CAP: Co-Scheduling Based on Asymptotic Profiling in CPU+GPU Hybrid Systems

Zhenning Wang*
znwang@sjtu.edu.cn

Quan Chen*
chen-quan@sjtu.edu.cn

Long Zheng*†
lzheng.aizu@gmail.com

Minyi Guo*
guo-my@cs.sjtu.edu.cn

*Shanghai Jiao Tong University
Shanghai, 200240
China

†The University of Aizu
Aizu-wakamatsu, 965-8580
Japan

ABSTRACT

Hybrid systems with CPU and GPU have become the new standard in high performance computing. Workloads are split into two parts and distributed to different devices to utilize both CPU and GPU for data parallelism in hybrid systems. But it is challenging for users to manually balance workload between CPU and GPU since GPU is sensitive to the scale of the problem. Therefore, current dynamic schedulers balance workload between CPU and GPU periodically and dynamically. The periodical balance operation causes frequent synchronizations between CPU and GPU and the synchronizations often degrade the overall performance. To solve the problem, we propose a Co-Scheduling Strategy Based on Asymptotic Profiling (CAP). CAP dynamically splits one task's workload to CPU and GPU and adopts the profiling technique to predict the workload in next partition. CAP is optimized for GPU's performance characteristics to balance workload between CPU and GPU with only a few synchronizations. We examine our proof-of-concept system with four benchmarks and results show that CAP produces up to 45.1% performance improvement compared with the state-of-art co-scheduling strategy.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PMAM'2013 February 23, 2013, Shenzhen [Guangdong, China]
Copyright © 2013 ACM 978-1-4503-1908-9/13/02 ...$15.00.

Keywords

GPU, Hybrid System, Scheduling, Data-Parallelism

1. INTRODUCTION

Multicore processor is still dominating the general-purpose processor market, but many-core architectures like GPU are popular in the high performance computing area. GPU provides the ability of highly parallel processing and overwhelms multi-core processor with both parallel computing power and energy efficiency if the algorithm can be parallelized and modified to fit the architecture of the GPU. Otherwise, the performance of GPU is poor [17]. Thus, systems with multi-core processor and GPU are becoming the trend in system design to fit different situations. Although hybrid systems with CPU and GPU are widely used, programmers may not utilize them efficiently since it is hard for programmers to balance workload between CPU and GPU.

Task-parallelism and data-parallelism are two kinds of parallelism in hybrid systems. Task-parallelism assumes that many tasks are to be executed and the scheduler distributes the tasks to different devices and maintaining dependency of tasks. Data-parallelism assumes that the CPU and GPU are processing a single task whose data can be computed in parallel so the scheduler partitions the data (workload) to different devices.

Many scheduling strategies, either static or dynamic, have been proposed to balance workload between CPU and GPU for data-parallelism. In a static strategy, workloads are distributed to CPU and GPU statically. Static strategies often cause unbalanced workload because it is hard to predict the performance of GPU for a particular application without knowing the details of the runtime. The GPU is highly sensitive to scale of problem and the performance of GPU is affected by memory access pattern, depth of branches and concurrent level of the algorithm.

On the other hand, in a dynamic strategy, the partition of workload is adjusted at runtime based on the execution time of workloads on the CPU and GPU at runtime. A small portion of the workload can be sampled and the workload is split according to the sampling results. Another method uses frequently synchronizations to balance workload step by step. The former is not as accurate as it is on the CPU and causes imbalance in partition while the latter introduces

more overhead and degrades the overall performance.

In summary, load-balance is a primary factor that affects the performance of programs on hybrid systems and existing strategies do not handle it well.

We propose CAP, a novel runtime scheduler to automatically balance workload among different devices. CAP uses a dynamic strategy and it is optimized for GPU with the performance characteristics of GPU in mind. CAP needs only several synchronizations to accurately predict the performance of GPU and CPU. CAP combines the advantages of two dynamic strategies and avoiding their disadvantages. These features are handled by the runtime system without programmer's effort. Our work focus on data-parallelism scheduling for one task but the method can be extended to task-parallelism.

We implement our scheduler with CUDA and pthread as an external library to evaluate our strategy. The evaluation results show that CAP achieves up to 45.1% performance improvement compared with the state-of-art co-scheduling strategy.

We make the following contributions in this paper:

- We analyze the performance characteristics of GPU and current dynamic co-scheduling strategy.
- We propose a dynamic scheduling strategy based on profiling for distributing workloads across CPU and GPU with only a few synchronizations between CPU and GPU.
- Evaluation results show that benchmarks achieve up to 45.1% performance improvement with CAP compared to the best of current co-scheduling strategies.

The rest of the paper is arranged as follows. Section 2 presents the performance characteristics of the GPU, analysis of current strategy and the design of CAP. Section 3 describes the implementation of CAP. Section 4 shows the evaluation environment, evaluation results and discusses issues about CAP. Section 5 provides the background and related work. Section 6 draws conclusions and discusses possible future work.

2. CO-SCHEDULING BASED ON ASYMPTOTIC PROFILING

This section introduces the background knowledge about GPU, analyses current scheduling strategies and presents the design of CAP.

2.1 Performance characteristics of GPU

GPU is a kind of many-core architecture processor which is vastly different from CPU both in programming interface and performance characteristics. The current generation of dedicated GPU has GDDR5 memory and separate memory space. GDDR5 is much faster than DDR3 that the CPU has and GPU has its own memory controller to schedule accesses to the memory. GPU also has compute engines to issue program to its compute units. Thus, computing and memory access in GPU are independent of CPU and can be done in parallel. The CPU can synchronize with GPU via GPU driver of operating system but this operation takes much time and should be avoided or using techniques to hide [28].

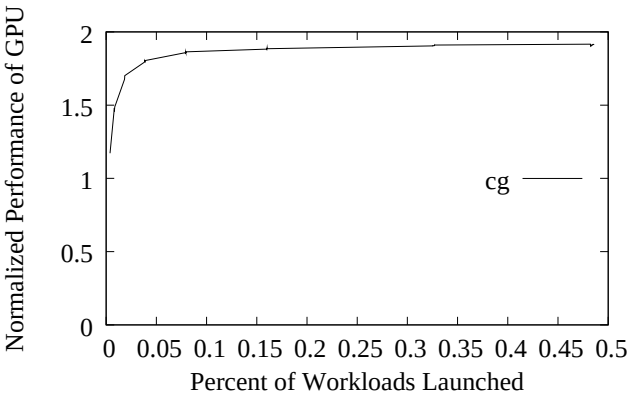

Figure 1: GPU performance curve of conjugate gradient with different workloads

GPU is famous for its number of cores and the ability to switch threads without the overhead due to large register files. If the algorithm requires little to none communication between threads or these communications have good space locality, the GPU can offer a significant speedup compared with CPU. The GPU is not good at executing programs with many branches and communications, some unoptimized algorithm may even be slower on the GPU than the CPU. The performance of GPU varies even for the same algorithm because different scales of the problem may affect the pattern of execution and memory access. So the performance of GPU is not as stable as a CPU.

Figure 1 shows the normalized performance (compared with CPU) curve of the Conjugate Gradient benchmark with different percent of workloads launched to the GPU. The evaluation environment is described in Section 4. The x-axis indicates the percent of workloads that launched to the GPU and the y-axis is the normalized performance of the GPU in the corresponding workloads. The figure shows that the performance increases as the workload increases. The performance increases rapidly in the beginning of the curve but goes stable at the end of the curve. So it is better to launch a large amount of computation to GPU one time instead of splitting it into many small parts.

In conclusion, GPU's computation, data transfer between CPU and GPU and CPU's computation are controlled by separated controller and can be done in parallel but the overhead of synchronization cannot be ignored. And the performance of GPU is related to the amount of computation that launched to it. Our scheduler takes this feature into account thus makes full use of CPU and GPU.

2.2 Current Scheduling Strategies

This subsection discusses three current scheduling strategies for CPU+GPU co-scheduling and Figure 2 shows the overview of these strategies.

2.2.1 Static Scheduling

The traditional scheduling strategy is static scheduling. It sets the ratio of performance between the CPU and GPU statically and partitions workload according to it at the beginning of the program.

This strategy does not work well for co-scheduling because the performance of GPU varies for different algorithms, scales

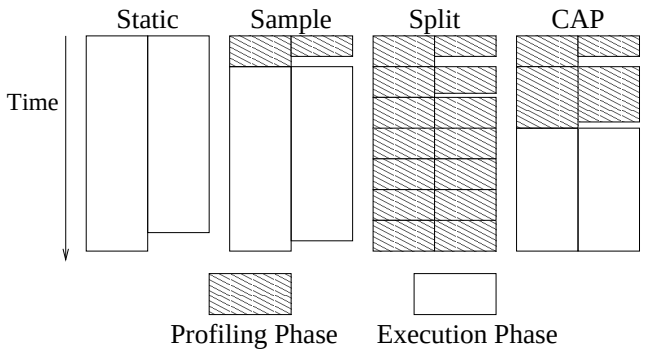

Figure 2: Strategies Overview

of the problem and implementations. Compilers can use performance models to calculate the performance ratio for a specific program and a specific GPU. The scheduler also can record the execution time of the previous execution in disk and calculate the performance ratio offline [19] but this method is not helpful on the first execution. Moreover, if the program is executed on other hardware environments, the result of offline analysis is no longer accurate. This strategy does not need synchronization and has little overhead in scheduling but the load-balance may not be good and degrade the performance of program in result.

2.2.2 *Sampling Scheduling*

Sampling scheduling strategy schedules the program in two phases to avoid the disadvantages of static scheduling strategy. In the first phase, it executes a small portion of the workload using static strategy. Then it collects the execution times of GPU and CPU to calculate the performance ratio of GPU and CPU. In the second phase, it executes the rest of the workload and partitions the workload with the ratio it calculates in the first phase.

This strategy can adjust the scheduling decision according to runtime information but the ratio it calculates may not be accurate. The performance of GPU changes when the workload changes and GPU may perform differently in the first phase and in the second phase. The size of sampling can be increased to get a more accurate result but it degrades the performance because the partition in the first phase is usually imbalance and the optimal size of sampling changes case by case. This strategy does not introduce much overhead and has a more accurate partition thus it usually gets better result compared with static scheduling.

2.2.3 *Splitting Scheduling*

Splitting Scheduling tries to get an accurate performance ratio of GPU and CPU. It splits the workload into several equally sized parts and synchronizes at the end of execution of each part. It calculates the performance ratio and partitions the next part according to the ratio in last execution.

This strategy gets best load-balance in the three strategies we discuss in this subsection. But it introduces much overhead since it needs to synchronize frequently. Splitting the workload into small parts also degrades the performance of GPU since GPU performs poorly if only a small amount of workload is launched to the GPU.

In conclusion, current strategies for data-parallelism are not good for co-scheduling of GPU and CPU. Thus we propose a novel strategy, Co-Scheduling Based on Asymptotic Profiling (CAP) to solve this problem.

2.3 Design of CAP

Figure 2 shows the overview of CAP. CAP breaks the whole execution into several phases. In the first phase, it uses static partition to execute a small portion of the workload and collects the execution times as sampling scheduling does. Instead of executing the rest of the workload and partitioning them with the ratio calculates in the first phase, CAP executes the next part which has doubled in size compared with the first part and further samples the performance of GPU and CPU. CAP continues sampling and every part has doubled in size compared with the previous part. If the variance of current partition and previous partition is smaller than the threshold, CAP will stop sampling and executes the rest of the workload.

For example, a task that has 65536 iterations which can be executed in parallel needs to be split for one CPU and one GPU co-scheduling. CAP first takes 1/128 (this parameter can be set statically) of iterations which is 512. Then CAP splits these iterations equally and assigns 256 iterations to the CPU and 256 iterations to the GPU. The CAP also transfers the needed data in these iterations to the GPU. After CPU and GPU finish their work, CAP synchronizes, transfers result from GPU and collects the execution times. Then it calculates the performance ratio by calculating the iterations of each device completes per unit time (second, for example). In the next phase, CAP compares the partition it calculated with the partition it used in the previous phase and calculates the variance of these two partitions. If the variance is small enough, CAP will think this partition is stable and reliable so it partitions the rest of the workload (65024 iterations) according it. If not, it will use the performance ratio it calculated in the previous phase to partition 1024 ($2 * 512$) iterations and doing the same job as previous phases until the next phase takes more than 1/2 of remaining iterations. If the next phase takes more than 1/2 of remaining iterations, CAP will simply execute the rest of the workload using current partition because it may not be possible for CAP to find a stable partition and CAP tries to maximize the performance of the GPU by executing a large amount of workload.

We profile the performance of GPU in an asymptotic way that CAP tries to find the stable point of the GPU's performance curve. In sampling scheduling, scheduler assumes that the performance of a processor is a constant but the performance of GPU is not a constant, so CAP tries to find a good estimate of performance by keep sampling until it is stable. In every phase, CAP adjusts the partition to get closer to the best partition. When it thinks the stable point is found, it stops profiling and partitions the remaining workload according to the best partition it can get.

Figure 1 shows that the inaccurate of sampling scheduling comes from the rapid change in the beginning of the curve and sampling scheduling samples the changing part of the curve rather than the stable part. It is risky to blindly execute a large portion of workload since the initial partition is inaccurate and imbalance. CAP increases the sampling part exponentially to find the stable point of the curve and adjusts the partition according to the execution time. Once the partition is stable, CAP can safely execute the rest of the workload and get good load-balance.

Table 1: Evaluation environment

Name	Description
CPU	Intel Xeon E5620 @ 2.4GHz
GPU	Nvidia Telsa M2090 @ 1.3GHz
CPU code compiler	GCC 4.6.3
GPU code compiler	NVCC 5.0
Operating System	Debian Wheezy (Linux Kernel 3.2)

Table 2: Benchmarks in the evaluation

Name	Description	Size
cg	Conjugate Gradient method	16Kx16K matrix
jacobi	Jacobi method	16Kx16K matrix
nbody	N-Body Simulation	16K bodies
mm	Matrix Multiplication	Two 1Kx1K matrix

CAP synchronizes only a few times since the sampling size of the workload is increased exponentially. The number of synchronizations of splitting scheduling is linear to smallest sampling size while CAP's number of synchronizations is only logarithmic to smallest sampling size. CAP also does not degrade the performance of GPU because CAP increase the amount of computation launched to GPU in each phase and it uses fixed partition only when the performance of GPU goes stable.

3. IMPLEMENTATION

We implement CAP in the form of an external library with CUDA and pthread as our proof-of-concept system. Algorithm 1 shows the algorithm we used in our implementation.

For the ease of programming, we first generate threads using pthread to handle all the devices we can use and assign devices to these threads. We initialize the devices before entering the accelerated region because it takes more time on initialization than computation on M2090 and this overhead makes scheduling inaccurate. The scheduling part is a critical section that only one thread can execute this segment of code while other threads waiting for the scheduling thread to finish its work. We use the main thread of the program as the scheduling thread. We schedule the task using the strategy described in Section 2 and the scheduler distributes the workloads to each thread then threads can do their work. If a GPU-assigned thread receives a work needs partial data when doing partial computation, it will just transfer the needed data and the overhead of data movement is included in profiling. Otherwise, it loads all data into GPU memory before the first execution. We use this technique to reduce the data transfer time between GPU and CPU. After all work is done, threads exit and synchronize at the exit point.

4. EVALUATION

This section evaluates our strategy. The evaluation environment is listed in Table 1.

We use four benchmarks, which are listed in Table 2, to measure our strategy.

Conjugate Gradient, Jacobi and Matrix Multiplication are classic matrix algorithms and N-body benchmark is a classic physics problem.

All benchmarks are implemented in three versions: CPU

Algorithm 1 Co-Scheduling Based on Asymptotic Profiling

Input: Input data set $I_1, I_2, \cdots, I_n$, number of devices d
Output: Expected output data set $O_1, O_2, \cdots, O_n$

```
1: Initialize all the devices and allocate necessary memory space on CPU and GPU.
2: Initialize the scheduler.
3: Generate d threads to handle each device. In each thread, we do the following work:
4: if This is ith thread then
5:   Assign ith device to this thread.
6:   loop
7:     Record the time of this point as the end time of previous execution.
8:     Wait until all threads reach this point.
9:     if This thread is the scheduling thread then
10:      if This is the first phase then
11:        Take a small portion of workload and partition it statically.
12:        Record the size of the portion as s
13:      else if There are remaining workload then
14:        Calculate the performance ratio of previous execution.
15:        Calculate the partition of current execution according to the performance ratio.
16:        Calculate the variance of previous and current partition.
17:        if The variance is small enough or s*2 is larger than 1/2 of remaining workload then
18:          Partition the remaining workload
19:        else
20:          s ← s * 2
21:          Partition s
22:        end if
23:      end if
24:      Distribute workloads to threads.
25:    end if
26:    Wait until scheduler gives workload to this thread.
27:    if All work has been done then
28:      Jump out of the loop.
29:    end if
30:    Record the time of this point as the start time of this execution.
31:    if The device assigned to this thread is not CPU then
32:      Copy the required input data into device memory.
33:    end if
34:    Execute the workload it receives.
35:  end loop
36: end if
37: Wait all threads to exit.
38: Aggregate the results of each device and put the final result into designated place.
```

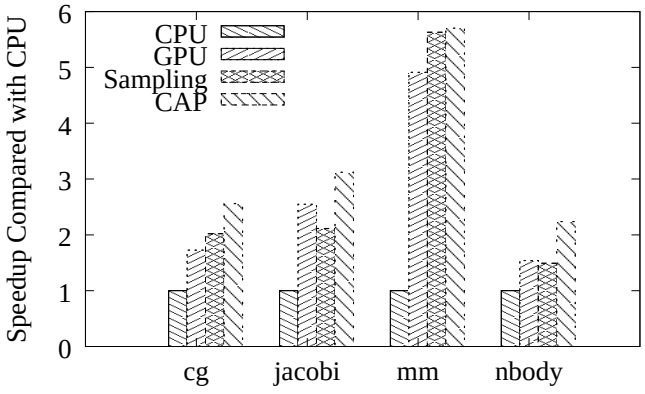

Figure 3: Speedup of CAP

Table 4: Difference in execution time between GPU and CPU of CAP (smaller is better)

Benchmark	Sampling Scheduling	CAP
cg	97.23%	1.95%
jacobi	90.43%	3.68%
mm	2.94%	1.17%
nbody	97.23%	5.83%

only (single thread), GPU only (using CUDA), and the hybrid version. The hybrid version has two scheduling strategies which are sampling scheduling and CAP. These versions are implemented in a simple way and are not fully-optimized. We do not optimize the GPU code to maximize the performance of the GPU. If the GPU code is fully optimized, The GPU can easily overwhelm CPU in performance and make co-scheduling meaningless. If the GPU is 100X faster than CPU, even if the efficiency of scheduler is 100%, co-scheduling only can get 1% performance improvement. Our purpose is to evaluate the efficiency of the scheduling strategy rather than the performance of the benchmark.

We only measure the execution time of accelerated region including kernel launching overhead, data transferring time, scheduling overhead and computation time. The initialization and data preparation time is excluded.

We use the Non-Parametric Test introduced in [9] to measure the speedup. This method calculates the speedup between two strategies/ algorithms/ configurations with 95% probability and it is more accurate than average execution time. Thus we use relative time (speedup) instead of absolute time (seconds) in our results.

We compare the performance of CAP with CPU, GPU and Sampling Scheduling implementation. The results are shown in Figure 3 and the numbers are listed in Table 3. The figure shows that CAP is significantly faster than Sampling Scheduling in cg, jacobi and nbody benchmark and achieves up to 45.1% performance improvement.

We measure the load-balance of execution by calculating the difference in execution time with the following expression:

$$\frac{|time_{CPU} - time_{GPU}|}{\min\{time_{CPU}, time_{GPU}\}}$$

Good performance of CAP comes from load-balance. As Table 4 shows, the differences in execution time between

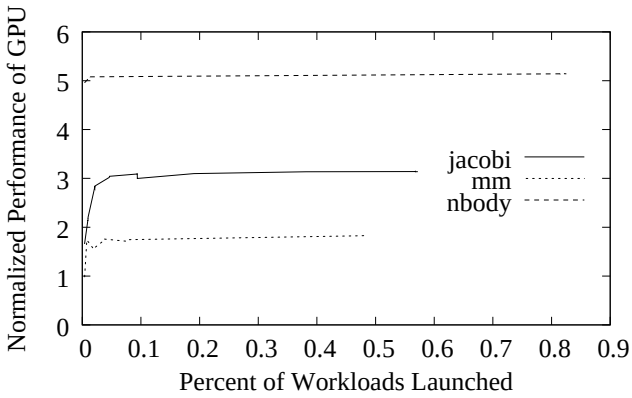

Figure 4: Performance curve of jacobi, mm and nbody with different workloads

GPU and CPU are huge for cg, jacobi and nbody benchmarks in sampling scheduling. This means that for cg, jacobi and nbody, the faster processor only needs half of the time to finish its work compared with the slower one because of inefficient co-scheduling. This reflects the fact that small sampling part is not enough to estimate the performance of the GPU. CAP balances workload well and the differences for all benchmarks are below 5%. For mm benchmark, sampling scheduling can balance workload well and the workload difference between sampling scheduling and CAP is only 1.77% so the performance improvement is also small (1.1%).

CAP adjusts the percent of profiling dynamically. If the program needs more profiling, it will profile more. Otherwise, it profiles fewer times to achieve better performance. The Table 3 shows that for cg, jacobi and nbody, CAP profiles 1/4 to 1/3 of the workload but for mm, it only profiles 1/50 of the workload. CAP only profiles when needed rather than profiling the 100% of workload like splitting scheduling. The percentage of profiling is related to the threshold of stopping profiling. If the threshold is high, the percentage will be lower, but the load-balance will be worse. If the threshold is low, the percentage will be higher but still below 50% in CAP. The optimal value of threshold has not been deeply studied and it is a possible future work. The execution phase still dominates the performance and the synchronization cost in profiling phases will not affect the performance much. The total computation sizes for each benchmark are different and mm benchmark has significantly larger computation. A small portion of the workload already reaches the stable point of the performance curve for mm. This makes the sampling scheduling quite efficient, but CAP is more efficient by ensuring the performance ratio in an extra profiling phase.

Figure 4 shows the normalized performance curve of jacobi, mm and nbody. Cg's normalized performance curve has been shown in Figure 1. The figure shows that the performance curves of cg, jacobi and nbody are similar to logistic function and mm's performance curve is a straight line. This explains why sampling scheduling performs well for mm because the performance of GPU for mm is stable across different workloads and the first profiling is already close to the best partition. But for cg, jacobi and nbody, the first profiling underestimates the performance of GPU and causes imbalance in workload partition.

Table 3: Speedup of CAP

Benchmark	CPU	GPU	Sampling Scheduling	CAP	Improvement	Average Percentage of Profiling
cg	1	1.726	2.019	2.558	26.3%	33.4%
jacobi	1	2.546	2.113	3.121	45.1%	26.0%
mm	1	4.911	5.628	5.701	1.1%	0.02%
nbody	1	1.540	1.491	2.237	37.7%	24.2%

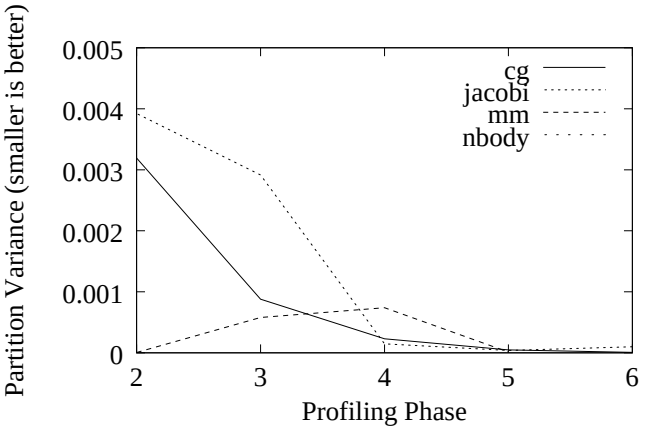

Figure 5: Partition variance of CAP in each profiling phase

Figure 5 shows the partition variances of CAP in different profiling phases. Partition variance is the variance of one profiling phase and the previous profiling phase. The number shows the changes of performance of the GPU. If the change is small, the variance will be smaller because the performance of CPU is stable and small variance means the performance of GPU is stable too. Otherwise the variance will be larger. We only show the numbers from second profiling phase because the static partition is usually inaccurate. We set the threshold to $5*10^{-5}$ but other values can be used to find a balance point that keeps load-balance while using only a few synchronizations. If the variance is smaller than the threshold, CAP will stop profiling. The figure shows that the first profiling of mm is quite accurate that CAP only needs one more profiling phase to ensure the partition is right. Others converge to the threshold quickly and five profiling is usually enough for good partition.

In conclusion, CAP significantly improves performance compared with sampling scheduling when a small portion of the workload is not enough to profile the performance of the GPU. CAP is optimized for the performance characteristics of GPU and can estimate the performance of GPU accurately without too much synchronization overhead.

5. RELATED WORK

GPU programming is becoming an important issue in the parallel programming area. Some programming language extensions like CUDA [22], Brook+ [6] and OpenCL [21] are published to utilize the hardware's raw performance. But the performances of these extensions are not portable between devices because they are close to the hardware. They can get good performance but programmers need to have a good understanding of the hardware to efficiently use both CPU and GPU.

Because GPU has its own memory, it introduces overhead when CPU transfers data and launches program on the GPU. One way to reduce it is utilizing the asynchronous property of GPU to make the computation hide the overhead of data transfer and synchronization [27]. This optimizations increase performance significantly and are not application-specific. Our work takes data movement into consideration and can benefit from the optimizations of data movement

GPU's performance characteristics have been deeply studied. [25] explored the optimizations of GPU and shows that a fully optimized GPU program is much faster than an unoptimized GPU program. [29] and [14] used performance models and analyzed the instructions that generated by NVCC to predict the performance of the GPU statically. [2] made a tool that does the analysis automatically. [15] extended [14] by integrating a power model into their performance model to get more detailed analysis. These models can be used in static analysis of the performance of the GPU but it is not useful at runtime because it introduces much overhead.

Scheduling is a well-studied problem in multi-core or distributed computing context. In the shared-memory environment, OpenMP [10] works well as a language extension for C/C++ and Fortran. WATS [8] is a workload-aware scheduling algorithm which improves the performance in asymmetric multi-core architecture. In distributed computing, Mapreduce [11] offered a simple yet powerful programming paradigm to easily write parallel programs if the algorithm does not have data dependency. Many works of scheduling in Mapreduce have been done to deal with the heterogeneous property of cloud environment. CellMR [24] is a Mapreduce framework for asymmetric Cell-based clusters. MOON [18] extended Hadoop [5] to make it work in grid computing which is highly heterogeneous.

GPU+CPU co-scheduling is also getting attention with the increasing usage of GPU in high performance computing. Several platforms are designed and implemented to combine the processing power of CPU and GPU. Mars [12], StarPU [1], Qilin [19] and Scout [20] offered different methods to map tasks to the CPU and GPU. OmpSS [7] extended OpenMP to provide co-scheduling ability. These platforms require the programmer to rewrite their code using a new programming language in the case of StarPU or Scout or using specific APIs in Mars and Qilin. Our work can be integrated into these platforms as an optional scheduling strategy.

Many works exist for the load-balance strategy in heterogeneous systems. [3], [23] and [16] focused on task-parallelism but we focus on data-parallelism scheduling. [13] proposed scheduling strategies based on profiling and used a performance model and SVM as a classifier to partition workload

into preset classes, it cannot get the best result since the best partition may not fall into the preset classes.

[26] is the state-of-art work to automatically schedule data-parallelism task between GPU and CPU based on Accelerated OpenMP [4]. We introduce the scheduling strategies used by [26] in Section 2 and have a detailed discussion about them. These strategies have their drawbacks and these drawbacks have been shown in the K-Means and Helmholtz benchmark results of [26].

6. CONCLUSION

Heterogeneous systems with CPU and GPU are becoming popular. It is important to use all the processors to solve a single task by taking advantages of data-parallelism. Existing profiling-based data-parallelism scheduling strategies do not take advantages of GPU's performance characteristics thus either it introduces too much overhead or it is not accurate enough.

We propose CAP, a novel profiling-based scheduling strategy to solve the problem by optimizing for the performance characteristics of the GPU. Our evaluation results show that comparing with the existing strategies, CAP can achieve up to 45.1% performance improvement by accurately estimating the performance of the GPU.

Although we describe our strategy in data-parallelism, this strategy can be extended to task-parallelism by recording the execution time and problem size of each task. It is more complex and it is a potential future work. Another potential future work is to explore the optimal settings of the parameters of CAP. CAP has several parameters affecting performance like how much the profiling size increases, when to stop profiling and so on. We only evaluate a set of good settings but it may not be the best settings and the best settings may be related to the application and hardware. We also use fixed static ratio for the the first partition. This is not efficient because analysis based on performance model can get a good static estimate to avoid huge imbalance in the first partition. It also gets a good start point for profiling. Compilers can give hints to the scheduler that how much work should be assigned to the GPU. Power-saving is a promising research work too. CAP does not take power consumption into account and balance workload only according to performance. The program runs faster but consumes more power. It is more energy-efficient to use the suitable processor rather than using all processors. The scheduler may schedule according to the power consumption and performance to get better performance-power ratio.

7. ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (Grant Nos. 61261160502 and 61272099), as well as the Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), China. This work is also partially supported by Japan Society for the Promotion of Science (JSPS) Research Fellowships for Young Scientists Program.

8. REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45(5):105–114, Jan. 2010.

[3] R. Bajaj and D. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *Parallel and Distributed Systems, IEEE Transactions on*, 15(2):107 – 118, feb 2004.

[4] J. Beyer, E. Stotzer, A. Hart, and B. de Supinski. Openmp for accelerators. In *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin / Heidelberg, 2011.

[5] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, Aug. 2004.

[7] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. *Euro-Par 2011 Parallel Processing*, pages 555–566, 2011.

[8] Q. Chen, Y. Chen, Z. Huang, and M. Guo. Wats: Workload-aware task scheduling in asymmetric multi-core architectures. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 249 –260, may 2012.

[9] T. Chen, Y. Chen, Q. Guo, O. Temam, Y. Wu, and W. Hu. Statistical performance comparisons of computers. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[10] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] W. Fang, B. He, Q. Luo, and N. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):608 –620, april 2011.

[13] D. Grewe and M. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[14] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[15] S. Hong and H. Kim. An integrated gpu power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289, June 2010.

[16] V. Jimenez, L. Vilanova, I. Gelado, M. Gil, G. Fursin,

and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. *High Performance Embedded Architectures and Compilers*, pages 19–33, 2009.

[17] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[18] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[19] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45 –55, dec. 2009.

[20] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Computing*, 33(10-11):648–662, 2007.

[21] A. Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, pages 11–15, 2009.

[22] C. Nvidia. Programming guide, 2008.

[23] A. Radulescu and A. van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 229 –238, 2000.

[24] M. Rafique, B. Rose, A. Butt, and D. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –12, may 2009.

[25] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[26] T. Scogland, B. Rountree, W. chun Feng, and B. de Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144 –155, may 2012.

[27] S. Venkatasubramanian, R. W. Vuduc, and n. none. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[28] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[29] Y. Zhang and J. Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382 –393, feb. 2011.