IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

An exact algorithm for energy efficient acceleration of task trees on CPU/GPU architectures

Mark Silberstein

CCIT Report #783
January 2011

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

An exact algorithm for energy-efficient acceleration of task trees on CPU/GPU architectures.

Mark Silberstein
marks@cs.technion.ac.il
Electrical Engineering Department
Technion – Israel Institute of Technology

CCIT Report #783
January 2011

ABSTRACT

We consider the problem of *energy-efficient acceleration* of applications comprising multiple interdependent tasks forming a dependency tree, on a hypothetical CPU/GPU system where both a CPU and a GPU can be powered off when idle. Each task in the tree can be invoked on both a GPU or a CPU, but the performance may vary: some run faster on a GPU, others prefer a CPU, making the choice of the lowest-energy processor input dependent. Furthermore, greedily minimizing the energy consumption for each task is sub-optimal because of the additional energy required for the communication between the tasks executed on different processors.

We propose an efficient algorithm, which accounts for the energy consumption of a CPU and a GPU for each task, as well as for *the communication costs of data transfers between them*, and constructs an *optimal* acceleration schedule with *provably minimal* total consumed energy.

We evaluate the algorithm in the context of a real application having task dependency tree structure, and show up to 2.5-fold improvement in the expected energy consumption versus CPU only or GPU only schedule, and up to 50% improvement over the communication unaware schedule on real inputs. We also show another application of this algorithm which allows to achieve up to a 2-fold speedup in real CPU/GPU systems.

1. INTRODUCTION

Energy efficiency has become one of the central goals in contemporary hardware designs, in particular for embedded processors and SoCs, often at the expense of the peak performance. To this end, many systems already implement software-controlled dynamic power management, sometimes allowing to completely shut down idle components, quickly turning them back on when necessary. For example, NVIDIA Optimus technology [1] enables dynamic switching between power-hungry high-performance discreet GPU to the integrated low-power GPU to extend the battery life.

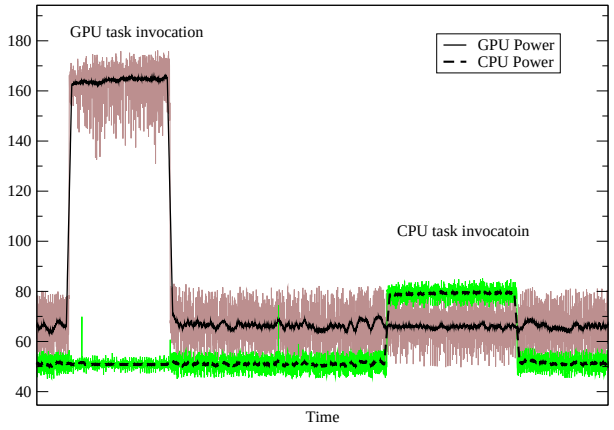

Figure 1: An example of power consumed by CPU and GPU when running the same task.

We believe that similar capabilities will be also available in the GP-GPU world, allowing for almost complete power off and zero-overhead power up of both a CPU and a GPU to enable prolonged battery life in mobile platforms.

However, without appropriate software support, energy-efficient hardware by itself will not allow energy-efficient execution. Consider an application which comprises two independent tasks. Both can be executed on a CPU or a GPU, but their performance vary: while the first runs faster on a GPU, the second does not benefit from the massive parallelism. In fact, it is easy to minimize makespan of the application by executing tasks where they perform best.

In order to achieve energy-efficient execution we also need to know the actual power dissipated by each processor. To illustrate the difference between the CPU and GPU power consumption today, Figure 1 shows the power consumed when executing the same task on all 4 cores of AMD Phenom 9500 Quad-core processor and a GTX285 NVIDIA GPU. While the GPU requires slightly less time, the CPU consumes about 50% less energy in total. Predicting GPU power consumption by statistical methods or via modeling has been investigated before [7], and can be used by the application scheduler to assign the tasks accordingly.

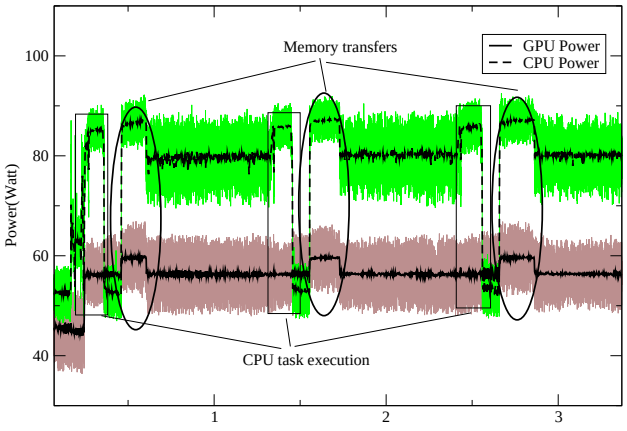

Figure 2: Power measurements of three memory transfers of 500,600 and 700 MB between CPU and GPU. A small CPU task was invoked before the transfer, followed by memory transfer and sleep() call.

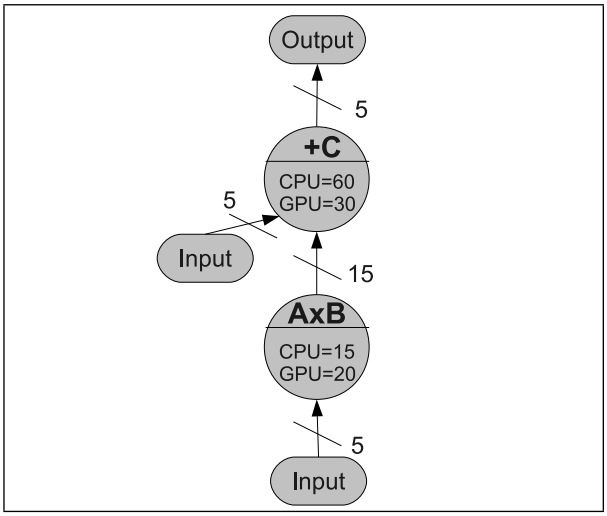

Figure 3: An illustration of the program task dependency graph for computing $A \times B + C$ of matrices A, B, C. The graph edge labels denote the time of the data transfer between a CPU and a GPU. The best schedule is to invoke both computations on a GPU despite the better CPU performance of the first kernel.

A greedy assignment of tasks to processors becomes suboptimal when there are data dependencies between the tasks. Addition energy is required to move the data between the tasks if they are executed on different processors.

We illustrate this problem using the example in Figure 3. The figure shows a *task dependency graph* of a program for computing $A \times B + C$ for three matrices A, B, C. The nodes and edges of the graph denote kernels and their data dependencies respectively. Computations are performed by traversing the graph according to the directionality of the edges. The computations of a node can be started only if all its predecessors in the graph are complete. In this example the first kernel computes $A \times B$ and the second one adds C to the result. The respective graph node labels denote the expected energy consumption in some abstract units (the lower the better) of the task on a CPU or a GPU. Edge labels denote the energy used by the data transfers given that the adjacent nodes are executed on different processors. Input data nodes represent the original input data residing in CPU memory.

Were the schedule to consider the energy consumption of each task alone, it would assign the product task to a CPU and the summation task to a GPU, consuming the total energy of 70 units. However, the best schedule requires only 65 energy units to complete, assigning both tasks to a GPU. Note that the higher energy cost of the data transfer between two tasks would increase the performance gap between the greedy and the optimal schedules.

To estimate the power consumption of data transfers in contemporary GPUs, we ran a few experiments using the same hardware as above. We transferred data from a GPU to a CPU using asynchronous memory copy calls, running the experiments with 500,600 and 700 MB (transferring from a CPU to a GPU produces exactly the same results). For validation we first ran a small CPU task to highlight the CPU power consumption due to the memory transfers. Our findings suggest that at least in this configuration the CPU is fully occupied by the memory transfers whereas the GPU power consumption increases by about 10% above the idle. We conclude that the communications indeed may incur a non-negligible power overhead.

The scheduling problem of finding an energy-efficient schedule for task dependency graphs resembles the well-known offline Directed Acyclic Graph (DAG) scheduling for parallel heterogeneous systems with communication costs. Unfortunately finding the optimal parallel schedule, even for the DAGs without undirected cycles (task dependency trees), is NP-hard [4].

We believe that energy-efficient scheduling problem is similar but different, and allows tractable optimal solution. The reason is as follows. Assume that we found an optimal assignment of tasks which minimizes the total consumed energy. Now consider two different executions: one where the tasks are invoked in parallel on the processors designated by the schedule, if such parallelism is available; and another one where they are executed on the respective processors using only one processor at a time. Provided that the energy consumption of an idle processor is negligible, both executions will result in *the same total energy consumption*. Thus, the energy-efficient scheduling permits only one processor to be busy at any given instant. Note that such schedule does not preclude parallel execution and leaves it as a secondary optimization.

We call this new problem the *energy-efficient acceleration* scheduling, and describe an efficient and easy-to-implement *optimal* algorithm for energy-efficient execution of task dependency trees. The algorithm produces the schedule which

minimizes the energy consumption of the complete application by evaluating all the tasks assignments jointly, including the overhead of the data transfer between the devices. To the best of our knowledge this is the first formulation of this problem and its optimal solution in general,and in the context of CPU/GPU architectures in particular.

The optimality of the algorithm ensures that it will produce minimal-energy acceleration schedule for any task tree. Furthermore, it is easy to build the worst case input where the algorithm would result in an arbitrary high power savings versus only-CPU, only-GPU or communication unaware greedy schedules. Consider the following task tree containing nodes A,B,C with the chain dependency (A->B->C), such that A is arbitrarily more efficient on a CPU than on a GPU, C is arbitrarily more efficient on a GPU than on a CPU, and B is marginally better on a GPU, but has arbitrarily large input to be transferred. Apparently the optimal schedule A(CPU)->B(CPU)->C(GPU) will be arbitrarily better than any of the trivial ones above.

Yet, to provide a more realistic evaluation we use a real application for inference in probabilistic graphical models. The computations comprises multiple tasks that form task dependency tree structure. We use six real inputs, each with hundreds of tasks and complex tree topology. We measure the actual execution times of each task on both a CPU and a GPU, and also the respective data transfer times for each task. We then compute the expected total energy cost of different schedules using various hardware parameters.

We found that the potential energy reduction of the optimal schedule over the best only-CPU or only-GPU schedule may reach about a factor 2.5 assuming realistic energy consumption observed in the experiments. The communication-unaware schedule may waste as much as 50% more power than the optimal one.

The algorithm can be also used to speed up the computations rather than minimize the power. Although it does not produce an optimal parallel schedule in that case, it ideally fits the programming model where a GPU is considered *a co-processor*. In such an asymmetric setup, a GPU cannot operate on its own; the CPU must dedicate some of its time to GPU management. Hence the algorithm optimizes the runtime for the case where the CPU or GPU do not concurrently execute tasks. The main advantage of this method is that it does not require changing the original sequential program flow, complementing other optimizations such as overlapping the data transfers with the GPU execution. We demonstrate 40% improvement over the greedy algorithm and up to 100% improvement over the fastest CPU-only or GPU-only execution on real-life inputs.

2. RELATED WORK

Task dependency graph scheduling minimizing the some target function for multi-processor systems has been an active research subject at least for the last 30 years. In the most general case this problem is NP-hard. Specifically, minimization of the makespan for the case of non-unit time communication delays and task execution times, there is no polynomial-time exact solution (unless $P = NP$) even when there is a fixed number of processors, all the processors have the same speed and the task graph is a tree [4]. There are some algorithms that provide good approximation [8], as close to the optimum as the maximum ratio between the task duration and the communication delays. There are numerous heuristics which do not provide any performance guarantees, but work well in many practical settings [6].

Several works attacked the problem of power-efficient scheduling in heterogeneous systems, proposing heuristic solutions and demonstrating their practical benefits. For example, the most recent work by Sanjeev [3] demonstrates successful heuristics for dual optimization of both the power and processing time.

Makespan minizaion of the DAG execution in the context of GPU-accelerated systems has been implemented in STARPU system [2], demonstrating substantial performance benefits. Yet, they do not account for the communication delays and apply dynamic, rather than static scheduling in this work.

Realization of the high variability in the GPU performance and the use of learning techniques to predict the running time of a task has been shown in [5].

Our work differs from the previous in that we present a new scheduling problem which has not been addressed before, and we provide exact, rather than approximate solution to it.

3. NOTATIONS AND PROBLEM STATEMENT

In this section we provide a more formal definition of the energy-efficient acceleration scheduling problem. To simplify the notation we will use only two processors: a CPU and a GPU, but it is easy to generalize the following to any number of processors.

Hardware model. Each processor has its own local memory which is not directly accessible to the other one. Processors communicate via message passing and explicitly transfer the data between their memories. The communication channel between the processors has finite bandwidth and is much slower than the local access, which in turn is assumed to have zero latency and infinite bandwidth.

We assume that the processors have negligible power consumption when idle, and incur no overhead when switched back on. On the other hand, data transfers are assumed to incur non-zero energy costs.

Task execution model. Each task is executed to completion without preemption. The *execution cost* of a task is the cost to complete the task on a given processor assuming that the task data is already in the processor's local memory. Similarly, the *communication cost* between two processors i and j is the cost of data transfer from the local memory of i to that of j.

Each task can be executed on all processors, but the task performance may vary substantially, and different tasks may perform favorably on different devices.

Task dependency graph. Task dependency graph $T(V, E, P, D)$ is a directed acyclic graph (DAG), where the

nodes V represent the tasks and the edges E denote the precedence constraints, or data dependencies, between the tasks. In this paper we focus on a particular case of a DAG whose underlying undirected graph is a tree and the edge are directed from the leaves to the root (in-tree).

There are two sets of weights P and D associated with the graph nodes and edges respectively. The weight of a node v is a task cost vector $P_v \in P$ with two entries for the task execution cost on a CPU and a GPU. The weight of an edge is a transfer cost matrix $D_v \in D$ with four entries for the cost of the data transfer across that edge for all the combinations of sources and destinations: GPU→CPU, CPU→GPU, GPU→GPU, CPU→CPU. We assume $D_v[CPU \rightarrow CPU] = D_v[GPU \rightarrow GPU] = 0$.

Energy efficient acceleration schedule. Consider task graph $T(V, E, P, D)$. An acceleration schedule of T onto a CPU-GPU system is a function $\mathcal{S} : V \rightarrow \{CPU, GPU\}$ which assigns each task $v \in V$ for execution on a CPU or a GPU. The cost of a schedule $\mathcal{S}$ of a graph T, is defined as

$$c(\mathcal{S}, T) = \sum_{v \in V} \left(P_v[\mathcal{S}(v)] + \sum_{w \in N_v} D_v[\mathcal{S}(w) \rightarrow \mathcal{S}(v)] \right), \quad (1)$$

where N_v is a set of the direct ancestors of v in T.

Energy-efficient acceleration schedule is a schedule with the minimum cost, provided that P and D represent the energy consumed by the processors for executing tasks and transferring the data respectively.

4. ALGORITHM FOR ENERGY-EFFICIENT SCHEDULING OF TASK TREES

The key observation which enables efficient algorithm is that the assignment of a task in one tree branch does not influence the assignment of a task in another branch. Hence, a dynamic programming approach can be used.

The algorithm presented in Figure 4 runs in two steps: cost update and backtacking.

In the cost update step the algorithm traverses the tree according to the precedence constraints. It updates the execution costs for each task v twice: first for v execution on a CPU, and second for a GPU. For each update it chooses the best processor for the direct ancestors of v while considering their own subtree costs for each assignment, and the cost of data transfer from them to v.

For every node $v \in V$, the algorithm maintains the following variables:

1. Subtree processing cost vector S_v of the subtree rooted at v, with two entries $S_v[CPU]$ and $S_v[GPU]$, each for the best processing cost of that subtree assuming v is executed on a CPU or a GPU respectively.
2. Subtree scheduling decision vector O_v, containing the task assignment $O_v^d[CPU]$ and $O_v^d[GPU]$ for every immediate ancestor (parent) d of v corresponding to $S_v[CPU]$ and $S_v[GPU]$. This variable stores d's assignment which resulted in the best total cost including the data transfer from d to v were d executed on a CPU or a GPU. It is used in the backtracking step.

Input: $T(V, E)$ - Task dependency tree, R - traversal order of T which complies with the precedence graph.
Output: Scheduling decisions A_v for all the nodes $v \in V$.

Forward traversal

while R is not empty **do**
  //get next tree node
  $v \leftarrow$ **pop**(R)
  push $v \rightarrow \hat{R}$ // maintain reverse order for backtracking
  for all $device \in CPU, GPU$ **do**
    // set the cost of v on $device$
    $S_v[device] \leftarrow P_v[device]$
    // compute the costs assuming d is executed on a CPU (GPU) and v on $device$
    for all $d \in$ child nodes of v **do**
      $CPUCOST \leftarrow S_d[CPU] + D_v[CPU \rightarrow device]$
      $GPUCOST \leftarrow S_d[GPU] + D_v[GPU \rightarrow device]$
      // choose the best schedule for d assuming v is executed on $device$
      if $CPUCOST > GPUCOST$ **then**
        $O_v^d[device] \leftarrow$ GPU
        $S_v[device] \leftarrow S_v[device] + GPUCOST$
      else
        $O_v^d[device] \leftarrow$ CPU
        $S_v[device] \leftarrow S_v[device] + CPUCOST$
      end if
    end for
  end for
end while

Backtrack

$v \leftarrow$ **pop**($\hat{R}$)
// choose the device to compute the root node
if $S_v[CPU] > S_v[GPU]$ **then**
  $A_v \leftarrow$ GPU
else
  $A_v \leftarrow$ CPU
end if
// traverse in reverse order
while $\hat{R}$ is not empty **do**
  for all $d \in$ child nodes of v **do**
    // schedule d on the device which led to the best cost for v
    $A_d \leftarrow O_v^d[A_v]$
  end for
  $v \leftarrow$ **pop**($\hat{R}$)
end while

Figure 4: Acceleration scheduling algorithm.

When this step completes, every node holds the best costs of computing its subtree for both its schedules on a CPU or a GPU. The backtracking step then traverses the tree starting from the last node being traversed and determines the assignment for all the nodes, using the optimal scheduling decision for their respective parents and generating an optimal schedule.

It is easy to see that the algorithm indeed minimizes the expression in Eq.1. The sketch of the proof is as follows. The algorithm always maintains two partial schedules for each subtree in the tree - one for the case when the root of that subtree is invoked on a CPU, and another one for the GPU invocation. The main step of the forward traversal is to unify the subtrees of a given root into a larger tree by pruning the partial schedules of the subtrees with larger total cost when the communication cost to the root is taken into account as well. Assuming that the two schedules for each subtree are indeed optimal, this step creates two optimal schedules for the root, but without deciding where to invoke

the root itself. This decision is left for the backtracking step, which iteratively chooses the best total cost for the tree root first, given the costs of its subtrees. Note that the costs of the partial schedules are computed recursively, but if the recursion is "unrolled" the result is exactly as in Eq 1.

The complexity of the algorithm is $O(|V|)$ for two processors and $O(N|V|)$ for N processors, since each node is visited twice in the forward step and once in the backtracking, and the amount of computations per node is linear in the number of processors.

5. APPLICATION TO SUM-PRODUCTS

The general sum-product computation is defined as:

$$\sum_{\mathbf{M}} \bigotimes_i f^i(\mathbf{X}^i), \quad \mathbf{M} \subseteq \bigcup_i \mathbf{X}^i, f^i \in \mathbf{F}, \tag{2}$$

where $\mathbf{F}$ is the set of all input probability functions (discrete), $\mathbf{M}$ is the set of summation variables, and *bigcup* is a tensor product.

Sum-product computations arise in a wide variety of real-life applications in artificial intelligence, statistics, image processing, and digital communications. Our motivation originated in the context of computing the likelihood of a hypothesis in large probabilistic models used in genetic analysis.

At a very high level, computing sum-products is similar to computing the chain matrix product of multi-dimensional matrices. Recall that in the chain matrix product the matrices are multiplied in a certain order, and the result of one product is used later as an input to some other product. If each product of two matrices is represented as a task, these tasks can be represented as a task dependency tree, traversed from the leaves to the root.

The same principle of building a task dependency tree is applicable to the sum-product computations. We omit all the details on how such a tree is built, and refer the reader to an in-depth overview of this subject [9].

In our implementation we allow each task in the tree to be executed both on a CPU and a GPU. The GPU implementation is not trivial and is described eslewhere [10].

One important characteristics of the tasks is that their performance on both GPU and CPU varies substantially. In [10] we show that the speedup of executing the kernel on GPU may range between factor of 50 to as low as 0.02! Thus, they present the realistic usecase for our scheduling algorithm.

6. EVALUATION

We evaluated the performance of the algorithm on six real probabilistic networks used for genetic analysis. The properties of the task dependency trees used for the evaluation are presented in Table 1.

We invoked each task on a CPU and a GPU and measured the actual execution time. We then used the peak power consumption for each processor to scale the execution time to estimate the task power consumption on that processor. This approximation is wrong in general case, since not all tasks actually bring the processor to the peak power. However for our tasks it appears to be valid, since in the measurements of different kernels the power variation did not exceed 10%.

Similarly to the task power, the energy spent on the data transfer is computed by deriving the transfer time from the hardware parameters and the data size, and scaling it by the respective power consumption.

6.1 Energy-minimal acceleration schedule

Table 2 shows the energy costs of different schedules assuming bandwidth of 1GB/s between the processors (average value measured for this input), 140 Watt due to the data transfer, 90 and 180 Watt peak consumption by a CPU and a GPU respectively, and zero idle time consumption.

We compare 4 different schedules: the optimal one produced by the algorithm, only-CPU, only-GPU and the hybrid greedy schedule, which ignores the communication overhead and assigns the device with the lowest expected energy consumption for a given task. We see that the optimal schedule results are substantially better than either CPU-only or GPU-only schedules. The optimal schedule is also tangibly better than the greedy schedule for the inputs with high communication demands, such as BN1, BN2 and BN6, but only marginally reduces the cost otherwise. Apparently, any optimization of the communication cost requires that cost to be relatively high to have any significant impact.

6.2 Applying the algorithm to running time minimization

The algorithm can be also applied to minimizing the running time of a task dependency tree execution. To improve the performance in such case, we partially relax the constraint of no concurrency between CPU and GPU execution by implementing CPU execution and GPU management in two different CPU threads. Thus, a CPU and a GPU may execute their tasks concurrently, until they run out of work because there are no ready tasks assigned to that processor by the schedule.

To make this experiment even more realistic, we did not rely on the existing task runtime measurements, but built a simple regression tree basted predictor from the profiles of the previous task invocations.

The experiments were invoked on a 4-core Intel Core 2, 2.33GHz CPU with the NVIDIA GTX285 GPU. The performance results are shown in Table 3.

We observed that the best CPU-only multi-threaded version or GPU-only version can be each up to a factor of two slower than the combined CPU-GPU execution using the schedule produced by our algorithm. Observe that the optimal acceleration schedule produced by the algorithm (the column marked in bold) usually results in more nodes being mapped to a GPU. Figure 5 shows a part of the task tree, with the diamonds denoting the nodes scheduled on a GPU by both schedules and the circles denoting those scheduled by the acceleration schedule only. Observe that the latter effectively reschedules the "islands" of CPU-scheduled nodes to a GPU.

	Tasks in tree	CPU time (msec)	GPU time (msec)	Speedup	Transfer sizes (kbytes)
BN1	390	0.01/2/196	0.15/0.3/10.5	0.04/1.4/77.08	0.016/8889/1061680
BN2	529	0.01/1/108	0.16/0.3/6.7	0.06/1.11/77.13	0.016/6371/1019220
BN3	268	0.02/23/274	0.18/1.5/32.7	0.08/6.62/62.92	0.128/999.688/33554
BN4	595	0.01/10/224	0.16/0.5/4.4	0.06/7.87/97.08	0.072/429.583/12754
BN5	1194	0.01/11/667	0.16/0.5/15.9	0.06/7.68/104.15	0.072/633.874/12754
BN6	505	0.01/3/370	0.15/0.4/14.9	0.06/1.33/115.79	0.032/76418.3/1630750

Table 1: Properties of the task dependency trees used in the experiments. Each entry is in the form minimum/average/maximum value.

	Optimal acceleration cost	Energy waste(%)		
		Hybrid-greedy	CPU-only	GPU-only
BN1	**9**	15	682	126
BN2	**11**	42	402	142
BN3	**70**	2	709	5
BN4	**39**	1	1361	28
BN5	**87**	3	1270	27
BN6	**17**	17	931	88

Table 2: Comparative performance analysis of the energy costs of different schedules.

We found, however, that for the task trees having a set of dominating complex tasks for which GPU performance substantially exceeds CPU performance and the I/O to CPU ratio is low, mapping kernels with larger input sizes is sufficient for achieving the best performance. Still, even then, the dynamic schedule that combines CPU and GPU execution remains superior to the static schedules that use only one or the other.

7. CONCLUSIONS

In this work we presented a new scheduling problem and the exact solution on a heterogeneous architectures. One clear conclusion of this work is that hybrid schedule which uses both a CPU and a GPU performs much better than only-CPU or only-GPU solution because of the substantial performance variability of GPUs.

Another important observation is that the low power consumption in the idle state can significantly simplify the algorithmic problem and enable more efficient solution.

Finally, while the benefit from using the exact algorithm may become higher with the increased amount of communications in the task graph, we see that a simple greedy algorithm may perform quite well for the case of the modest communication requirements.

8. ACKNOWLEDGMENTS

We would like to thank Dr. Naoya Maruyama for helping us with the power measurements.

9. REFERENCES

[1] Nvidia optimus technology. `http://www.nvidia.com/object/optimus_technology.html`.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing*, volume 5704, pages 863–874. Springer Berlin / Heidelberg, 2009.

[3] S. Baskiyar and R. Abdel-Kader. Energy aware dag scheduling on heterogeneous systems. *Cluster Computing*, 13:373–383, 2010.

[4] S. Fujita and M. Yamashita. Approximation algorithms for multiprocessor scheduling problem. *IEICE Transactions on Information and Systems*, 83:503–509, 2000.

[5] V. Jimenez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers*, volume 5409, pages 19–33. Springer Berlin / Heidelberg, 2009.

[6] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

[7] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical Power Consumption Analysis and Modeling for GPU-based Computing. In *Workshop on Power Aware Computing and Systems (HotPower '09)*, 2009.

[8] A. Munier. Approximation algorithms for scheduling trees with general communication delays. *Parallel Computing*, 25(1):41 – 48, 1999.

[9] P. Pakzad and V. Anantharam. A new look at the generalized distributive law. *IEEE Transactions on Information Theory*, 50(6):1132–1155, June 2004.

[10] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.

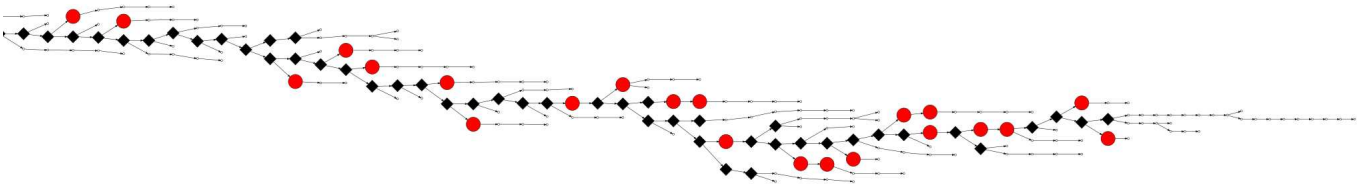

Figure 5: Part of the 268-node task tree with the hybrid-naïve and hybrid transfer-aware schedule. Nodes marked with black diamonds denote the tasks assigned to a GPU by both schedules. Red circles denote only the tasks assigned to a GPU according to the transfer-aware schedule. All unmarked nodes are assigned to a CPU.

Tasks in tree	Runtime (seconds)				Nodes mapped to GPU	
	Optimal acceleration	Hybrid-greedy	CPU-only	GPU-only	Optimal acceleration	Hybrid-greedy
390	**110**	126	213	211	25	14
529	**86**	86	119	174	41	28
268	**55**	55	408	67	86	62
595	**21**	25	174	33	139	111
1194	**35**	44	364	60	301	230
505	**126**	140	494	250	46	21

Table 3: Comparative performance analysis of the dynamic schedule for several real genetic analysis inputs. The optimal acceleration schedule is the one produced by the algorithm described here. The hybrid-greedy schedule considers only single kernel performance.