of dynamic and static power for both CPU and GPU. The static power goes for the idle circuit which has no workload while the dynamic power goes for the circuit with executing tasks.

EN_h, the total energy consumption for computing node n_h , can be expressed in Equation (1). en_C^i is the energy consumption of $C_{h,i}$ and en_G^i is the energy consumption of $G_{h,i}$. $EN_{overall}$, the overall energy consumption of a given GPU cluster is given in Equation (2). Recall that each node n_h can have three states: *busy* which means that all the *CPU-GPU* pairs are running; *spare* which means that part of the *CPU-GPU* pairs are running and the other *CPU-GPU* pairs are idle; *sleep* which means that all of the *CPU-GPU* pairs are powered down (the energy consumption can be ignored). Thus, the energy consumption only needs to consider the *busy* and *spare* nodes.

$$EN_h = \sum_{i=1}^{m} (en_C^i + en_G^i) \qquad (1)$$

$$EN_{overall} = \sum_{h=1}^{k} EN_h \qquad (2)$$

The CPU runtime power model is discussed in reference [18]. Here, let the static power consumption be *power*(C,s) , the dynamic power consumption be *power*(C,d) and the total lasting time be t_{total} which includes execution time t_{run} and idle time t_{idle}. So we can calculate the i^{th} CPU energy consumption in node n_h as follows:

$$en_c^i = \left(power(C,s) + power(C,d)\right) \times t_{run}^i + power(C,s) \times t_{idle}^i = power(C,s) \times t_{total}^i + power(C,d) \times t_{run}^i \quad (3)$$

But for the DVS-enabled CPU processor, the energy consumption E which is expressed in Equations (4) and (5) is related to the supply voltage V based on the well-known cube-root rule for CMOS devices [1][2].

$$E = C_{eff} \cdot f_C \cdot V^2 \cdot t \quad (4)$$

$$f_C = kV \quad (5)$$

where C_{eff} is the effective switching capacitance of this chip, f_C is the processing speed of the CPU, and t is the processing time.

Like the CPU energy consumption model, in [13], the authors also propose a novel GPU dynamic power model. When the static power consumption is *power*(G,s) and the dynamic power consumption is *power*(G,d) . We model the GPU energy consumption as follows:

$$en_G^i = \left(power(G,s) + power(G,d)\right) \times t_{run}^i + power(G,s) \times t_{idle}^i = power(G,s) \times t_{total}^i + power(G,d) \times t_{run}^i \quad (6)$$

D. Formulation of the Energy Efficient Mapping Problem with Deadline Constraint

Based on the GPU cluster model, task model and the energy consumption model, the energy efficient mapping problem can be formulated as an optimization problem with deadline constraint. It can be expressed in Equation (7).

$$Min: EN_{overall}$$
$$s.t. \forall\, t_x \in ReadyTaskSet, FinishTime(t_x) \leq d(t_x) \quad (7)$$

In summary, we try to map all the incoming tasks onto a GPU cluster with minimal energy consumption where the deadline of each task must be satisfied.

IV. Concept of the Waterfall Model

A. Mapping Problem Overview

The mapping heuristics can be grouped into two categories: immediate-mode and batch-mode. In the immediate-mode, a task is mapped onto a machine as soon as it arrives at the mapper. In the batch mode, tasks are not mapped onto the machines as they arrive; instead they are collected into a set that is examined to execute mapping triggered by some mapping events. Because the batch-mode outperforms the immediate-mode in most cases, we only consider the batch-mode in this paper.

Consider μ ready tasks $t_1, t_2, t_3, \cdots, t_\mu$, that have to be processed on a GPU cluster. Here, we adopt the common expression of ready tasks similar to reference [15]. For the v^{th} mapping event, the ready tasks set M_v is mapped at time τ_v. The ready tasks set M_v, for $v > 0$, consists of tasks that arrived after the last mapping event and the tasks that were mapped but did not start executing. Let b be the beginning execution time of t_x. M_v can be expressed as follows:

$$M_v = \{t_x | \tau_{v-1} \leq r(t_x) < \tau_v\} \cup \{t_x | r(t_x) < \tau_{v-1} \wedge b(t_x) > \tau_v\} \quad (8)$$

We propose some important parameters to simplify our mapping algorithm, respectively:

1) *Δ-value*: we set *Δ-value* for given task t_x expressed as
$$\Delta(t_x) = d(t_x) - CompletionTime(t_x) \quad (9)$$
where $CompletionTime(t_x)$ is the finishing time needed to run a given task t_x on a idle *CPU-GPU* pair.
2) *Task distribution ratio θ*: For a given task t_x executing on a *CPU-GPU* pair, let $Exe(C, t_x)$ be the execution time of CPU subtask on CPU C and $Exe(C, t_x)$ be the execution time of GPU subtask on GPU G. The task distribution ratio depicts the processing volume distribution of this task between CPU and GPU. It can be expressed as:
$$\theta(t_x) = \frac{Exe(C,t_x)}{Exe(G,t_x)} \quad (10)$$
3) *Earliest available time for given node*: The earliest completion time of any *CPU-GPU* pair in the node n_h is expressed as $C\text{~}G_e^h$. It depicts the earliest available time of one *CPU-GPU* pair for this computing node.
4) *Latest available time for node*: The latest completion time of any *CPU-GPU* pair in the node n_h is formulated as $C\text{~}G_l^h$. It gives the latest available time of one *CPU-GPU* pair for this computing node.
5) *Load Balance Displacement π: Let* $\pi(n_h)$ be the load balance displacement for node n_h which can be formulated as
$$\pi(n_h) = C\text{~}G_l^h - C\text{~}G_e^h \quad (11)$$

Overall, there are two kinds of events to trigger a new scheduling procedure: task deadline triggered event and counter triggered event.

1) *Deadline Triggered Event:* we consider a hard real time system which must meet all incoming tasks' deadline requirements. When the deadline of a waiting task will not be met by any idle *CPU-GPU* pairs, a scheduling event occurs. We use the *Δ-value* of the task to measure this urgency. When $\Delta(t_x) \leq \varepsilon, \varepsilon$ *is a small positive value , a mapping event is triggered* and the mapper must map this task into the appropriate *CPU-GPU* pair immediately.
2) *Counter Triggered Event:* Besides the deadline triggered event, we set a Task Counter to trigger the mapping event. When an arriving task makes $|M_v|$ greater than or equal to predetermined number *Constant*, a mapping event occurs. This value of *Constant* is a constant and it is related to the arrival rate and completion rate of the tasks.

B. Prediction Performance Model

The assumption that these expected execution times are known is commonly made when studying mapping heuristics for heterogeneous computing systems. We also make this assumption.

The approaches to getting the CPU estimated execution times for a given task based on task profiling and analytical benchmarking have been proposed for many years, and are discussed in references [9][10]. Fortunately, like CPU implementations, there are also several current performance models to predict GPU execution time. In [12], the authors propose an analytical performance model which greatly depends on the GPU features and characteristics. Despite the accurate results guaranteed, this model greatly involves the details such as the GPU architecture and application implementations. So this model will incur more difficulties and overheads as the program's complexity grows. In the *Qilin* system, they exploit a linear fitting method to predict the execution time on GPU. They assume that the execution time is linear to the problem size of a task. But this is not true in many cases.

In this paper, based on the core features of above approaches, we adopt another way to more practically and accurately predict the execution time on GPU and CPU. We integrate and unify the features between CPU and GPU, and propose the normalized processing volume $w(t)$ for a given task. Therefore, if the CPU processor speed is $f_{C,i}$, $1 \leq i \leq n$; and GPU processor speed is $f_{G,j}$, $1 \leq j \leq m$, then the expected time of task t on C_i and on G_j is expressed as follows:

$$T_C(t,i) = \frac{w(t)}{f_{C,i}} + S(t,i) \quad (12)$$

$$T_G(t,j) = \frac{w(t)}{f_{G,j}} + S(t,j) \quad (13)$$

where $S(t,i)$, $S(t,j)$ are constant factors which stand for the start up time on C_i and G_j for the given task t.

C. *Waterfall Energy Consumption Model*

Inspired by the features of a waterfall, which tries to drop down to lower potential energy level as soon as possible, we provide the Waterfall Model to express our basic idea on an energy efficient task mapping mechanism.

Our Waterfall Model divides the system energy consumption into three different energy level, *high, middle* and *low* energy levels. We define the traditional model as the one which does not adopt any energy efficient strategies and it will always keep the whole system in the *high* energy level. The system based on the Waterfall Model can employ some coarse-grained energy efficient strategies focusing on computing nodes, such as the DRS (Dynamic Resource Scaling) mechanism (a wakeup-sleep mechanism for computing nodes). Employing such coarse-grained strategies will keep the system in the *middle* energy level which means that it can consume less energy consumption than the traditional model, but still has the potential to improve the energy efficiency. Based on the coarse-grained strategies, we propose some other fine-grained strategies focusing on the CPU-GPU pair, which will keep the system in *low* energy level, such as DVS strategies and β-migration for GPU subtasks. We summarize the Waterfall Model in Fig.1. Our heuristic algorithm is designed based on this model.

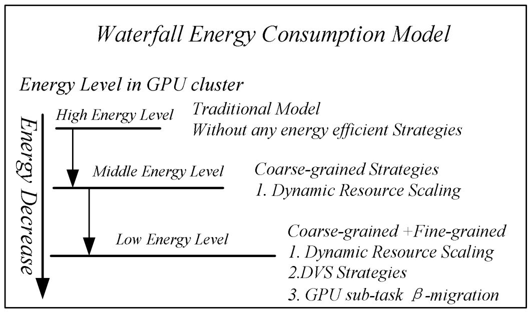
Waterfall Energy Consumption Model
Energy Level in GPU cluster
High Energy Level
Traditional Model
Without any energy efficient Strategies
Middle Energy Level
Coarse-grained Strategies
1. Dynamic Resource Scaling
Low Energy Level
Coarse-grained +Fine-grained
1. Dynamic Resource Scaling
2.DVS Strategies
3. GPU sub-task β-migration
Energy Decrease

Fig. 1: Waterfall Energy Consumption Model

V. Mapping Algorithms

Our mapping algorithm (mapper) targeting on GPU clusters will experience three phases: in the first phase, the mapper selects the appropriate task from the schedulable task set M_v and assigns it into the optimal CPU-GPU pair (C_o, G_o). In the second phase, the mapper employs the DRS mechanism to wake up or power down some nodes if necessary. In the third phase, the mapper takes the DVS mechanism and β-migration for GPU subtask to adjust the execution of this task.

A. *Matching Process*

The matching process is the core part in the mapping process. It involves how to select appropriate *CPU-GPU* pair for every task and how to select a task from the ready task set.

1) *CPU-GPU pair selection*: Firstly, we need to select he *CPU-GPU* pair for each task in the ready task set. Here, let the available time of CPU and GPU be $Avail(C_i^h)$ and $Avail(G_j^h)$, where $1 \leq h \leq k$; $1 \leq i, j \leq m$. Therefore, the completion time for a task t_x on *CPU-GPU* pair (C_i, G_j) on node h is given as follows:

$$ct(t_x, C_i^h, G_j^h) = Max\{Avil(C_i^h) + T_C(\alpha \cdot w_x, i), \\ Avil(G_j^h) + T_G((1-\alpha) \cdot w_x, j)\} \quad (14)$$

We select the optimal *CPU-GPU* pair (C_o, G_o) to minimize the completion time of this task t_x :

$$optimal(C_0, G_0) = Min\{ct(t_x, C_i^h, G_j^h)\} \quad (15)$$

In addition, there are still some things to be noted. If there are two nodes (n_k, n_h) with $C{\sim}G_e^h = C{\sim}G_e^k$ but $\pi(n_h) > \pi(n_k)$, then we will select the *CPU-GPU* pair on node n_h. Intuitively, this method can avoid creating larger unbalanced load on existing node so help to make more nodes transitioned into *sleep* state.

2) *Task selection based on Δ-value*: As mentioned before, the ready schedulable tasks currently are denoted as a set M_v. At time τ_{v-1} the scheduling event occurs. For each task t_x in the ready schedulable set M_v, in order to meet its deadline, the task t_x with minimum *Δ-value* will be scheduled first. This strategy is inspired by the well-known Earliest Deadline First (EDF) [17] algorithm, but we combine the deadline and the completion time for a task and we always schedule the task with the minimum *Δ-value* firstly. Intuitively, our strategy tries to schedule more urgent task

(task with smaller *Δ-value*) at the same time considering the computing capability (completion time) of selected *CPU-GPU* pairs.

B. Dynamic Resource Scaling Algorithm

Every computing node will transition between three states, respectively *busy*, *spare* and *sleep*. In order to reduce the energy consumption, we need to keep much fewer nodes executing more workload and try to place more nodes in *sleep*. In other words, we need make the number of nodes in *spare* state as small as possible. If the scale of the GPU cluster is large but the workload is too small, it is obviously unwise to power up all nodes to execute this work. Therefore, a Dynamic Resource (computing nodes) Scaling algorithm based on the Waterfall Model is proposed. Although it is a relatively coarse-grained method, many existing data centers exploit a similar method to reduce the energy consumption in the implementation of cloud computing or virtualization [16]. Our Dynamic Resource Scaling algorithm is illustrated in Fig. 2, where $S(k)$ stands for the current state of the node n_k and current scheduled task is a task in *ReadyTaskSet* with minimum *Δ-value*, as is discussed in task selection section.

Dynamic Resource Scaling algorithm:

```
1:  if ($d(t_x)$-$ct(t_x, C_o, G_o)$ <0)
        Means no busy or spare node can meet the deadline of $t_x$
2:  then wake up a new node $n_k$ whose $S(k)$ = sleep
3:          change its state $S(k)$ into spare
4:          schedule $t_x$ onto this node
5:  else assigned $t_x$ onto the node $n_h$ with minimal $C\sim G_e^h$
        and maximal $\pi(n_h)$
6:      if all CPU-GPU pairs of node $n_h$ are busy
7:      then set $S(h)$ = busy
        endif
8:  endif
9:  for each node $n_h$ in the GPU cluster
10:     if all CPU-GPU pairs are idle
11:         then power down this node and set $S(h)$ = sleep
12:     endif
13:     if some CPU-GPU pairs are idle and some are busy
14:         then set $S(h)$ = spare
15:     endif
16: end for
```

Fig. 2: Dynamic Resource Scaling Algorithm

C. Voltage selection based on DVS mechanism

In real-time systems with QoS constraints, the DVS technique is used in order to save the energy consumption as well as to meet the task deadline. The basic idea is to slowdown the CPU speed and make the task completion moment just fit the task deadline. The DVS mechanism of CPU is exploited and applied ubiquitously for many years [1][2][3], but the DVS technology of GPU is still not possible. Therefore, we adopt the DVS mechanism for the CPU processor only. We assume that the voltage levels in this paper are discrete from V_1, to V_s (which satisfies $V_1<V_2<\ldots<V_s$). According to the cube-root rule and DVS energy consumption in Equation (4), the lower the CPU voltage level, the less the energy consumption.

Here, the task distribution ratio $\theta(t_x)$ of *CPU-GPU* pair for the given task t_x is expressed as $\theta(t_x) = \frac{T_C(\alpha w_x, i)}{T_G((1-\alpha)w_x, j)}$. Our mapping purpose is to achieve better load balance between CPU and GPU, which means the closer $\theta(t_x)$ is approaching 1, the better the load balance is.

The dynamic DVS voltage selection algorithm is illustrated in Fig. 3, where C_o stands for the CPU of the optimal *CPU-GPU* pair for this task.

Dynamic Voltage Scaling algorithm:

```
1:  for each supply voltage $V_i$ from $V_s$ down to $V_1$
2:      do re-calculate the execution time at voltage $V_i$
3:          $Exe(C_0, t_x, V_i)$
4:  end for
5:  Let Selected Voltage $SV = V_s$
6:  for each supply voltage $V_i$ from $V_{s-1}$ to $V_1$
7:      if $|\theta(t_x, V_i)-1|<|\theta(t_x, SV)-1|$ then $SV = V_i$
8:      endif
9:  end for
```

Fig. 3: DVS voltage selection algorithm.

D. β-migration for GPU Sub-tasks

As is mentioned before, the typical implementation of a task on GPU cluster contains two parts: CPU sub-task and GPU sub-task. According to the empirical facts and detailed application, CPU sub-task always includes some controlling/logical instructions, sequential instructions, some instructions with much higher loop execution overhead, or frequent I/O operation instructions, etc. The GPU sub-task always includes high data parallel instructions, and is executed on GPU to achieve a larger speedup, such as scientific computation with a high degree of data parallelism and much less communication.

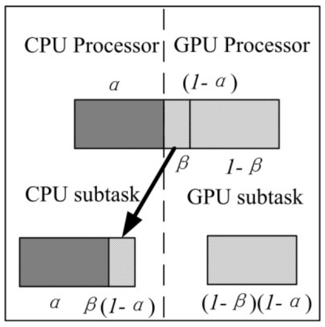

Fig. 4: β-migration for GPU Sub-task

For the given task on the optimal *CPU-GPU* pair, the unequal completion time of the CPU sub-task the GPU sub-task will lead to wasted energy and computing resources. In most cases the CPU can execute certain portions of the GPU work, while the GPU cannot execute the CPU subtask because some CPU instructions are not supported by GPU. Therefore, we consider dividing the GPU sub-task and migrating β fraction to the idle CPU in this CPU-GPU pair, as illustrated in Fig. 4. This strategy is called β–migration for GPU sub-task (shortly called β–migration). In the *Qilin* system, they adopt a similar method to divide the application into CPU and GPU parts.

Next we discuss how to migrate β fraction of the GPU sub-task from GPU to CPU on optimal CPU-GPU pair. Let $ct(t_x, C_o, G_o)$ be the completion time for the task t_x, and it is calculated by Equation (14). After the β–migration, the fraction of CPU sub-task is adjusted into $(\alpha+\beta(1-\alpha))$ and the fraction of sub-task executing on GPU is $(1-\alpha)(1-\beta)$. Therefore the completion time of this task is changed into Equation (16). Then we select an optimal value of β to minimize the completion time of task t_x, and we get the optimal β value given in Equation (17).

$$ct\left(t_x, C_i^h, G_j^h, \beta\right) = Max\{Avail\left(C_i^h\right) + T_c\left(\left(\alpha+\beta(1-\alpha)\right)w_x, i\right), Avail\left(G_j^h\right) + T_G\left((1-\beta)(1-\alpha)\right)w_x, j)\} \quad (16)$$

$$\beta_{optimal} = min_\beta\, ct\left(t_x, C_i^h, G_j^h, \beta\right) \quad (17)$$

Obviously, when $Avail\left(C_i^h\right) + T_c\left(\left(\alpha+\beta(1-\alpha)\right)w_x, i\right)$ and $Avail\left(G_j^h\right) + T_G((1-\beta)(1-\alpha))w_x, j)$ are equal, this β achieves its optimal value, which means the sub-tasks on CPU and GPU will be finished at the same time, when the task distribution ratio $\theta(t_x)$ of *CPU-GPU* pair is accurately equal to 1. Therefore, the load balance of the system is greatly improved. When β achieves its optimal value $\beta_{optimal}$, we have $\theta(t_x) = 1$.

Task Mapping Algorithm for GPU clusters:

```
1: do until all tasks in $M_v$ are mapped
2:     Find the task $t_{min}$ with minimum $\Delta(t_x)$ in $M_v$
3:     Apply the DRS algorithm for $t_{min}$
4:     Assign task $t_{min}$ to the optimal CPU-GPU pair ($C_o$,$G_o$)
5:     Apply the DVS algorithm for $C_o$
6:     Apply sub-task $\beta$-migration from $G_o$ to $C_o$
7:     Delete the task $t_{min}$ from $M_v$
8: end do
```

Fig.5: Task Mapping Algorithm in GPU cluster.

E. Overall Task Mapping Algorithm

Now, we propose the complete process of our mapping algorithm focusing on GPU cluster based on the above discussion. The mapping process includes selecting the urgent task as current task to be scheduled, applying DRS for the current task, selecting optimal *CPU-GPU* pair (C_o, G_o) for current task, applying the DVS for C_o, applying β–migration from G_o to C_o and updating the related parameters. All of them are illustrated in Fig. 5. By applying the complete process, the system energy consumption can be dropped down from high to low level, as shown in Fig. 1.

VI. Evaluation and Simulation results

A. Simulation Methodology

In a way, system performance such as high execution efficiency is what traditional models focus on. Usually, the execution time of tasks will be considered the first metric of those traditional systems. In other words, tasks should be processed as soon as possible. Instead, in the Waterfall Model the greatest concern is the system's energy performance on the premise that all tasks should be finished before their deadlines.

It is known that the mapping strategies will adversely affect the systems' performance. The object of the simulation is to quantify the difference on energy consumption between the system without energy saving strategies and the system adopting our energy efficient mapping strategies. Thus, two systems based on same hardware architecture will process the same tasks in our simulation test. This guarantees that the differences of the simulation results are only caused by the different strategies.

B. Simulation Scenarios

We conduct the simulation in Matlab R2009a. The simulation scenario contains a Task Generator and the two compared system models, the traditional system model in high energy level and the Waterfall Model in lower energy level with both coarse-grained and fine-grained energy efficient strategies. The system running time in the simulation is set to be more than 24 hours.

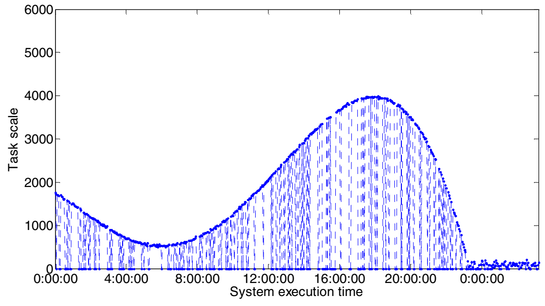

Fig. 6: The number of tasks generated with time in the simulation

In the real world, tasks usually distribute widely in scale (the processing volume for a task discussed before) even in a single day. We choose the statistic data of online tasks from the Modern Education Information Centre at Wuhan University of Science and Technology [28] as our experiment reference. According to the statistics, we fit the task's scale in 24 hours into Equation (18), that is to say, the TaskScale is a function of time x whose unit is second. (i.e., from 0:00:00 to 24:00:00, x is converted to [0 , 1, 2, 3 … 86400]).

$$TaskScale(x) = -0.0001x^6 + 0.0064x^5 - 0.2251x^4 + 2.9466x^3 - 7.634x^2 - 34.53x + 174.7863 \quad (18)$$

Besides, in reality, tasks are not arriving all the time. Similarly, the Task Generator does not produce tasks all the time; instead, it follows the binomial probability. The generating probability of the Task Generator, is *70%*, that is to say, *30%* of the time, the Task Generator will not provide the system with any task. Tasks generated in the simulation are shown in Fig. 6.

What's more, every single task consists of a CPU sub-task and a GPU sub-task. The portion α of CPU sub-task to GPU sub-task is a random value between 0 and 1, conforming to normal distribution $N(0.6, 0.05)$. In addition, every task t_x has a *Δ-value* to depict its deadline decided by Equation (9). *Δ-value* obeys a normal distribution $N(20, 3)$, The *Δ-value* diminishes one every second until it equals to zero.

Both of the models have their own *ReadyTaskSet* and Task Counter. The *ReadyTaskSet* consists of the tasks generated by the Task Generator temporarily and the Task Counter will count every single task once entering into the *ReadyTaskSet*. Thus, the same tasks generated by Task Generator will be sent to the two system models' *ReadyTaskSets* at the beginning. Moreover, two models have the same GPU cluster deployment, which consists of the same 100 Nodes, and each Node contains ten CPU-GPU pairs. CPUs are assumed to possess the DVS policy, and Table I shows the DVS configuration for CPUs in the Waterfall Model.

TABLE I DVS CONFIGURATION OF CPUS IN WATERFALL MODEL

CPU status C_i	Normalized CPU Voltage	Normalized Speedup of CPU	Power
C_1	1	1	95W
C_2	0.7	0.7	60W
C_3	0.3	0.3	40W
C_{sleep}	*0*	*0*	3W

TABLE II SIMULATION PARAMETERS

Parameters	Description	Values
k (#*node*)	The number of Nodes	100
Num_{CPU}	#CPU in a node	10
Num_{GPU}	#GPU in a node	10
$power(C, s)$	Power consumption when CPU is running no task	30W
$power(C, d)$	Power consumption when CPU is running a task at full speed	95W
$power(G, s)$	Power consumption when GPU is running no task	110.13W
$power(G, d)$	Power consumption when GPU is running a task at full speed	236W

In addition, we refer to several GPU clusters hardware configuration in reality, such as Tianhe-1A [29], to calculate the CPU and GPU's computing capability (for example the speedup over serial). The GPU's *spare* state power is calculated according to the *Qilin* system.

The Mapper will start to work when either of two conditions is satisfied: *a*) the Task Counter counts to *10*; *b*) the *Δ-value* of any tasks in the *ReadyTaskSet* equals to *0*. Meanwhile, once these tasks in *ReadyTaskSet* are sent to the Mapper, the *ReadyTaskSet* is released and the Task Counter's value returns to zero.

Every time when tasks are transferred from the *ReadyTaskSet* to the Mapper, we deal with these tasks in increasing order of the tasks' current *Δ-value*. That is because a smaller *Δ-value* indicates a closer time to the task deadline, i.e., the task is more urgent.

C. Differences between Waterfall Model and Traditional Model

There are three differences between the Waterfall Model and Traditional model. They are, *a*) separate scheduling policies; *b*) whether the node owns the Dynamic Resource Scaling (sleep-awake) mechanism; *c*) whether the CPU owns DVS policy and *β-migration*. In other words, Traditional Model in our simulation means the system owns the same hardware architecture as the Waterfall Model but without the DRS mechanism, the DVS policy and *β-migration*.

To be specific, in the Waterfall Model, the right node n_h will be chosen at first, which is the node awake with the minimum $\pi(n_h)$ (when two nodes with equal load balance displacement π, we always select the node with larger $C\text{~}G_l^h$). However, if the minimum available time $C\text{~}G_e^h$ of all the *CPU-GPU* pairs in this node n_h cannot meet the requirement of the task's deadline, a *sleep* node will be woken up to perform the task. Then, the system will perform the Task Mapping Algorithm, choose the suitable CPU voltage (DVS strategy) and calculate the exact β value (*β-migration*), and divide the GPU sub-task properly to obtain the minimum energy consumption.

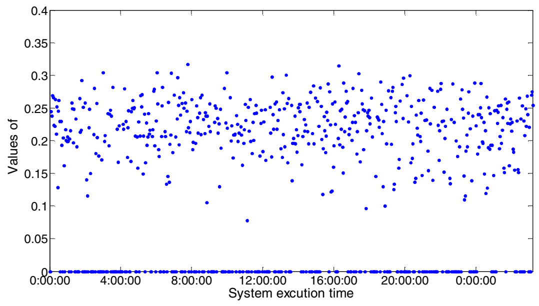

Fig. 7: β values in *β-migration*

However, in the traditional Model, the task is simply delivered to the *CPU-GPU* pair owning the minimum available time without *β-migration* for GPU sub-task. CPU sub-task will be processed by CPU and GPU sub-task will be processed merely by GPU. Moreover, there are no DRS mechanisms for all the nodes and the DVS policy for CPU in the traditional model defined in the simulation, that is to say, all nodes' CPUs of the traditional model only run in two states: one is the *spare* state and the other is the *busy* state with the fixed voltage.

Fig. 7 shows all the β values in *β-migration* calculated in the Task Mapping Algorithm in our Waterfall Model.

D. Energy Consumption Results and Discussion

To some extent, the performance of the task scheduling algorithm could affect a system's total performance greatly. Our goal is to reduce the system's energy consumption in the premise that tasks should be processed before their deadline. In our simulation, we choose the system power and the total system energy consumption as the metrics to evaluate the scheduling strategies. Simulation results on instant system power and the total system energy consumption are shown in Fig. 8 and Fig. 9.

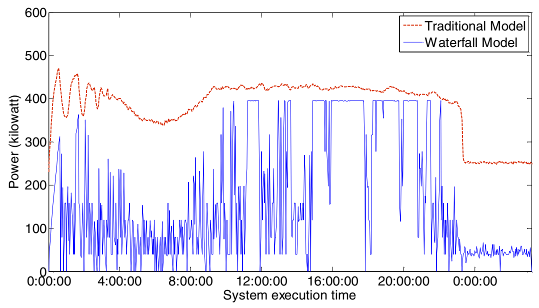

Fig. 8: System power curve over one day

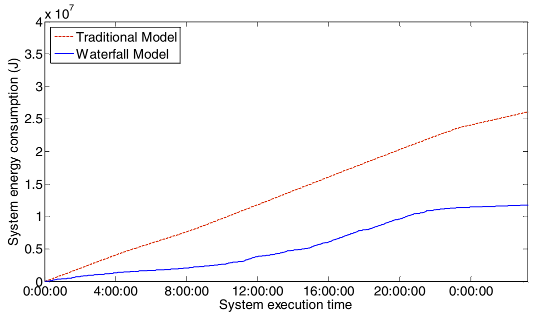

Fig. 9: System energy consumption curve over one day

As we can see in Fig. 8 and Fig. 9, the task mapping algorithm does contribute to reduce the system's power apparently, especially when the task scale is relatively small (see between 4:00:00 and 8:00:00). The reason is that our Waterfall Model will shut down nodes whose *CPU-GPU* pairs are in *spare* state to the *sleep* status (the percentage of sleep-level nodes showed in Fig. 10). Obviously, the smaller task scale, the more sleep nodes.

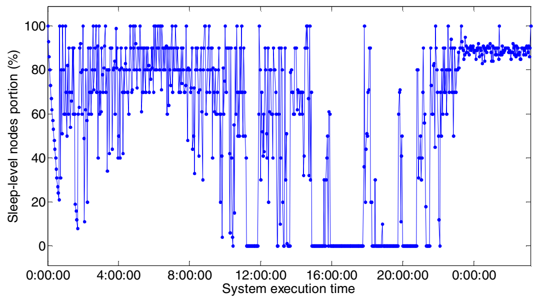

Fig. 10: The percentage of *sleep*-state nodes curve over one day

Even though the power consumption of Waterfall Model increases greatly with the task scale's increasing (see between16:00:00 and 20:00:00), the Waterfall Model is still more energy efficient than the traditional model, since we have introduced both coarse-grained and fine-grained energy saving strategies into the system to keep a balance between task processing and energy consumption. Thus, in the simulation, we have obtained a significant result, almost *50%* energy savings in a day.

Importantly, more *spare*-state *CPU-GPU* pairs could mean more energy consumptions (as showed in Fig. 11). In the Waterfall Model, the node choosing strategy, choosing the node awake with the minimum $\pi(n_h)$ (when two nodes with equal load balance percentage π, we always select the node with larger $C\sim G_l^h$), is to reduce the interval the node stays in *spare* level because a large π *value*, for instance $\pi(n_h)$ equals to one, means that all the *CPU-GPU* pairs in the node will finish processing their task at the same moment when the node could enter into the *sleep* status immediately.

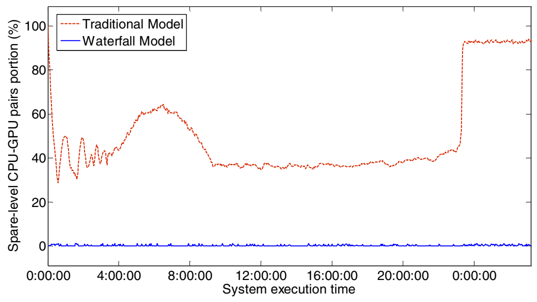

Fig. 11: The percent of *spare*-state CPU-GPU pairs curve over one day

E. *Task Average Waiting Time and Average Processing Time*

Average Waiting Time and Average Processing Time of tasks should also be listed in the metrics of the scheduling strategies. Average Waiting Time indicates the period from the releasing of the task to the beginning when the task is actually being processed by the processors, and Average Processing Time indicates the period from the beginning of the task being processed by the processors to the accomplishing of the task.

In most cases, the relation between system energy consumption and system performance is like playing the teeter-totter game. In other words, the energy savings is usually at the expense of weakening the system performance.

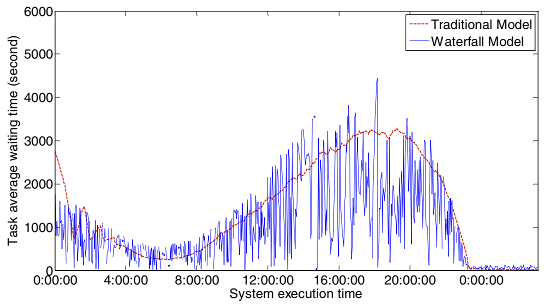

Fig. 12: Task average waiting time curve over one day

The simulation results in Fig. 12 show that task average waiting time of traditional system model is relatively stable but the situation is different for the Waterfall Model which possesses a strong volatility. This phenomenon could be explained by the DRS mechanism. Every time when the *Δ-value* of the task being scheduled is negative or equals to zero, which means that that task cannot be completed before its deadline in the current resource state, a sleeping node will be woken up. Thus, new tasks would be added to this node until the node's load balance percentage $\pi(n_h)$ is not the minimum. In this process, all tasks' waiting time will be rather small since there is no task running in the latest awoken node at the beginning, leading to the decline of the average waiting time. In summary, the DRS mechanism leads to the volatile task waiting time.

However, as we have discussed, the task processing time of the Waterfall Model will be less than that of the traditional model as the task mapping algorithm of the Waterfall Model obtains a balance of workloads between CPU and GPU. The simulation results on task average processing time are shown in Fig. 13.

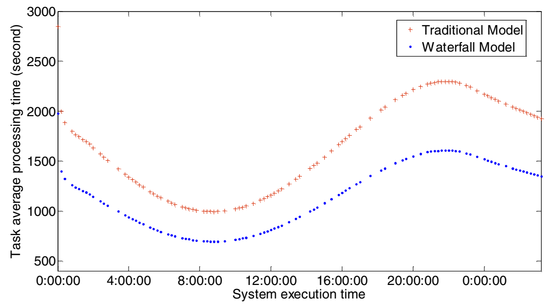

Fig. 13: Task average processing time curve over one day

VII. Conclusions

We have proposed an energy efficient task mapping algorithm with much less energy consumption and better load balance in the large-scale GPU cluster. Our major contribution is to propose a Waterfall Energy Consumption Model to guide system designers to employ different energy-saving strategies to keep the system remaining the energy levels as low as possible. These energy efficient strategies include both a coarse-grained method focusing on the computing nodes and fine-grained mechanisms focusing on the CPU-GPU pair, such as Dynamic Resource Scaling, Dynamic Voltage Scaling and β-migration for GPU sub-task. As the simulation results show, the energy savings in the low energy level compared with traditional model in the high energy level achieves nearly 50%.

Acknowledgment

This research is supported in part by National Natural Science Foundation of China ((No. 61073008, 60773148 and No.60503039), Beijing Natural Science Foundation (No. 4082016), NSF Grants CNS-0708307, IIP-0934114, OCI-0904461 (Bader), and the Center for Adaptive Supercomputing Software for Multithreaded Architectures (CASS-MT).

References

[1] N. Kappiah, V. W. Freeh, and D. K.Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs", Proceedings of the ACM/IEEE SC 2005, Seattle, USA, November 2005.

[2] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters," Proceedings of the ACM/IEEE SC 2005, Seattle, USA, November2005.

[3] C. Hsu and W. Feng, A Power-Aware Run-Time System for High-Performance Computing, Proceedings of the ACM/IEEE SC2005, Seattle, USA, November 2005.

[4] G. Diamos and S. Yalamanchili, "Harmony: An execution model and Runtime for heterogeneous many core systems," in *HPDC'08*. Boston, Massachusetts, USA: ACM, June 2008.

[5] G. Diamos and S. Yalamanchili, "Speculative Execution on Multi-GPU Systems," in *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, April, 2010.

[6] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on Heterogeneous multiprocessors with adaptive mapping," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 45–55, New York, NY, USA, December, 2009. ACM.

[7] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi, Profile-based Optimization of Power Performance by using Dynamic Voltage Scaling on a PC cluster," Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rhodes Island,Greece,April2006.

[8] S. Ghiasi,T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. In Proceedings of the 2nd Conference on Computing Frontiers, May, 2005, pp. 199–210.

[9] M. Maheswaran, T. D. Braun, and H. J. Siegel, Heterogeneous distributed computing, in Encyclopedia of Electrical and Electronics Engineering' (J. G. Webster, Ed.), Vol. 8, pp. 679-690, Wiley, NewYork, 1999.

[10] H. J. Siegel, H. G. Dietz, and J. K. Antonio, Software support for heterogeneous computing, in The Computer Science and Engineering Handbook (A. B. Tucker, Jr. ,Ed.), pp. 1886-1909, CRC Press, Boca-Raton, FL, 1997.

[11] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, in Proceedings of the 15th *EuroPar Conference* 2009, Vol. 5704 of Lecture Notes in Computer Science, pp. 863–874, Springer, Delft, The Netherlands.

[12] S. Hong, H. Kim. An analytical model for a GPU Architecture with memory-level and thread-level parallelism awareness. In *Proceeding of ACM International Symposium Computer Architecture (ISCA' 2009)*, pp. 152–163, Austin, TX, USA, June 2009.

[13] S. Hongand H. Kim. An integrated GPU power and performance model. In *Proceeding of ACM International Symposium Computer Architecture (ISCA 2010)*, pp. 280–289. June 19-23, 2010, Saint-Malo, France, USA.

[14] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In Proceeding of the 4th *International Conference on High Performance Embedded Architectures and Compilers* (*HiPEAC 2009*), pp. 19 – 33. Berlin, Heidelberg, 2009. Springer-Verlag.

[15] H. J. Siegel, S. Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. Journal of Systems Architecture. 2000;46 (8): 627-639.

[16] M. Armbrust, et al, "Above the Clouds: A Berkeley View of Cloud Computing", UCB/EECS -2009-28, Berkeley, Feb, 10, 2009.

[17] F. F. Yao, A. J. Demers, S. Shenker, A scheduling model for reduced cpu energy. In Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (1995), 374–382.

[18] C. Isciand, M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In Proceedings of the 36th *International Symposium on Microarchitecture* (*MICRO, 2003*), pp. 93-104, December 2003.

[19] T. N'Takpe, F. Suter, H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07). 2007:35-35.

[20] T. N'Takpé and F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In 12th Int. Conf. on Parallel and Distributed Systems (ICPADS), pages 3–10, July 2006.

[21] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. Parallel and Distributed Systems, IEEE Transactions on, vol. 13(3): pp. 260-274, Mar 2002.

[22] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Tureka, “Exploring weak scalability for FEM calculations on a GPU-enhanced cluster,” *Parallel Computing*, vol. 33, pp. 685-699, Nov 2007.

[23] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W. Hwu, “QP: A Heterogeneous Multi-Accelerator Cluster,” *in Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[24] V.V. Kindratenko, J.J. Enos, G. Shi, et al. GPU clusters for high-performance computing. 2009 IEEE International Conference on Cluster Computing and Workshops. 2009:1-8.

[25] R. Bryce. Power struggle. Interactive Week, December 2000. http://www.zdnet.com/intweek/, found under stories/news/0, 4164, 2666038, 00.html.

[26] D. Braun, J. Siegel, N. Beck, etc. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*. 2001;61(6):810-837.

[27] D. Fernandez- Baca, Allocating modules to processors in a distributed system, IEEE Trans. Software Engineering. 15, 1 (Nov.1989), 1427-1436.

[28] Modern Education Information Centre at Wuhan University of Science and Technology is available: http://its.wust.edu.cn/123.htm.

[29] Chinese top petaFLOPS supercomputer: Tianhe I: http://en.wikipedia.org/wiki/Tianhe-I

Wenjie Liu is a MS candidate in the Department of Computer Science and Technology at Tsinghua University, China. He is doing some research in the High Performance Computing Institute at Tsinghua University. His research interests include scheduling algorithm design on heterogeneous systems, energy efficient algorithm on multi-core/many-core systems.

Zhihui Du received the BE degree in 1992 in computer department from Tianjian University. He received the MS and PhD degrees in computer science, respectively, in 1995 and 1998, from Peking University. From 1998 to 2000, he worked at Tsinghua University as a postdoctor. From 2001 to current, he worked at Tsinghua University as an associate professor in the Department of Computer Science and Technology. His research areas include high performance computing and grid computing.

Yu Xiao is an undergraduate in the Automation School at Beijing University of Posts and Telecommunications, China. He is doing some research as an intern in the High Performance Computing Institute at Tsinghua University. His main research interests are wireless sensor networks and system simulation design.

David A. Bader is a professor in the School of Computational Science and Engineering and executive director of high performance computing at Georgia Institute of Technology. He received the PhD degree in 1996 from the University of Maryland and was awarded a US National Science Foundation (NSF) Postdoctoral Research Associateship in Experimental Computer Science. From 1998 to 2005, he served on the faculty at the University of New Mexico. He is a fellow of the IEEE and a member of the ACM. Prof. Bader has coauthored more than 100 articles in journals and conferences, and his main areas of research are in parallel algorithms, combinatorial optimization, and computational biology and genomics.

Chen Xu is a candidate for Dual Degree Bachelors in Telecommunications Engineering and Management at Beijing University of Posts and Telecommunications and University of London. She is doing some research as an intern in the High Performance Computing Institute at Tsinghua University. Her main research interests are mobile networks and scheduling algorithm.