Parallel Computing 40 (2014) 107–115

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# CPU + GPU scheduling with asymptotic profiling

Zhenning Wang [a], Long Zheng [a,b], Quan Chen [a], Minyi Guo [a,*]

[a] Shanghai Jiao Tong University, Shanghai, 200240, China
[b] The University of Aizu, Aizu-wakamatsu, 965-8580, Japan

ARTICLE INFO

*Article history:*
Available online 7 December 2013

*Keywords:*
GPU
Hybrid system
Scheduling
Data-parallelism

ABSTRACT

Hybrid systems with CPU and GPU have become new standard in high performance computing. Workload can be split and distributed to CPU and GPU to utilize them for data-parallelism in hybrid systems. But it is challenging to manually split and distribute the workload between CPU and GPU since the performance of GPU is sensitive to the workload it received. Therefore, current dynamic schedulers balance workload between CPU and GPU periodically and dynamically. The periodical balance operation causes frequent synchronizations between CPU and GPU. It often degrades the overall performance because of the overhead of synchronizations. To solve the problem, we propose a Co-Scheduling Strategy Based on Asymptotic Profiling (CAP). CAP dynamically splits and distributes the workload to CPU and GPU with only a few synchronizations. It adopts the profiling technique to predict performance and partitions the workload according to the performance. It is also optimized for GPU's performance characteristics. We examine our proof-of-concept system with six benchmarks and evaluation result shows that CAP produces up to 42.7% performance improvement on average compared with the state-of-the-art co-scheduling strategies.



## 1. Introduction

Multicore processors are still dominating the general-purpose processor market, while many-core architectures like GPU are popular in the high performance computing area. GPU provides high processing power if the algorithm can be modified to fit the architecture of GPU. Hybrid systems with CPU and GPU are becoming the trend in system design. Although hybrid systems with CPU and GPU are widely used, programmers may not utilize them efficiently since it is challenging for the programmer to split and balance the workload between CPU and GPU.

Data-parallelism assumes that CPU and GPU are processing a single task whose data can be processed in parallel. The scheduler partitions the data (workload) to different devices to utilize the system. As shown in [1], CPU provides comparable performance to GPU in most cases even if the code for GPU is written by low-level language extensions like CUDA [2], Brook+ [3] and OpenCL [4]. Thus co-scheduling in data-parallelism can benefit the overall performance of the system.

Load-balance is important for data-parallelism scheduling. Many scheduling strategies, either static or dynamic, have been proposed to balance the workload between CPU and GPU. In a static strategy, the workload is split and distributed to CPU and GPU statically. Static strategies often cause unbalanced workload because it is difficult to predict the performance of GPU for a particular program without knowing the details of the runtime. GPU is highly sensitive to scale of the workload. The performance is unknown to the scheduler without previous executions.

* Corresponding author.
*E-mail addresses:* znwang@sjtu.edu.cn (Z. Wang), lzheng.aizu@gmail.com (L. Zheng), chen-quan@sjtu.edu.cn (Q. Chen), guo-my@cs.sjtu.edu.cn (M. Guo).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.11.003

On the other hand, in a dynamic strategy, the partition of the workload is adjusted based on the performance of CPU and GPU at runtime. A small portion of the workload is profiled and the workload is split according to the collected information. Another method uses frequently synchronizations to balance the workload step by step. The former causes imbalance because the profiling is not accurate enough, 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 scheduling strategy to dynamically balance the workload between CPU and GPU. CAP needs only a few synchronizations to accurately predict the performance ratio between GPU and CPU. These features are handled by the runtime system without programmer's effort. CAP combines the advantages of two dynamic strategies. Our work focus on data-parallelism scheduling but it can be extended to task-parallelism.

We implement our scheduler with CUDA and POSIX thread (pthread) as an external library to evaluate our strategy. The evaluation results show that CAP achieves up to 42.7% performance improvement on average compared with the state-of-the-art co-scheduling strategy.

We make the following contributions in this paper:

- We analyze the performance characteristics of GPU and current dynamic co-scheduling strategies.
- We propose a dynamic scheduling strategy based on profiling for splitting and distributing workload across CPU and GPU with only a few synchronizations between CPU and GPU.
- Evaluation results show that CAP achieves up to 42.7% performance improvement on average compared to the state-of-the-art co-scheduling strategies.

The rest of the paper is arranged as follows. Section 2 presents the performance characteristics of GPU and analyzes current strategies. Section 3 introduces the design of CAP and make a comparison of different strategies. Section 4 describes the implementation of CAP. Section 5 shows the evaluation environment, evaluation results and the analysis of these results. Section 6 discusses related work. Section 7 draws conclusions and discusses possible future work.

## 2. Background

This section introduces the background knowledge about GPU and analyses three current scheduling strategies [5].

GPU is a kind of many-core architecture processor which is vastly different from CPU both in programming interface and performance characteristics. CPU can synchronize with GPU via the driver but this operation has high overhead and it should be avoided.

GPU is famous for its parallel processing ability. If the algorithm requires little to none communication between threads or these communications have good space locality, 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 GPU than CPU.

The performance of GPU varies for different sizes of workload. GPU needs a large number of running threads to hide the latency of memory access. The number of running threads is related to the workload that GPU receives. If the workload is too small, GPU will not be able to reach its full performance.

Fig. 1 shows the normalized performance (compared with CPU) curve of all benchmarks in our evaluation for different sizes of workload. The evaluation environment is described in Section 5. The x-axis indicates the percent of workload that launched to GPU and the y-axis is the normalized performance of GPU for the corresponding sizes of workload.

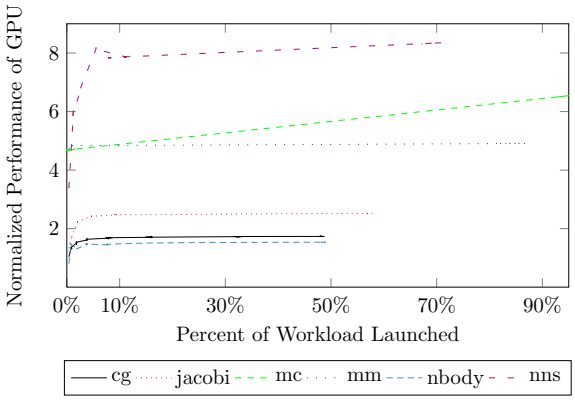


**Fig. 1.** Performance curve of different benchmarks with different sizes of workload.

The performance increases rapidly at the beginning of the curve but goes stable at the end of the curve. The synchronization overhead dominates the execution time when the workload is small. It is better to assign large workload to GPU one time instead of splitting it into many small chunks. The non-linear performance curve makes it more difficult to balance the workload because GPU may perform differently for different sizes of workload.

In conclusion, the synchronization overhead between CPU and GPU may affect performance. The performance of GPU is related to the workload it receives. Our scheduler takes these characteristics into account and fully utilizes CPU and GPU.

### 2.1. Current scheduling strategies

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

#### 2.1.1. Static scheduling

The traditional scheduling strategy is static scheduling. It sets the partition of the workload statically before execution. The partition is set by the programmer or the scheduler.

This strategy does not work well for co-scheduling on hybrid systems because the performance of GPU varies for different algorithms, sizes of workload and implementations. The scheduler can record previous execution time in disk and calculate the performance ratio offline [6]. But this method is not helpful on the first execution. The scheduler can also analyzes the code generated by the compiler to calculate the performance ratio for a specific program and a specific GPU. However these methods will fail if the program is executed on other hardware environments because the previous execution time and the offline analysis are no longer accurate.

This strategy does not need synchronization and has little overhead. The load-balance may not be good and it may degrade the overall performance.

#### 2.1.2. Quick scheduling

Quick scheduling strategy [5] executes the program in two steps to avoid the disadvantages of static scheduling strategy. In the first step, it executes a small portion of the workload with static partition. Then it collects the execution times of GPU and CPU to calculate the performance ratio. In the second step, it partitions the remaining workload with the ratio it calculates in the first step.

This strategy tries to maximize the workload assigned to GPU while keeping good load-balance. It adjusts the partition according to runtime information but the ratio it calculates may not be accurate. The performance of GPU changes when the workload changes. GPU may perform differently in the first step and in the second step. The size of profiling can be increased to get a more accurate result. But it may degrades the overall performance because the partition in the first step is usually imbalance and the optimal workload in the first step changes case by case.

This strategy does not introduce much overhead. It has a more accurate partition so it usually gets better result compared with static scheduling. However, the performance of quick scheduling is sensitive to the size of profiling and the optimal value is challenging to get without previous execution.

#### 2.1.3. Split scheduling

Split scheduling strategy [5] splits the workload into equal-sized chunks and executes them in several steps. It partitions the next part of workload according to the performance ratio in the previous step.

This strategy tries to get an accurate performance ratio of GPU and CPU. It has the best load-balance in the three strategies we discuss in this subsection. It profiles in each step and collect much more information than other strategies. But it also

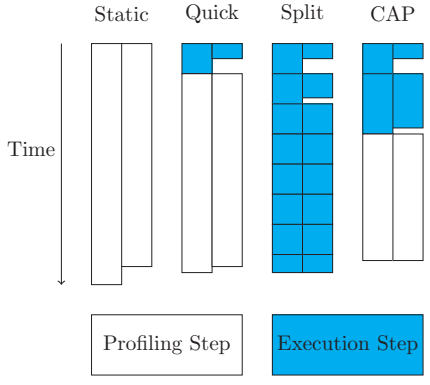


**Fig. 2.** Strategies overview.

introduces much overhead because it needs to synchronize frequently. The performance of split scheduling is sensitive to the chunk size. If the size is too small, the synchronization overhead will be too high. If the size is too large, the load-balance is affected.

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

## 3. Co-Scheduling Based on Asymptotic Profiling

This section presents the design of CAP and compare CAP with the strategies we discussed in the previous section.

### 3.1. Design of CAP

Fig. 2 shows the overview of CAP. CAP breaks the whole execution into several steps. In the first step, it executes a small portion of the workload with static partition and collects the execution time like quick scheduling. Instead of partitioning the remaining workload with the ratio it calculates in the first step, CAP executes the next step whose size of workload is doubled. In this method, CAP profiles the performance of GPU for different sizes of workload. CAP continues profiling and doubles the size of workload in each step. To find the stable point of performance, CAP calculates the variance of the current and the previous performance ratio in each step. If the variance is smaller than the threshold, or the remaining workload is smaller than the workload CAP tries to profile, CAP will stop profiling and execute the remaining workload. Algorithm 1 shows the strategy in pseudocode.

**Algorithm 1** Co-Scheduling Based on Asymptotic Profiling

```
1: if This is the first step then
2:    Take a small portion of the workload and partition with static partition.
3:    Record the size of the workload as s
4: else if There is remaining workload then
5:    Calculate the performance ratio of the previous execution.
6:    Calculate the partition of the current execution according to the performance ratio.
7:    Calculate the variance of the two partitions.
8:    if The variance is small enough or s * 2 is larger than 1/2 of remaining workload then
9:       Partition the remaining workload
10:   else
11:      s ← s * 2
12:      Partition s
13:   end if
14: end if
```

For example, CAP schedules a program that has 65536 iterations which can be executed in parallel on a CPU + GPU hybrid system. CAP first takes 1/128 (this parameter is set statically before execution) of the iterations, which is 512. Then CAP splits these iterations with static partition. We assume that the partition is 1:1 in this case. CAP assigns 256 iterations to the CPU and 256 iterations to the GPU. CAP also transfers the required data of these iterations to the GPU before launching GPU kernel. After the CPU and the GPU finish their work, CAP synchronizes, transfers the result from the GPU to the CPU and collects the execution times of both devices. Then it calculates the iterations of each device completes per unit time (second, for example).

In the second step, CAP compares the partition it calculated with the partition it used in the previous step. It calculates the variance of the two partitions. If the variance is smaller than the threshold, the performance of GPU is stable enough for CAP to make a good partition. Then It partitions the remaining workload (65,024 iterations) according to it. If not, CAP will use the performance ratio it calculated in the previous step to partition 1024 ($2 \times 512$) iterations.

CAP keeps profiling until the next step will take more than 1/2 of remaining iterations. If the next step takes more than 1/2 of remaining iterations, CAP will simply execute the remaining workload with current partition because it may not be possible for CAP to find a stable partition without degrading the performance of GPU. CAP tries to maximize the performance of GPU by assigning a large amount of workload to GPU.

CAP profiles the performance of GPU in an asymptotic way. CAP tries to find the stable point of GPU's performance curve by assigning different sizes of workload to GPU. In quick scheduling, scheduler assumes that the performance of GPU is a constant but it is not. CAP keeps profiling until the performance of GPU is stable to estimate the performance. In every step, CAP adjusts the partition to get closer to the best partition. When it finds the stable point, it stops profiling and partitions the remaining workload according to the best partition it can get.

### 3.2. Comparison of different strategies

We compare the strategies in three aspects: initial partition, partition method and workload selecting.

All the strategies we discussed use static partition in the first step. It is natural because the scheduler does not know anything about the performance in the first step. Quick, split and CAP only execute a small portion of the workload in the first step to avoid performance loss by load-imbalance.

Static strategy uses static partition all the time. Quick strategy, split strategy and CAP use performance ratio in the previous step to make the partition. Static and quick strategy assume that the performance of GPU is consistent while split and CAP assume that the performance of GPU changes. Split and CAP collect the execution time to calculate the performance ratio. Then they make the partition based on it.

Static strategy executes in only one step. The first step is also the last step. Quick strategy splits workload into one small chunk and one large chunk. It profiles the small chunk and executes the big chunk with the information collected in the small chunk. Split strategy uses several steps and the size of the workload of each step is the same. CAP uses a dynamic strategy to decide the number of steps it uses. CAP calculates the variance of partitions to check whether the performance of GPU is stable. If it is stable, CAP will stop profiling.

Fig. 1 shows that the inaccurate of quick scheduling comes from the rapid change in the beginning of the curve. Quick scheduling profiles the changing part of the curve rather than the stable part. It is risky to execute a large portion of workload blindly because the prediction may not be accurate. CAP increases the size of the workload in each step 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 remaining workload and get good load-balance.

CAP synchronizes only a few times because the size of the workload in each step is increased exponentially. The number of synchronizations in split scheduling is linear to the smallest workload while CAP's number of synchronizations is only logarithmic to the smallest workload. CAP does not degrade the performance of GPU much because CAP increase the size of the workload assigned to GPU in each step. It uses a fixed partition only after the performance of GPU goes stable.

## 4. Implementation

We implement CAP in the form of an external library with CUDA and pthread as our proof-of-concept system. The library glues CPU code and GPU code together. We write separate code for CPU and GPU in our implementation.

Asynchronous operations and extra threads can be used to handle the devices. Asynchronous operations is more complex than extra threads. So 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 the GPU we use (Nvidia Tesla M2090) and this overhead makes scheduling inaccurate.

The scheduling code is a critical section and only one thread can execute this code segment while other threads waiting for the scheduling thread to finish its work. We use the main thread of the program as the scheduling thread in our implementation. We schedule the task using the strategy described in Section 3 and the scheduler distributes the workload to each thread then threads can do their work.

If a GPU-assigned thread receives a task which only 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 will load all the data into GPU memory before the first step. We use this technique to reduce the data transfer time between GPU and CPU. After all work is done, threads synchronize at the exit point and exit.

## 5. Evaluation

This section evaluates our strategy. The evaluation environment is listed in Table 1. We use six benchmarks, which are listed in Table 2, to measure our strategy.

Conjugate Gradient, Jacobi and Matrix Multiplication are classic matrix algorithms. N-body benchmark is a classic physics problem. mc is the Monte Carlo method for european option pricing and Nearest Neighbour Search is widely used in machine learning.

**Table 1**
Evaluation environment.

| Name | Description |
|---|---|
| CPU | Intel xeon E5620 @ 2.4 GHz |
| GPU | Nvidia tesla M2090 @ 1.3 GHz |
| CPU code compiler | GCC 4.6.3 |
| GPU code compiler | NVCC 5.0 |
| Operating system | Debian wheezy (Linux 3.2) |

**Table 2**
Benchmarks in the evaluation.

| Name | Description | Size |
|---|---|---|
| cg | Conjugate Gradient method | 16 K × 16 K matrix |
| jacobi | Jacobi method | 16 K × 16 K matrix |
| mc | European option pricing | 64 M iterations |
| mm | Matrix multiplication | Two 1 K × 1 K matrix |
| nbody | N-body simulation | 16 K bodies |
| nns | Nearest neighbour search | 16 K points with 16 K queries |

All benchmarks are implemented in three versions: CPU only (single thread), GPU only (using CUDA), and the hybrid version. The hybrid version has three scheduling strategies which are quick scheduling, split scheduling and CAP. The initial workload for quick and CAP is 1/128 and the workload for split strategy is 1024. We only use simple optimization (simple tiling, reduction in shared memory, etc.) in our implementation. CPU and GPU provide comparable performance in our benchmarks. Although we do not fully optimize the code, the performance ratios are similar to the fully optimized code [1]. For benchmarks that one device overwhelms the other device, co-scheduling benefits little.

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 relative time (speedup) instead of absolute time (seconds) in our results and the speedup is the performance compared with the performance of CPU-only version. We also explore the setting of initial workload in CAP in our evaluation.

We compare the performance of CAP with quick and split strategies. The result is shown in Fig. 3 and Table 3. The figure shows that CAP is significantly faster than quick scheduling in cg, jacobi, nbody and nns benchmarks. CAP also performs much better than split scheduling in mc, mm and nns benchmarks. CAP achieves 33.4% improvement to quick and 42.7% improvement to split on average.

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

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

Good performance of CAP comes from load-balance. As Fig. 4 shows, the differences in execution time between GPU and CPU are huge for all the benchmarks except mm in quick scheduling. This means that the slower devices waits for 1/3 of the execution time in quick scheduling and the processing power is wasted. It reflects the fact that a small portion of workload is not enough to estimate the performance of GPU for different sizes of workload. The size of workload for each benchmark are different. mc and mm benchmarks have significantly larger workload and a small portion of the workload already reaches the stable point of the performance curve. For mm benchmark, quick scheduling balances workload well and the performance improvement between sampling scheduling and CAP is small (1.1%). CAP is a little more efficient by ensuring the performance ratio in an extra profiling step.

Split scheduling has better load-balance but frequent synchronizations and small workload degrades the performance of GPU. Split scheduling performs poorly in mc and mm benchmarks because the total workload is much larger than other benchmarks. The number of synchronization is also larger. In other benchmarks, split scheduling performs well.

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, nbody nns, 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 split scheduling. However, CAP is not perfect and it profiles less than needed in mc.

The execution step still dominates the performance and load-balance. Profiling steps will not affect the performance and overall load-balance much because the size of workload is small compared with execution step. CAP has even better load-balance than split for nbody and nns benchmarks because of it.

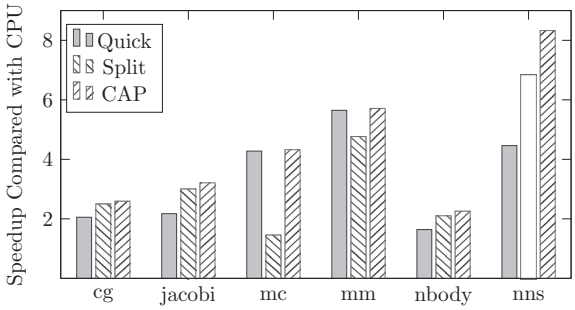


**Fig. 3.** Speedup of CAP.

**Table 3**
Improvment of CAP.

| Benchmark | Improv. to quick (%) | Improv. to split (%) | Percent of profiling (%) |
|---|---|---|---|
| cg | 26.1 | 3.6 | 37.9 |
| jacobi | 47.6 | 6.9 | 26.8 |
| mc | 1.1 | 197.3 | 3.1 |
| mm | 1.1 | 19.8 | 3.1 |
| nbody | 37.5 | 7.2 | 26.8 |
| nns | 86.7 | 21.6 | 25.0 |

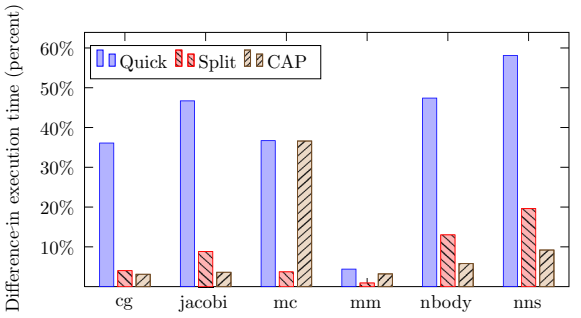


**Fig. 4.** Difference in execution time between GPU and CPU of CAP (the smaller the better).

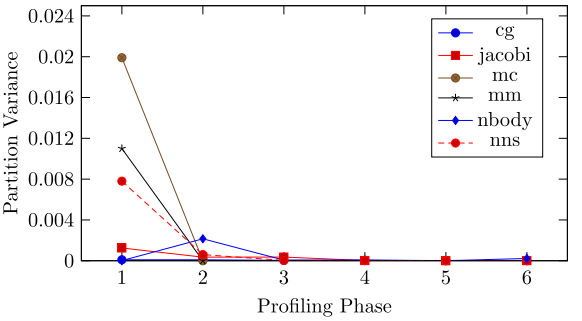


**Fig. 5.** Partition variance of CAP in each profiling step (the smaller the better).

Fig. 5 shows the partition variances of CAP in different profiling steps. Partition variance is the variance of the partition in one profiling step and the partition in the previous profiling step. The variance shows the changes of performance of GPU. If the change is small, the variance will be small. Otherwise the variance will be large. CAP tries to find the stable point by keeping profiling and calculating the variance.

The percentage of profiling is related to the threshold. 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. We set the threshold to $5 \times 10^{-5}$ but other values can be used to find a balance point that keeps load-balance with only a few synchronizations.

If the variance is smaller than the threshold, CAP will stop profiling. The figure shows that the first profiling step for mm is quite accurate that CAP only needs one more profiling step to ensure the partition is correct. Others converge to the threshold quickly and five steps is usually enough for good partition.

The initial workload of CAP can be set statically and this parameter affects the performance. We evaluate three settings of initial workload: 1/256, 1/128 and 1/64 of the total workload. They are called CAP-256, CAP-128 and CAP-64.

The size of initial workload may affect the performance of CAP. Small initial workload avoids the imbalance in the first step. However it requires more profiling steps to reach the stable point. It also degrades the performance of GPU because the workload is small.

Fig. 6 shows the speedup of different settings. The figure shows that different settings affect the performance but the changes are small. 1/128 is a good setting for different benchmarks and other setting benefits little. So CAP can adapt to different programs without tuning the parameters.

In conclusion, CAP significantly improves performance compared with current co-scheduling strategies. CAP can estimate the performance of GPU accurately without too much synchronization overhead. The performance of CAP is related, but not sensitive to the parameters.

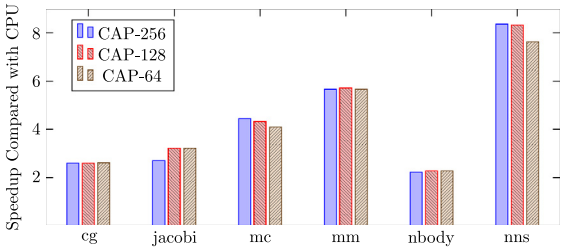


**Fig. 6.** Comparison of different setting of initial workload.

## 6. Related work

GPU programming is becoming an important issue in the parallel programming area. Some programming language extensions like CUDA [2], Brook+ [3] and OpenCL [4] are published to utilize the hardware's raw performance. But the performance 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.

GPU's performance characteristics have been deeply studied. [7] explored the optimizations of GPU and shows that a fully optimized GPU program is much faster than an unoptimized GPU program. [8,9] used performance models and analyzed the instructions that generated by NVCC to predict the performance of GPU statically. [10] made a tool that does the analysis automatically. [11] extended [9] 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 GPU but it is not useful at runtime because it introduces much overhead.

Scheduling is a well-studied problem in the shared-memory environment. OpenMP [12] works well as a language extension for C/C++ and Fortran. WATS [13] is a workload-aware scheduling algorithm which improves the performance in asymmetric multi-core architecture. In distributed computing, Mapreduce [14] 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 [15] is a Mapreduce framework for asymmetric Cell-based clusters. MOON [16] extended Hadoop [17] 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 [18] used Mapreduce as its programming paradigm. StarPU [19], Qilin [6] and Scout [20] offered different methods to map tasks to CPU and GPU. OmpSS [21] 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. Bajaj and Agrawal [22], Radulescu and van Gemund [23] and Jimenez et al. [24] focused on task-parallelism but we focus on data-parallelism scheduling. Grewe and OBoyle [25] 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.

Scogland et al. [5] is the state-of-the-art work to automatically schedule data-parallelism task between GPU and CPU based on Accelerated OpenMP [26]. We introduce the scheduling strategies used by Scogland et al. [5] 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 [5].

## 7. Conclusion and future work

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

We propose CAP, a novel scheduling strategy to solve the problem by profiling asymptotically. Our evaluation result shows that compared with the existing strategies, CAP can achieve up to 42.7% performance improvement on average by accurately estimating the performance of 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 have shown that these parameters do not affect the performance much. But optimal settings still exist for the best performance. The best settings may be related to the application and hardware.

We also use fixed static ratio for 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 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.

## 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.

## References

[1] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 451–460, http://dx.doi.org/10.1145/1815961.1816021. URL: <http://doi.acm.org/10.1145/1815961.1816021>.
[2] C. Nvidia, CUDA C programming guide 5.0, 2012.
[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: ACM SIGGRAPH 2004 Papers, SIGGRAPH '04, ACM, New York, NY, USA, 2004, pp. 777–786, http://dx.doi.org/10.1145/1186562.1015800. URL <http://doi.acm.org/10.1145/1186562.1015800>.
[4] A. Munshi, The OpenCL specification version: 1.2, 2011.
[5] T. Scogland, B. Rountree, W. chun Feng, B. de Supinski, Heterogeneous task scheduling for Accelerated OpenMP, in: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012, pp. 144–155.
[6] C.-K. Luk, S. Hong, H. Kim, Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-42, 2009, pp. 45–55.
[7] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, 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 Programing, PPoPP '08, ACM, New York, NY, USA, 2008, pp. 73–82.
[8] Y. Zhang, J. Owens, A quantitative performance analysis model for GPU architectures, in: IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), 2011, pp. 382–393.
[9] S. Hong, 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, ACM, New York, NY, USA, 2009, pp. 152–163, http://dx.doi.org/10.1145/1555754.1555775. URL <http://doi.acm.org/10.1145/1555754.1555775>.
[10] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-m.W. Hwu, An adaptive performance modeling tool for GPU architectures, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programing, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 105–114, http://dx.doi.org/10.1145/1693453.1693470. URL <http://doi.acm.org/10.1145/1693453.1693470>.
[11] S. Hong, H. Kim, An integrated GPU power and performance model, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 280–289, http://dx.doi.org/10.1145/1815961.1815998. URL <http://doi.acm.org/10.1145/1815961.1815998>.
[12] L. Dagum, R. Menon, OpenMP: an industry standard api for shared-memory programming, Comput. Sci. Eng. IEEE 5 (1) (1998) 46–55.
[13] Q. Chen, Y. Chen, Z. Huang, M. Guo, WATS: Workload-aware task scheduling in asymmetric multi-core architectures, in: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012, pp. 249–260.
[14] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113, http://dx.doi.org/10.1145/1327452.1327492. URL <http://doi.acm.org/10.1145/1327452.1327492>.
[15] M. Rafique, B. Rose, A. Butt, D. Nikolopoulos, CellMR: A framework for supporting mapreduce on asymmetric cell-based clusters, in: IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009, 2009, pp. 1–12.
[16] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, Z. Zhang, MOON: Mapreduce on opportunistic environments, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, 2010, pp. 95–106.
[17] A. Bialecki, M. Cafarella, D. Cutting, O. OMALLEY, Hadoop: a framework for running applications on large clusters built of commodity hardware, Wiki at <http://lucene.apache.org/hadoop11>
[18] B. He, W. Fang, Q. Luo, N.K. Govindaraju, T. Wang, Mars: a mapreduce framework on graphics processors, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, ACM, New York, NY, USA, 2008, pp. 260–269, http://dx.doi.org/10.1145/1454115.1454152. URL <http://doi.acm.org/10.1145/1454115.1454152>.
[19] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. Pract. Exp. 23 (2) (2011) 187–198.
[20] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, S. Cummins, Scout: a data-parallel programming language for graphics processors, Parallel Comput. 33 (10–11) (2007) 648–662.
[21] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, J. Labarta, Productive cluster programming with OmpSS, Euro-Par 2011 Parallel Processing (2011) 555–566.
[22] R. Bajaj, D. Agrawal, Improving scheduling of tasks in a heterogeneous environment, Parallel Distributed Systems, IEEE Transactions on 15 (2) (2004) 107–118.
[23] A. Radulescu, A. van Gemund, Fast and effective task scheduling in heterogeneous systems, in: Proceedings on 9th Heterogeneous Computing Workshop, (HCW 2000), 2000, pp. 229–238.
[24] V. Jimenez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, N. Navarro, Predictive runtime code scheduling for heterogeneous architectures, High Perform. Embedded Architectures Compilers (2009) 19–33.
[25] D. Grewe, M. OBoyle, A static task partitioning approach for heterogeneous systems using OpenCL, in: Compiler Construction, Springer, 2011, pp. 286–305.
[26] J. Beyer, E. Stotzer, A. Hart, B. de Supinski, OpenMP for accelerators, in: OpenMP in the Petascale Era, Vol. 6665 of Lecture Notes in Computer Science, Springer, Berlin/ Heidelberg, 2011, pp. 108–121.