Vol.2011-HPC-130 No.19
2011/7/27

An execution time prediction analytical model for GPU with instruction-level and thread-level parallelism awareness

Luo Cheng[*1] and Reiji Suda[*1,*2]

Even with a powerful hardware in parallel execution, it is still difficult to improve the application performance without realizing the performance bottlenecks of parallel programs on GPU architectures. To help programmers have a better insight into the performance bottlenecks of parallel applications on GPU architectures, we propose an analytical model that estimates the execution time of massively parallel programs which take the instruction-level and thread-level parallelism into consideration. Our model contains two components: memory sub-model and computation sub-model. The memory sub-model is estimating the cost of memory instructions by considering the number of active threads and GPU memory bandwidth. Correspondingly, the computation sub-model is estimating the cost of computation instructions by considering the number of active threads and the application's arithmetic intensity. We use ocelot[1)] to analysis PTX codes to obtain several input parameters for the two sub-models such as the memory transaction number and data size. Basing on the two sub-models, the analytical model can estimates the cost of each instruction while considering instruction-level and thread-level parallelism, thereby estimating the overall execution time of an application. We compare the outcome from the model and the actual execution in GTX260; and the results show that the model can reach 90 percentage accuracy in average for the benchmarks we used.

1. Introduction

In recent years, the computing power of the Graphics Processing Units (GPUs) has been improved tremendously. For example, the latest GPU from Nvidia, GeForce GTX560 GPUs[2)] provide 1075 Gflop/s in single precision with 336 cores. Comparing to Intel's latest CPU, Intel Core i7-980 XE[3)] providing 107.6 Gflop/s, GPUs have considerably higher computing power than CPUs. Although the hardware is providing high computing performance, how to write parallel programs to make full use of GPUs is still a big challenge.

Nvidia now supports C for Compute Unified Device Architecture (CUDA)[4)] for programmers to write parallel programs on GPU. There are also some other new programming languages that help programmers with writing parallel applications for GPUs such as OpenCL[5)] and Brook+[6)]. With these programming and architectural features, programmers can quickly port their programs to a GPU based platform. However, if programmers want to have a better performance, they still need to spend much time and effort to optimize their applications. They need to have a further understanding at the various features of the low-level architecture and the associated performance bottlenecks in their applications, which will increase their burdens in writing parallel applications.

To help programmers understand the performance bottlenecks and to release their burdens in GPU architectures, we propose an execution time prediction analytical model. The model can be used to predict execution time cost without running applications on GPUs. The execution time cost of applications can be divided into two parts: computation part and memory access part. In the computation part, there exist instruction-level parallel executions within warps and between warps which can be varied due to different application types. Therefore, we introduce computing parallel degree (CPD) to describe the parallel execution for the computation instructions and to present the features of applications. In the memory access part, the latency of each memory access can be hidden by executing multiple memory accesses concurrently. We also introduce memory parallel degree (MPD) to present the maximum number of memory accesses can be executed concurrently. Using the two definitions, we analysis the low-level assembly codes of the CUDA program to estimate the overall execution time of a CUDA program.

We test our model with CUDA programming language in version 3.2. We compare the results from our model prediction and actual execution time on GPUs. The experiments show that the model can predict the execution time cost with 90 percentage accuracy in average.

The contributions of our work are as follow:

1, We propose an execution time prediction model with instruction-level and

*1 Presently with The University of Tokyo
*2 Presently with CREST,JST

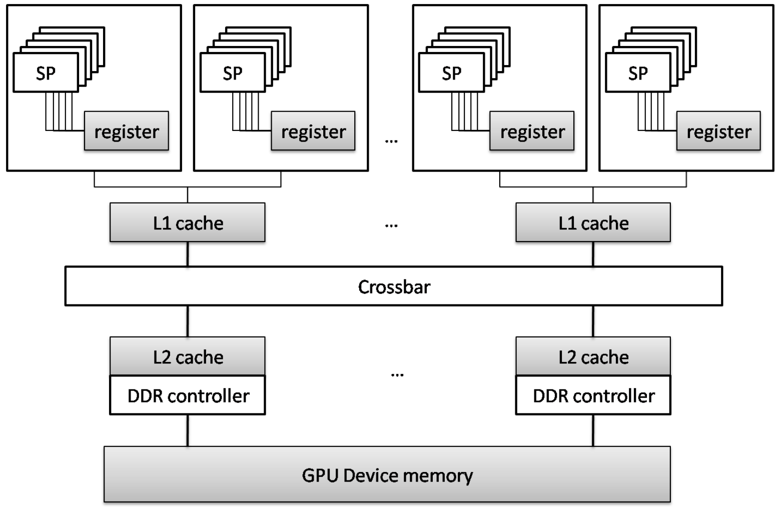

Fig. 1 GPU architecture in GeForece 200 Series

thread-level awareness.

2, We propose two definitions, CPD and MPD, to predict the execution time by considering both the feature of applications and the performance of GPU devices.

2. Background and Motivation

2.1 GPU architecture and CUDA programming model

The GPU architecture contains a scalable number of streaming multiple processors (SMs) as illustrated in **Fig. 1**. Each SM contains 8 streaming processors (SPs) in the old GPUs like GTX 260[2] while the number has reached to 32 SPs in an SM in the latest Fermi chips[7]. The GTX 260 has 24 SMs, which makes for a total of 192 processing cores.

Programmers can use CUDA API to create large number of threads to execute program on GPUs. Threads are grouped into blocks while blocks make up grid. Blocks are serially assigned for execution on each SM. One SM can have multiple concurrently running blocks we called **active blocks** in this paper. The number of blocks running on one SM is determined by the resource requirement of the blocks. Each block is divided into SIMD groups called warps which contains 32 threads in current devices and can be executed by an SM at one time. In this paper, we use a warp as a batch of 32 threads. CUDA has a very high efficient scheduling with zero cost to enable warp stalled on a memory access operation to be swapped for another warp.

The GPU has various memories at different level. In each SM, there is a set of 32-bit registers shared by the threads in one block running on the SM. There are also 16 KB of shared memory like a user-managed cache for each SM and they can be shared by all threads in one block running on the SM. The GTX 260 has 1 GB off chip global memory which can be accessed by all threads in the grid and will cost hundreds of GPU clocks for each access.

Computations to be executed on the GPU can be specified in the program as kernels. Before launching a kernel, all the data required for the computations should be transferred from the CPU (host) memory to the GPU (device) global memory. A kernel call will hand over the control to the device, and the device code will be executed on this data.

2.2 Motivation

Ideally, we thought that the more threads we use to run applications, the better performance we can have. In fact, the performance will not always be improved with the increase of the number of threads. There are lots of factors that can affect the performance such as the processor clock, the bandwidth of GPU globe memory and the application type. For computing intensive applications, increasing the number of threads will lead to a linear increase of performance. This is because the application can utilize the computing power of all processors with more threads. However, when the utilization of GPU reaches the peak computing power, the increase of threads will lead to degradation of performance because of the increase of extra overheads such as thread launch overheads and the thread synchronization overheads. Thus, the limited computing power becomes a performance bottleneck. For the memory access intensive application, there is a similar issue. The bandwidth of global memory is limited while the increase of threads will lead to the increase of bandwidth used by application. In this case, the bandwidth of global memory comes to be the performance bottleneck. Therefore, to help programmers have a better understanding with these performance bottlenecks, a performance analytical model with memory-level and

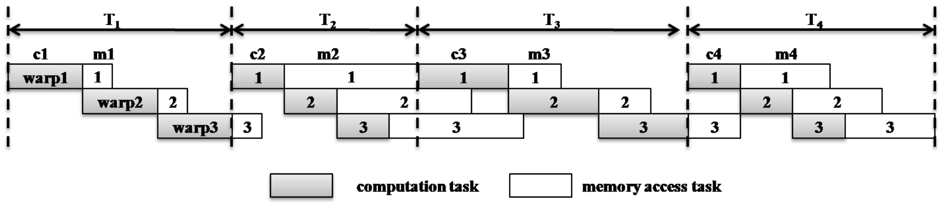

Fig. 2 Multiple warps execution in GPU architecture

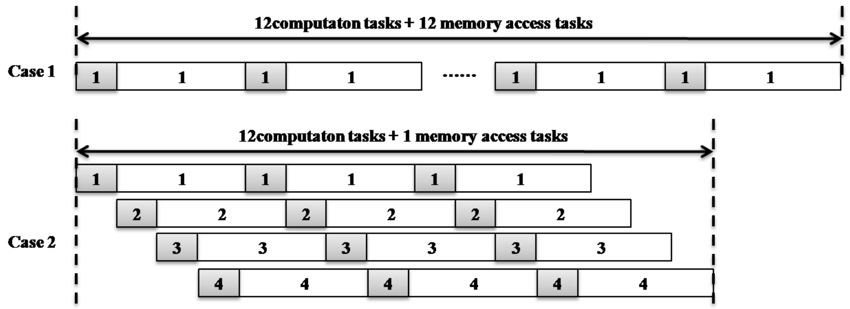
12computaton tasks + 12 memory access tasks
Case 1
12computaton tasks + 1 memory access tasks
Case 2

Fig. 3 The execution of multiple warps with MPD awareness

instruction-level awareness is required.

3. Analytical model

3.1 Assembly code analysis

Parallel Thread Execution (PTX)[8] is a pseudo-assembly language used in Nvidia's CUDA programming environment. The NVCC compiler translates the CUDA programs into PTX codes, and the GPU driver has a compiler which translates the PTX codes into machine codes to execute on GPUs. By analyzing the PTX codes, we can have a deep insight into the performance bottlenecks in GPU architecture.

With the help of ocelot[1], many details of PTX codes from CUDA program can be obtained. Ocelot is a modular dynamic compilation framework for heterogeneous system, providing various backend targets for CUDA programs and analysis modules for the PTX virtual instruction set. To get time cost of each PTX instructions, we design a set of micro benchmarks which repeat instructions 10000 times. For each PTX instruction, two kernels are designed in which there exists only one PTX instruction difference. By calculating the difference of time between two kernels, we can get the time cost for the PTX instruction. **Table 1** shows the **time cost** of part of PTX instructions thus obtained in GTX 260.

Table 1 Time cost of part of PTX instructions in GTX 260 (Unit: GPU clocks)

	int_const	int_reg	float_const	float_reg
add	22	65	22	65
sub	22	65	22	65
mul	44	136	22	65
div	728	753	748	783
neg	22	22	17	17
min	62	62	62	62
max	62	62	62	62
and	64	64	64	64
or	62	62	62	62
xor	62	62	62	62
not	22	22	22	22
ld/st	200			

3.2 Execution of multiple warps

To explain how the execution of multiple warps in each SM affects the total execution time, we use a typical scenario to illustrate as shown in **Fig. 2**.

For each warp, the PTX codes can be considered as an instruction queue of computation instructions and memory access instructions. We define a set of continuous computation instructions in one warp as a **computation task**. As the same, we also define a set of continuous memory access instructions in one warp as a **memory access task**. With these definitions, the PTX codes can be considered as a crossed permutation of computation tasks and memory access tasks. The time period from the beginning of one computation task to the beginning of the next computation task in one warp is called **calculate period**. The time cost of the ith calculate period is T_i. In GPU architecture, each SM can only execute one warp at a time. Therefore, the computation tasks between warps cannot be paralleled. However, the memory access tasks between warps can be executed in parallel. During the memory access waiting time, another active warp will be swapped to execute until the next memory access arrive.

3.3 MPD sub-model

MPD is the memory access parallel degree which is used to present the maximum warp number that can be executed in parallel. The MPD can greatly affect the total execution time. For example as shown in **Fig. 3**, when there are not enough warps to execute or the value of MPD is very low (an extreme example is 1), the execution process would be serial execution like the case 1. When there are enough warps to execute and the value of MPD is very high (the value of MPD is higher than the number of active warps), the execution process would be like case 2. With high MPD, the latency of each memory access can be hidden by executing multiple memory access concurrently.

The value of MPD can be affected by the bandwidth of GPU device, the bandwidth used by each warp, the number of active warps in each SM and the number SMs in the GPU device. For each memory access task, we introduce the following equations to calculate MPD:

$$Warp_{bwt} = (N_{thread} * D_{mem})/(N_{trans} * t_{mem}), \tag{1}$$

$$MPD = \min\{N_{act}, \lfloor GPU_{bwt}/(N_{act} * N_{sm} * Warp_{bwt})\rfloor\}. \tag{2}$$

N_{thread}: the number of threads in one warp, in this paper is 32;

D_{mem}: the data size required for each thread during each memory access;

N_{trans}: the number of memory transactions for each memory access instruction;

N_{act}: the number of active warps in one SM;

N_{sm}: the number of SMs;

t_{mem}: the latency of memory access;

$Warp_{bwt}$: the bandwidth used by one warp during one memory access;

GPU_{bwt}: the bandwidth of GPU device.

We obtain the memory access addresses of half-warp threads with ocelot[1) and calculate the number of memory transactions by following the rule of the generation of memory transaction in PTX 1.4[8)

3.4 CPD sub-model

CPD is the computation parallel degree which is used to present the parallel execution between warps and within warps in one SM. The parallel executions for the computation instructions in GPU are so complex that it is hard to give a perfect model to present. Many factors can affect the parallel execution degree such as the relationship of adjacent instructions, instruction types, computing resource requirements, the number of warps and the features of applications. To simplify the model, we only take the number of warps and the features of applications into consideration.

We use **computation instruction proportion** in the PTX codes to present the features of applications which is defined as follows. We sum up all PTX instructions time cost and only the computation instructions time cost respectively. Then the computation part time cost is divided by the total time cost of all PTX instructions to get computation instruction proportion. When the computation instruction proportion is very low, the increase of the number of warps will lead to the increase of parallel execution of computation instructions. With more warps, the number of computation instructions which can be executed in parallel will show a linear increase. Because of the low computation instruction proportion, the computing resources are always available to execute computation instructions in parallel. On this reasoning, we propose the following equation to calculate the CPD:

$$CPD1 = (c - P) * (N_{act} - b) + a. \tag{3}$$

P: the computation instruction proportion in the PTX codes;

a, b and c: the empirical parameters which get from each specific GPU device. (We write a micro benchmark to obtain these parameters. In the GTX 260, a is set to 3, b is set to 11 and c is set to 0.5).

The GPU will schedule warps to execute once there are spare computing resources. When the computation instruction proportion is high enough, the increase rate of the CPD will decrease along with a big enough warp number. Although the increase of warps leads to a linear increase of computation instructions that can be parallelized, the available computing resources become fewer and the increase rate of computation instructions that have enough computing resource to execute in parallel decreases. Therefore, when all computing resources are used up, the CPD will come to a limitation. In this situation, we use the following equations to calculate the CPD:

$$CPD2 = (n/(m-1)) * \sqrt{(m-1)^2 - (N_{act} - m)^2} + a, \tag{4}$$

$$n = d * (P - c)^2. \tag{5}$$

m: the maximum warp number can be executed in GPU;

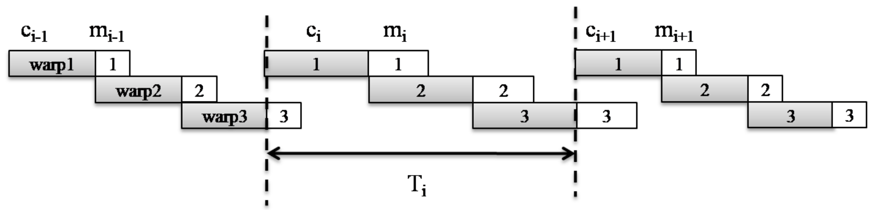

Fig. 4 Calculate period type 1

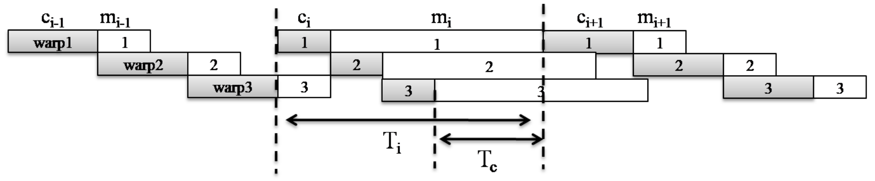

Fig. 5 Calculate period type 2

d: the empirical parameters which get from each specific GPU device.

We write a micro benchmark to obtain the parameter and d is set to 80 in GTX260. Therefore, the final value of CPD for a specific number of warps is equal to $\min\{CPD1, CPD2\}$.

3.5 Execution time prediction model

So far, we have explained the execution of multiple warps and two sub-models. In this section, we put them all together into the prediction model to predict the total time cost of the execution.

By analyzing the PTX codes, we can calculate the time cost of each calculate period and sum them up to get the total time cost. The calculate methods for each calculate period may be different due to long memory access waiting from current calculate period or previous calculate period. Ideally we hope the processors always have instructions to execute. However, long memory access tasks can let the processors wait because the following computation tasks need the results from the previous memory access tasks. Therefore, according to whether the calculate period has been affected, we classify the calculate periods into 4 types. We can analyze the relationships between time cost of computation tasks and memory access tasks in current period and previous period to select a corresponding type.

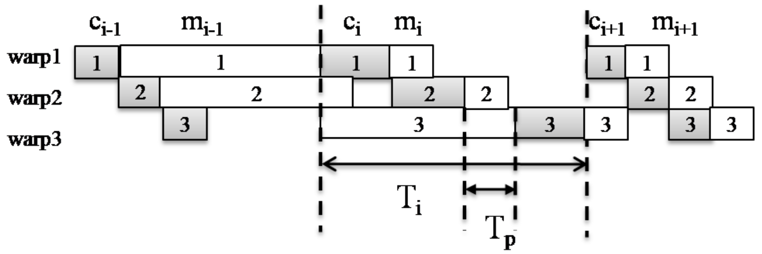

Fig. 6 Calculate period type 3

As **Fig. 4** shows, the type 1 is that there is no long memory access waiting influence from current period and previous period which means $c_{i-1} \geq m_{i-1}$ and $c_i \geq m_i$. Here, the parameters are defined as follows:

c_i: the time cost of computation task in the ith calculate period;

m_i: the time cost of memory access task in the ith calculate period;

T_i: the time cost of the ith calculate period.

Therefore, only the computation tasks make contribution to the total time cost. We sum up the time cost of all computation tasks while the parallel execution of computation parts should also be taken into consideration. We can use the following equation to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil. \quad (6)$$

In type 2, $c_{i-1} \geq m_{i-1}$ and $c_i < m_i$ as illustrated in **Fig. 5**. The long memory access tasks in current period will cause a waiting period between the last computation task in current period and the first computation task in the following period. In this case, we can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_c, \quad (7)$$

$$T_c = max\{m_i - (n-1) * c_i, 0\}. \quad (8)$$

T_c: the extra time cost caused by the long memory access tasks in current period.

In type 3, $c_{i-1} < m_{i-1}$ and $c_i \geq m_i$ as illustrated in **Fig. 6**. The long memory access tasks in previous period will cause a waiting time between the execution

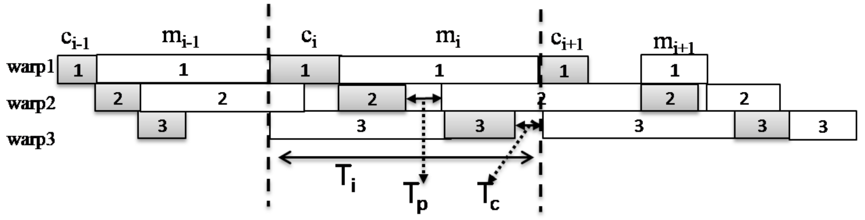

Fig. 7 Calculate period type 4

of computation tasks in current period because the results of memory access task in previous period do not arrive. Therefore, we can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_p, \tag{9}$$

$$T_p = \max\{m_{i-1} * \lfloor (N_{act}/MPD) - 1 \rfloor - (N_{act} - 1) * c_i, 0\}. \tag{10}$$

T_p: the extra time cost caused by the long memory access task in previous period.

In type 4, $c_{i-1} < m_{i-1}$ and $c_i < m_i$ as illustrated in **Fig. 7**. The long memory access tasks from current period and previous period both make extra time cost in the current calculate period. We can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_p + T_c, \tag{11}$$

$$T_p = \max\{m_{i-1} * \lfloor (N_{act}/MPD) - 1 \rfloor - (N_{act} - 1) * c_i, 0\}, \tag{12}$$

$$T_c = \max\{m_i - ((N_{act} - 1) * c_i + T_p)\}. \tag{13}$$

Finally, we can calculate the time cost for each calculate period according to the different scenarios and sum up to obtain the total time cost.

Table 2 The features of GPUs used in this work

features	GTX260
the number of SMs	24
the number of SPs	192
Graphics clock	576 MHz
Processor clock	1242 MHz
Memory size	896 MB
Memory bandwidth	111.9GB/s
Peak Gflop/s	715

4. Experiment

4.1 Experiment configuration

The GPUs used in our experiments are shown in **Table 2**. We use cudaEventRecord API in CUDA 3.2 to measure the execution time of GPU kernels. All the benchmarks are compiled with NVCC.

To test the performance of our prediction model, we use 5 different benchmarks that are mostly used in Linderman's work[9] and we port them from multi-core platform to GPU platform. Basing on the blackscholes, we make the reasult from each computing equation plus an additional parameter which requires a memory access. Therefore, blackscholes-7 is generated by increasing the number of memory access instructions to reduce the computation instruction proportion in blackscholes. The benchmarks we used to test our work are explained in **Table 3**. The computation instruction proportions of the 6 benchmarks are different from very low 26.23% to very high 86.97%. We use these in the hope of proving our model can have good prediction results in all kinds of applications.

4.2 Results

We compare the results from measured and predicted execution time on GTX260 as shown in **Fig. 8**. The number of warps per SM is varied from 1 to 32 which is the maximum number of warps that one SM can have. For each benchmark, we change the number of warps to run the kernel with the same data size. In another words, when we increase the number of warps, the data size

Table 3 The features of benchmarks

benchmarks	description	input size	com proportion
Svm[9]	Kernel from a SVM-based algorithm	512 × 768	26.23%
Matrix	Matrix multiple	256 × 256	28.2%
Linear[9]	Image filter to compute 9-pixels avg	800 × 800	45.84%
Sepia[9]	Filter for artificially aging images	800 × 800	52.97%
Blackescholes-7	Modified European option pricing	900000	76.62%
Blackescholes[9]	European option pricing	900000	86.97%

Table 4 The means of accuracy for each benchmark in GTX260

benchmarks	svm	matrix	linear	sepia	blocks-7	blocks
arithmetic means(%)	95.6	90.7	91	93.9	82.6	88.4
geometric means(%)	95.4	90.5	90.8	93.8	82.5	88.1

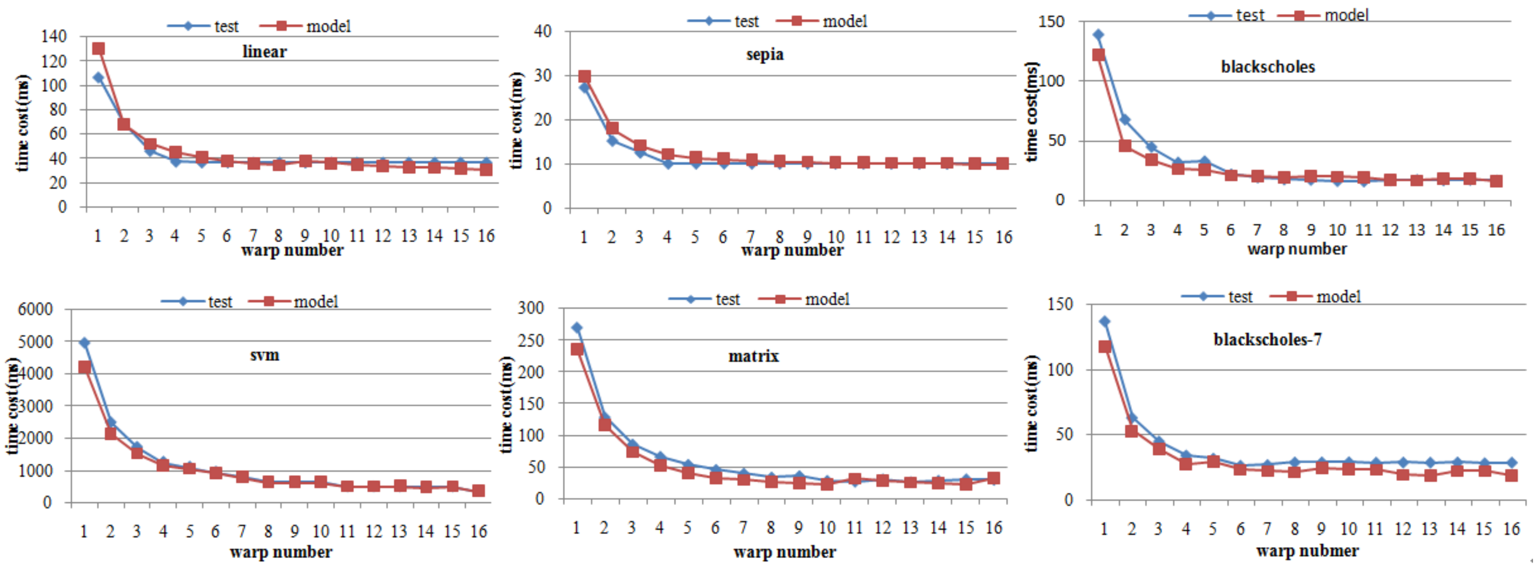

Fig. 8 The total execution time of the benchmarks on GTX260

for each warp decreases. Therefore, the execution time decreases along with the increase of the number of warps as more and more memory access latency are hidden by the parallel execution. Due to the limitation of global memroy bandwidth, the execution time will come to a limitation when the number of warps is big enough. We define the **accuracy** to present the performance of our model. We introduce the following equation to calculate accuracy:

$$P_{acc} = \min\{T_{test}, T_{model}\} / \max\{T_{test}, T_{model}\}. \tag{14}$$

P_{acc}: the accuracy for a specific number of warps;

T_{test}: the time cost for a specific number of warps from measured results;

T_{model}: the time cost for a specific number of warps from predicted results.

The accuracy of the benchmarks are shown in **Fig. 9**. The arithmetic mean and geometric mean of accuracy for each benchmark are shown in table 4.

The memory access intensive benchmarks such as svm, matrix, linear and sepia have a higher accuracy than 90% while the computing intensive benchmarks such as blockscholes and blockscholes-7 have a accuracy between 80% and 90%. This is because the CPD sub-model is an empirical model based on a set of experiment data which will reduce the accuracy of CPD. This will lead to unsteady performance of our model on computing intensive applications. In the future works, we will try to build up a better model for CPD to have a better prediction for computing intensive applications.

5. Related work

The parallel algorithms community has provided several models for the design and analysis of parallel algorithms such as Log-P[10] and QRQW[11]. These models can help programmers to find out the problem in the parallelism. However, they are mostly architecture independent which makes them to provide litter help for an insight into a specific architecture.

In Ryoo's work[12], they discuss the parameter space and present ways to reduce the size of space and get an optimized code. Schaa[13] provides an extension to Ryoo's work where they consider the multiple GPU design space optimization.

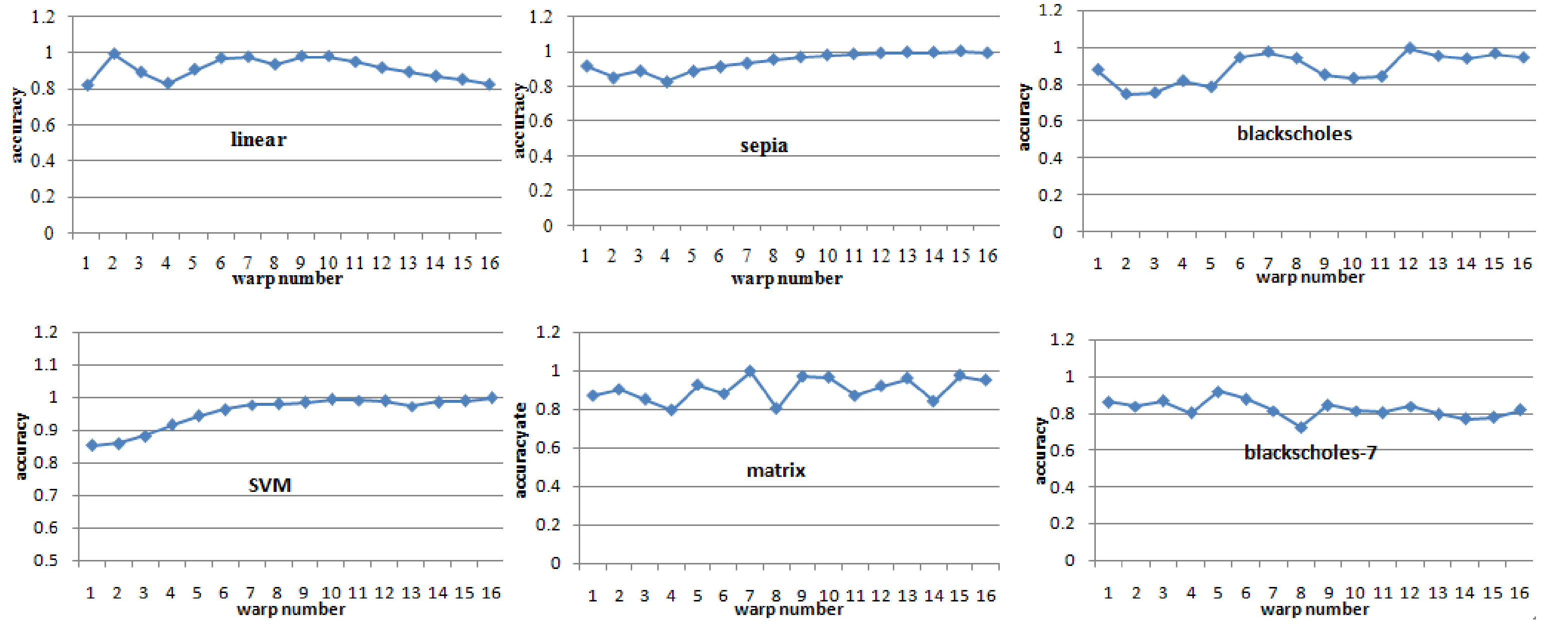

Fig. 9 The accuracy of the benchmarks on GTX260

There are also some similar works for performance prediction. In Meng's work[14], they focus on separating memory and compute requirements. However, their work is only applicable for a class of specific programs. Hong and Kim[15] propose a model that can predict the runtime of a kernel on the GPU with a set of 23 parameters. Although we have a similar approach, a few differences exist between our works. First of all, we provide an insight to the assembly codes of CUDA programs for performance analysis. We analyze the assembly codes with instruction-level parallelism awareness to obtain a high accuracy analysis results. Secondly, we take the parallelism execution of computation instructions into consideration.

6. Conclusion

This paper proposed and evaluated an execution time prediction analytical model for GPU architecture with instruction-level and memory-level awareness. We use ocelot to analysis PTX codes and obtain time cost of each PTX instructions. With the time cost of each PTX instruction as input, We use an MPD sub-model to dynamically calculate the maximum number of warps for concurrent memory access and a CPD sub-model to present the parallel execution of computation instructions between warps and within warps. With the two submodels, we predict the total execution time cost of CUDA programs with the analysis of assembly codes. Our evaluation shows that the average accuracy of our model on a set of benchmarks is 90%. We believe that this analytical model can help programmers to improve their applications.

Acknowledgments Thanks to Sunpyo Hong for applying information about GPU emulator and benchmarks. Thank to Michael Linderman for applying Merge benchmarks. Thanks to the gpuocelot groups' patient technological reply.

This work is partially supported by Core Research of Evolutional Science and Technology (CREST) project "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scienific Research of MEXT Japan.

References

1) Ocelot: a dynamic compilation framework for PTX, available from ⟨http://code.google.com/p/gpuocelot/⟩(accessed 2011/6/20).
2) NVIDIA Geforce series GTX560, GTX 260, available from ⟨http://www.nvidia.com/geforce⟩(accessed 2011/6/20).
3) Intel Core i7-980 XE processors, available from ⟨http://www.intel.com/products/processor/corei7ee/⟩ (accessed 2011/6/20).
4) NVIDIA Corporation: *CUDA Programming Guide*, Version3.2.
5) Khronos: OpenCL - the open standard for parallel programming of heterogeneous systems,available from ⟨http://www.khronos.org/opencl/⟩(accessed 2011/6/20).
6) Brook+ sc07 bof session, available from ⟨http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf⟩ (accessed 2011/6/20).
7) NVIDIA Fermi architecture, available from ⟨http://www.nvidia.com/object/fermi_architecture⟩ (accessed 2011/6/20).
8) NVIDIA Compute: *PTX: Parallel Thread Execution ISA Version 1.4*.
9) Linderman, M.D., Collins, J.D., Wang, H., and Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. *ASPLOS XIII, Proceedings of the 13th international conference on Architectural support for programming languages and operating system*, ACM, New York, NY, USA, pp.287-296(2008).
10) Culler, D., Karp, R., Pattersion, D., Sahay, A., K.E.S., Santos, E., Subramonian, R., and Voneicken, T.: LogP: Towards a Realistic Model of Parallel Computation. *PPOPP'93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA, pp.1-12(1993).
11) Gibbons, P.B., Matias, Y., and Ramachandran, V.: *The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms.* SIAM Journal on Computing, v.28 n.2, pp.733-769(1999).
12) Ryoo, S., Rodrigues, C.I., Stone, S., Bagshorkhi, S.S., Ueng, S.-Z., Stratton, J.A., and Hwu, W.W.: Program Optimization Space Pruning for a Multithreaded GPU. *CGO'08 Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ACM, New York, NY, USA, pp.195-204(2008).
13) Schaa, D., and Kaili,D.: Exploring the Multiple GPU Design Space. *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, pp.1-12(2009).
14) Meng,J., and Skadron,K.: Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. *ICS'09: Proceedings of the 23rd international conference on Supercomputing*, ACM, New York, NY, USA, pp.256-265 (2009).
15) Hong, S., and Kim, H.: An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. *ISCA'09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp.152-163 (2009).