Execution Time Modeling for Heterogeneous Computing System

Hyeokjun Seo[1,3], Eui-Young Chung[1,4], and Sungroh Yoon[2,5]
[1]*Dept. of Electrical and Electronic Engr., Yonsei University, Seoul, Korea*
[2]*Dept. of Electrical and Computer Engr., Seoul National University, Seoul, Korea*
[3]*jjsky7@dtl.yonsei.ac.kr*, [4]*eychung@yonsei.ac.kr*, [5]*sryoon@snu.ac.kr*

Abstract

In modern computer architecture, many computing devices are equipped with heterogeneous processing units. Accordingly we can boost system performance in many applications. We propose a methodology that models the execution time of tasks running on heterogeneous processing units, CPU and GPU, and predicts which computing unit will execute each task faster. By adopting our model, tasks can be allocated to appropriate units dynamically during run-time. Given the trend of integrating heterogeneous computing units into systems, our methodology provides an effective means to accelerate applications running on them.
Keywords: heterogeneous computing, modeling

1. Introduction

Recently, the number of computing devices that have heterogeneous computing units either at chip or board level is rapidly increasing. Not only desktop computers or workstations have powerful GPU cards inside, but even embedded systems such as smartphones and smart TVs contain both CPU and GPU these days. Given this trend, we clearly need for utilizing heterogeneous computing cores.

When we assign a task to either CPU or GPU, we choose the one that can process the task faster. In this regard, it is critical to determine which processor is the right processor for each task. In general, it is expected that GPU is more appropriate for applications with data level parallelism. However, depending on the characteristics of the given input data and the parallelization overhead, the winner may vary, making the prediction challenging. [1] For instance, a major bottleneck in GPU-based parallelization turns out to be the overhead coming from the data transfers between the host and device memories. Various approaches [2,3] have been proposed to reduce this overhead, thus expanding the horizon for GPU-based parallelization.

In this paper, we propose a methodology to construct the execution-time model of each processing unit, which plays a crucial role in selecting the right processing units for different tasks. Based on our method, we can further make the decision on which processing unit to handle them in runtime. Note that the proposed methodology is generally applicable.

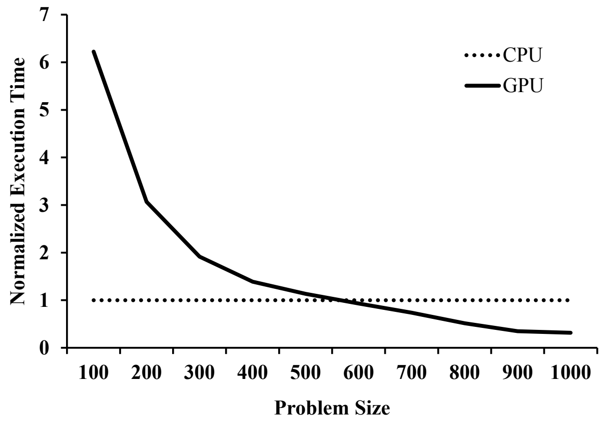


Figure 1. Normalized execution time of square-matrix multiplication on GPU to that of MCCPU. The Problem size is defined as the numbers of rows in the input matrix.

2. Motivation

Many programmers would expect that the execution of a task on GPU will run faster than on CPU if the task has sufficient data-level parallelism. However, it is normally difficult to quantify how much data-level parallelism is sufficient for GPU execution. In order to investigate this issue, we perform matrix multiplications for various sizes of square matrices on both CPU and GPU. The result is shown in Figure 1 by normalizing the execution time on GPU to that on CPU. As expected, GPU shows better performance than CPU for large size matrix multiplications. However, CPU outperforms GPU when the matrix size is under 600 x 600 in this particular example. This is because the performance gain by the parallel execution on GPU is overwhelmed by overheads, such as memory transfers between the main memory and the GPU memory. In other words, the faster processing unit varies depending on the input size (eg., matrix size), even for the same problem (eg., matrix multiplication).

This example clearly motivates the need to have a way to predict the execution time of each processing type taken to run a given workload of different sizes. To this end, we propose to use a formula to model the runtime of a task on CPU or GPU as the function of input size. Based on this model, given a program that consists of multiple tasks, we can assign each task to either CPU or GPU whichever can run it faster during run-time, achieving improved overall performance.

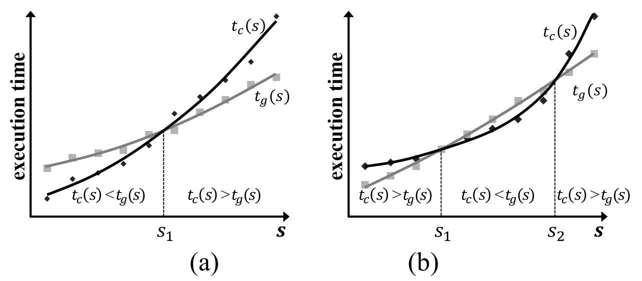
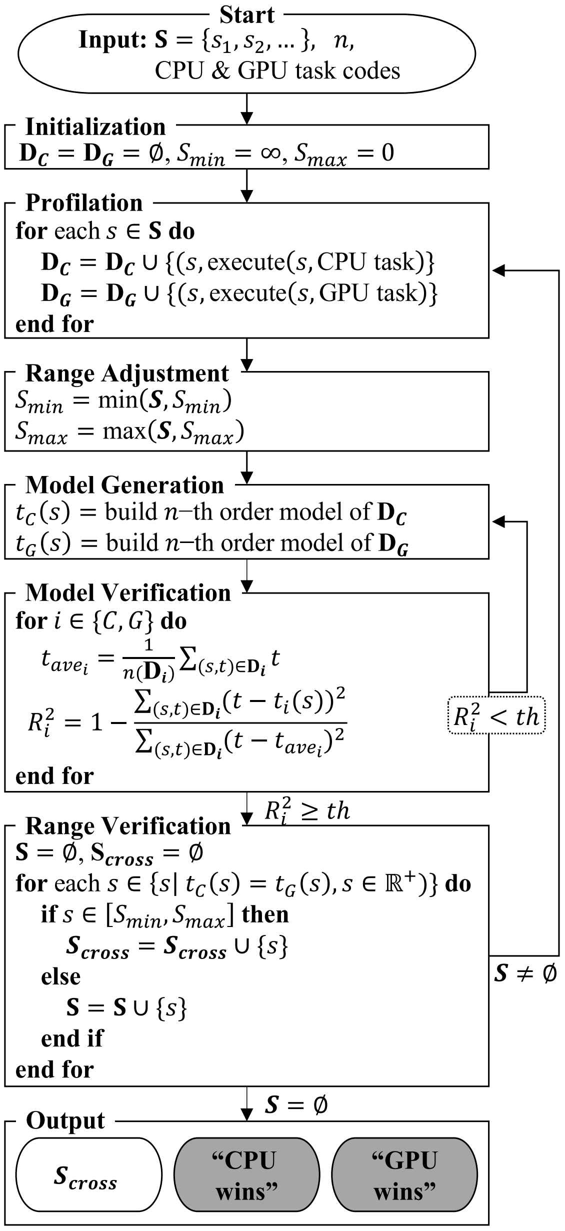


Figure 2. Cross points and intervals: (a) two intervals with one point, (b) three intervals with two points



```
Input: $\mathbf{S} = \{s_1, s_2, \dots\}$, $n$,
       CPU & GPU task codes

Initialization
$\mathbf{D}_C = \mathbf{D}_G = \emptyset, S_{min} = \infty, S_{max} = 0$

Profilation
for each $s \in \mathbf{S}$ do
    $\mathbf{D}_C = \mathbf{D}_C \cup \{(s, \text{execute}(s, \text{CPU task})\}$
    $\mathbf{D}_G = \mathbf{D}_G \cup \{(s, \text{execute}(s, \text{GPU task})\}$
end for

Range Adjustment
$S_{min} = \min(\boldsymbol{S}, S_{min})$
$S_{max} = \max(\boldsymbol{S}, S_{max})$

Model Generation
$t_C(s) = $ build $n$–th order model of $\mathbf{D}_C$
$t_G(s) = $ build $n$–th order model of $\mathbf{D}_G$

Model Verification
for $i \in \{C, G\}$ do
    $t_{ave_i} = \frac{1}{n(\mathbf{D}_i)} \sum_{(s,t) \in \mathbf{D}_i} t$
    $R_i^2 = 1 - \frac{\sum_{(s,t) \in \mathbf{D}_i} (t - t_i(s))^2}{\sum_{(s,t) \in \mathbf{D}_i} (t - t_{ave_i})^2}$
end for

Range Verification
$\mathbf{S} = \emptyset, \mathbf{S}_{cross} = \emptyset$
for each $s \in \{s \mid t_C(s) = t_G(s), s \in \mathbb{R}^+\}$ do
    if $s \in [S_{min}, S_{max}]$ then
        $\boldsymbol{S}_{cross} = \boldsymbol{S}_{cross} \cup \{s\}$
    else
        $\mathbf{S} = \mathbf{S} \cup \{s\}$
    end if
end for

Output
$S_{cross}$    "CPU wins"    "GPU wins"
```

Figure 3. Algorithm for execution time modeling

3. Execution Time Modeling

In our approach, we want to know which processing unit runs faster for a given problem size, without actually executing the task every time. To this end, we need to build computational models that predict the runtime for a problem size. To build such a model, we use the idea of curve fitting. That is, we run some test cases of different problem sizes and measure their runtime for them. Each measurement gives a data point in the size-runtime space. We then carry out curve fitting of those points using a polynomial equation to get CPU and GPU models.

By equaling the formulae of the two models, we obtain solutions indicating cross points (problem sizes for which CPU and GPU take identical time to process). Such cross points define intervals in the problem-size space, as shown in Figure 2. In each interval, either CPU or GPU runs faster than the other one.

More detailed algorithm is shown in Figure 3. The input consists of set $\boldsymbol{S}=\{s_1, s_2, \cdots\}$, parameter n, CPU and GPU task codes. Element $s_i \in \boldsymbol{S}$ represents the size of an input data set, and n corresponds to the order of polynomial to build runtime models. The minimum and maximum in $\boldsymbol{S}$ are denoted by S_{min} and S_{max}, respectively and the problem size with execution time data points for CPU and GPU are denoted by $\mathbf{D}_C$ and $\mathbf{D}_G$ respectively.

Based on the data points produced in profilation step, we constructs empirical models of running time. The independent variable of this model is the problem size s, and the response variable is the running time t. We employ the polynomial curve fitting for model construction (more sophisticated fitting methodology is also possible depending on the user's need in practice). For instance, the model for CPU runtime $t_C(s)$ is given by $t_C(s) = a_n s^n + a_{n-1} s^{n-1} + \cdots + a_0$, where n is the polynomial order, and a's are the coefficients of the polynomial determined in least-square sense from $\mathbf{D}_C$. The model for GPU runtime $t_G(s)$ is similarly constructed from $\mathbf{D}_G$.

After building the runtime models, we tests the goodness of each model in terms of R^2 or the coefficient of determination [4]. As R^2 approaches 1 more closely, the model fits the data better. If R^2 is lower than a user-specified threshold, we increases the order n and tries to fit the data again with the higher-order polynomial. This process is repeated until R^2 surpasses the threshold.

The next step is to find cross points using the models of runtime. To this end, we set $t_C(s) = t_G(s)$ and find the solution(s) of this equation numerically. In order for a solution to be valid, it should be a positive real number, and its range should be between S_{min} and S_{max}. If its range is not satisfied, then we include it in $\boldsymbol{S}$ (the set of problem sizes) and go back to profilation step to expand the search range. For some tasks, no s_{cross} would be found which means either CPU or GPU is always faster.

4. Experiment

We conduct our experiments on a Windows machine equipped with a 3.3GHz Intel i5 2500 processor, and Geforce GTX 260 graphics card. We used 7 benchmarks shown in Table 1. BF, MM are from NVIDIA CUDA SDK [5], and the others are from RODINIA benchmark suite [6].

Table 1. Benchmark Description

Name	Domain
Box Filter (BF)	Image processing
Matrix Multiplication (MM)	Mathematics
Breadth First Search (BFS)	Graph algorithm
Hot Spot (HS)	Physics simulation
Needleman-Wunsch (NW)	Bioinformatics
Path Finder (PF)	Grid traversal
Speckle Reducing Anisotropic Diffusion (SRAD)	Image processing

Table 2. Modeling analysis results

Benchmark	s_{cross}	Error	Perf. Loss
BF	4,726k	1.38%	0.22%
BFS	732k	1.78%	0.27%
HS	GPU wins	N/A	N/A
MM	601k	6.49%	2.89%
NW	4,532k	5.69%	2.17%
PF	1,181k	1.24%	0.37%
SRAD	GPU wins	N/A	N/A

We measured the accuracy of the execution time model and the result is shown in Table 2. We also measure the true values of s_{cross} for each benchmark in order to assess the estimation error defined by

$$\text{Error} = (1 - \text{estimated } s_{cross} / \text{true } s_{cross}) \times 100\ (\%)$$

For HS and SRAD, using GPU always gives the better result, and thus no s_{cross} is found. For the other benchmarks, the better processing unit varies depending on the problem size. The estimation error defined above is kept reasonable in all cases. Also, such inaccuracy in modeling should incur only marginal performance degradation, since it is hard to distinguish which is better between CPU and GPU near s_{cross}.

We conjecture that the error is mainly due to the variance of the execution time induced by erratic memory accesses. Nevertheless, such inaccuracy in modeling should incur only marginal performance degradation, since it is hard to distinguish which is better between MCCPU and GPU near s_{cross}. In practice, for the MM benchmark that shows the largest modeling error (6.49%), the performance degradation by using the estimated s_{cross} is merely 2.89%.

For the benchmarks we test, second- or third-order polynomials are sufficient for modeling, and the number of iterations for finding s_{cross} remains small. For real workloads with similar execution-time versus faster-processing-unit patterns, the modeling overhead would remain practical.

5. Conclusion

We have described a methodology for using CPU and GPU in order to reduce the running time of workloads. This method is based on modeling the execution time of CPU and GPU for tasks and make tasks dynamically allocable to either CPU or GPU, whichever is faster for each task.

The proposed execution-time modeling is simple and accurate enough for selecting the faster processing unit for tasks. According to our experiments with seven public benchmarks, the modeling error was merely 1.24 ~ 6.49 % (3.31% on average) for finding a cross point (a problem size at which the execution time of CPU and GPU are the same) with only 0.22 ~ 2.89% (1.18% on average) of the execution time less near the cross points.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation (NRF) funded by the Ministry of Education (2013R1A1A2011208) and by Samsung Electronics.

References

[1] V. W. Lee et al., “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,” in *Proc. ISCA’10*, pp. 451–460, 2010.
[2] S. Lee et al., "OpenMPC: Extended OpenMP for Efficient Programming and Tuning on GPUs", *Intl. Journal of Computational Science and Engineering*, pp 4-25, 2012.
[3] S. Grottel et al, “Optimized data transfer for time-dependent, GPU-based glyphs,” in *Proc. PacificVis’09*, pp. 65–72, 2009.
[4] N. R. Draperet al., “Applied regression analysis,” *Wiley New York*, vol. 3, 1966.
[5] Compute Unified Device Architecture (CUDA). URL: http://developer.nvidia.com/cuda
[6] S. Che et al., “Rodinia: A benchmark suite for heterogeneous computing,” in *Proc. IISWC’09*, pp. 44–54, 2009.