Journal of Electronic Research and Application, 2024, Volume 8, Issue 2
http://ojs.bbwpublisher.com/index.php/JERA
ISSN Online: 2208-3510
ISSN Print: 2208-3502

Time Predictable Modeling Method for GPU Architecture with SIMT and Cache Miss Awareness

Shaojie Zhang*

School of Software Engineering University of Science and Technology of China, Hefei 230026, China

***Corresponding author:** Shaojie Zhang, zsj98@mail.ustc.edu.cn

Abstract: Graphics Processing Units (GPUs) are used to accelerate computing-intensive tasks, such as neural networks, data analysis, high-performance computing, etc. In the past decade or so, researchers have done a lot of work on GPU architecture and proposed a variety of theories and methods to study the microarchitectural characteristics of various GPUs. In this study, the GPU serves as a co-processor and works together with the CPU in an embedded real-time system to handle computationally intensive tasks. It models the architecture of the GPU and further considers it based on some excellent work. The SIMT mechanism and Cache-miss situation provide a more detailed analysis of the GPU architecture. In order to verify the GPU architecture model proposed in this article, 10 GPU kernel_task and an Nvidia GPU device were used to perform experiments. The experimental results showed that the minimum error between the kernel task execution time predicted by the GPU architecture model proposed in this article and the actual measured kernel task execution time was 3.80%, and the maximum error was 8.30%.

Keywords: Heterogeneous computing; GPU; Architecture modeling; Time predictability

Online publication: March 29, 2024

1. Introduction

Neural networks, originally developed for computer image recognition technology, have seen substantial advancements in recent years. To address the immense computational demands of neural networks, a multitude of hardware manufacturers both domestically and internationally have emerged, introducing their respective AI acceleration devices that work in concert with CPUs to handle computationally intensive tasks; Notably, Nvidia's GPU products have demonstrated exceptional performance in AI acceleration computations.

In terms of AI applications, this technology has also fostered growth in other sectors. For instance, when integrated with embedded real-time systems, it has led to groundbreaking technological achievements such as unmanned aerial vehicle (UAV) target identification and tracking, as well as advanced driver assistance systems (ADAS) in the automotive industry.

However, within the interdisciplinary field that combines neural networks, heterogeneous computing (CPU + GPU), hardware architectures, and embedded real-time systems, there exist several pressing issues that

require resolution. Chief among them is the issue of temporal predictability, which stems from the stringent time constraints inherent to real-time systems. In such systems, correctness not only depends on the logical outcome of computations but also on the timing at which these results are produced. Due to the rigorous nature of time constraints in real-time systems, temporal predictability becomes a critical performance requirement.

This paper revolves around the exploration and modeling of GPU architectures and employs established models to predict kernel function execution times. The principal contributions of this study are outlined in three key areas.

(1) This paper comprehensively considers the effects of SIMT (Single Instruction Multiple Thread) execution and cache miss scenarios on the execution time of GPU kernel functions.

(2) Building upon previous works, the paper takes an in-depth look at GPU architectures from the perspective of parallel computation, conducts a more detailed analysis, and develops a model to predict execution times.

(3) Lastly, in order to enhance the credibility of our findings, we conduct experiments using ten benchmark programs on actual GPU hardware and validate the accuracy of the proposed model through empirical results.

2. Literature review

Hong and Kim presented a relatively detailed model and two novel metrics: Memory Warp Parallelism (MWP) and Compute Warp Parallelism (CWP) [1]. Wong employed microbenchmarking to infer architectural aspects of the GPU, including the scale of cache structures [2]. The authors propose a new Fine-grained P-chase methodology to deduce various properties of the GPU's L2 Cache such as its size and cache line size, and refute the assumption that the replacement policy for the L2 cache is Least Recently Used (LRU) [3]. Building upon the Bulk Synchronous Parallel (BSP) model, Amaris and Cordeiro introduced a simple yet effective prediction model for GPU kernel execution time, with an error margin of around 10% [4]. From the perspective of GPU kernel functions, the authors propose a time-predictable model to analyze and forecast the execution times of Convolutional Neural Networks (CNNs) [5]. Adbelkhalik used microbenchmarking to infer the specific details of the Nvidia Ampere architecture and conducted fine-grained analysis at the instruction-level granularity [6]. Wang and Chu conducted experiments on 4 different GPU hardware platforms using 20 benchmark programs. They obtained a large amount of data, conducted architecture modeling, and obtained the voltage and frequency that achieved the best performance for each program on different hardware platforms [7]. Restuccia and Biondi conducted architecture modeling for DNN accelerators deployed on FPGA, mainly considering bus conflicts, memory access, OCM, and self-made hardware to analyze and predict the worst-case execution time of DNN programs more accurately [8]. Hong and Kim proposed a comprehensive architecture model to analyze the power consumption and performance of GPUs [9]. Song proposed an intuitive and accurate model for analyzing the performance and power consumption of GPU architecture [10].

3. GPU architecture and CUDA

This section mainly provides a brief background introduction to the CUDA programming model [11] and GPU architecture, on which the analysis model proposed in this article is based.

3.1. CUDA programming model

CUDA is a programming model launched by Nvidia that aims to fully utilize the performance of GPU

hardware. Programmers can use CUDA to write parallel computing applications and compile them using the NVCC compiler launched by Nvidia.

CUDA provides programmers with many mechanisms, but in general, it can be divided into three aspects: thread management, memory management, and data synchronization. The first is the thread group hierarchy for thread management, which is from top to bottom: thread grid (Grid), thread block (Block), and thread. The thread grid consists of a series of thread blocks, each of which consists of many threads. All threads within a thread block can share data and perform data synchronization through shared memory. All threads within a thread block execute concurrently on the GPU. Programmers can specify the number of threads per block and the number of thread blocks per grid.

After the kernel function is launched, multiple threads will simultaneously execute the code in the kernel function. This is the SIMT mechanism of CUDA, which means single instruction multiple threads. In a GPU clock cycle, all threads in a thread bundle execute the same instruction. Compared to SIMD, SIMT is more flexible. Through SIMT, CUDA achieves full utilization of GPU performance.

3.2. GPU architecture

The GPU architecture mainly consists of two parts: GPU core and GPU memory. The GPU core consists of L2 cache and stream multiprocessors (SM). Each SM contains 32 stream processor cores, 4 special function units, a multi-thread instruction fetch and issue unit, registers, constant memory, shared memory, and L1 cache.

SM is executed in units of warp, each warp contains 32 threads. When programmers specify the number of thread grids, thread blocks, and threads, the GPU hardware automatically divides the threads into warp. If the number is not divisible by 32, the excess threads will still be consolidated into one warp, but this will waste computational resources on SM. Shared memory is implemented as an SRAM in each SM, with very low access latency and high bandwidth. However, since the thread bundles access shared memory together, access conflicts may occur. In addition, shared memory shares a 64KB on-chip storage area with L1 cache. Programmers can configure the size of the L1 cache and shared memory through cudaFuncSetCacheConfig.

4. GPU architecture model

In the heterogeneous system, the execution process of a complete kernel function is as follows: The host-side code calls the CUDA kernel function to offload the task to the GPU device. This offloading process involves data transfer and instruction invocation. When calling the kernel function, the number of Grid/Block is specified, which determines the number of threads. Then, the GPU performs the computational tasks. After the computation is complete, the results are transferred from the GPU global memory to the CPU global memory through the PCIe interface.

The complete model parameters and their definitions are summarized in **Table 1**.

Table 1. Model parameter definitions

Parameters	Definitions	How to achieve
D^S	Data size of kernel	Kernel setup
N^B	Total number of blocks	Kernel setup
D^W_{pb}	Number of warps per block	Kernel setup
N^W	Total number of warps of kernel	Kernel setup
N^{GT}	Number of Global load/store transactions per warp	Nvidia profiler

Table 1. (Continue)

Parameters	Definitions	How to achieve
N^C	Number of computation instructions of kernel	Nvidia profiler
N^{SM}	Number of active warps	Nvidia profiler
N_{act}^W	Number of concurrent warps per SM	Nvidia profiler
γ	L2 cache hit rate of all transactions from SM	Nvidia profiler
L^m	Time consumption of one DRAM access transactions	Formula derivation
D^m	Average data access time of global memory considering L2 cache hit rate	Formula derivation
P^m	Time consumption over the link from SM to DRAM (or vice verse) of one transaction	Equation
T^{act}	Cycles for executing one round of active warps on a SM	Equation
$\overline{P}^m$	Average access latency of global memory considering L2 cache hit rate	Equation
L^{l2}	Time consumption of one L2 cache access transaction	Microbenchmarking
P^{l2}	Time consumption over the link from SM to L2 cache (or vice verse) of one transaction	Microbenchmarking
D^{l2}	Data access time on L2 cache of one transaction	Microbenchmarking
L^{sh}	Latency of one shared memory transaction	Microbenchmarking
T^i	Latency for the SM instruction type i	Microbenchmarking
T^m	Total execution time of multiple global memory requests	Microbenchmarking
P^d	Depth of pipeline	Microbenchmarking
P^{exe}	Total execution time of a kernel	Equation

4.1. SIMT and cache miss

The fine-grained P-chase method has its shortcomings, and this shortcoming lies in its failure to consider the impact of SIMT on L2 Cache access. Due to the existence of the SIMT mechanism, for a matrix operation, the first step is to partition the data, and according to the indexing calculation formula, different threads in a thread block correspond to different data for calculation. This leads to two memory access modes, namely the merged access mode and the independent access mode.

In our further research, we changed the number of variables on the right side of the addition equation, namely the variable value M in Equation 1, to control the amount of data that each thread in the thread bundle needs to load into the L2 Cache during the operation.

$$A[i] = \sum_{j=0}^{m} B[i][j] \quad (1)$$

4.2. GPU hardware data

The hardware we selected is GeForce Titan X, which has powerful performance. It has 28 multiprocessors, and each multiprocessor has 128 CUDA cores. In other words, Titan X GPU has a total of 3584 CUDA cores. The global memory size is 12196MB, and the constant memory size is 65536 bytes. Other data such as clock frequency and maximum thread count are shown below.

(1) Total amount of global memory: 12196 MBytes (12788498432 bytes)
(2) (28) Multiprocessors, (128) CUDA cores/MP: 3584 CUDA Cores
(3) GPU max clock speed: 1531 MHz (1.53 GHz)
(4) Memory clock speed: 5005 Mhz
(5) L2 cache size: 384-bit

(6) L2 cache size: 3145728 bytes
(7) Maximum texture dimension (x,y,z): 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
(8) Maximum layered 1D texture size, (number of layers): 1D=(32768), 2048 layers
(9) Maximum layered 2D texture size, (number of layers): 2D=(32768, 32768), 2048 layers
(10) Total amount of constant memory: 65536 bytes
(11) Total amount of shared memory per block: 49152 bytes

5. Experiment

5.1. Methodology

In this paper, we used 10 real GPU kernel functions as test benchmarks, similar to the ones of Rodinia [12] and Wang [7], as shown in **Table 2**. By using the NVIDIA Profiler[13] tool provided by NVIDIA Corporation, we extracted the performance data of the GPU kernel functions we tested. The parameters related to GPU hardware are fixed, while the rest of the kernel settings and code are different in different applications. Moreover, these performance data help us analyze the instruction distribution of the benchmark tests. Different kernels have different instruction distribution patterns. Some kernels contain a large number of texture memory transactions, while others contain a large number of shared memory transactions. These kernels pose challenges for designing an accurate and versatile performance model. In addition, these instruction statistics help us identify the main aspects that consume a large amount of time during kernel execution.

Table 2. Benchmarks and error

Abbreviation	Application name	Warps	Mean absolute percentage error：
VA	Vector Addition	131072	3.80%
MMG	MatrixMul (Global)	1024	7.25%
MMS	MatrixMUL (Shared)	2000	5.46%
BP	Backprop	3268	4.48%
BS	BlackScholes	131072	8.02%
CG	ConjugateGradient	131072	6.50%
FWT	FastWalshTransform	65536	8.30%
HSP	Hotspot	1095	5.66%
Hist	Histogram	1440	7.91%
NN	Near neighbor	1344	6.39%

5.2. Results

We used two metrics, the mean absolute percentage error ε(MAPE) relative to hardware measurement results, and the correlation coefficient (r) between modeling and measurement times, to evaluate our time-predictable model. The mean absolute percentage error measures the accuracy of our model, while the correlation coefficient tells us whether the trend of the model's results is similar to what we see in measurements. ε and r are calculated using equations 2 and 3, where x is the model's predicted value and y is the actual measured value. We repeated the experiments 100 times and calculated the average values. We list the ε values for predicting each kernel, and the experimental results are shown in **Table 2**.

$$\varepsilon = 100\% \times \frac{1}{N}\sum_{i=1}^{N}\frac{|x_i - y_i|}{y_i} \tag{2}$$

$$r = \frac{\sum_{i=1}^{N}(x_i - \bar{x}_i)(y_i - \bar{y}_i)}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x}_i)^2 \sum_{i=1}^{N}(y_i - \bar{y}_i)^2}} \tag{3}$$

6. Conclusion

In this paper, we propose a new GPU architecture model for analyzing and predicting the execution time of GPU kernel functions. This model builds upon previous work by further considering the impact of the SIMT mechanism and cache miss on kernel function execution time. In addition, it comprehensively considers more factors such as DRAM memory access latency and throughput, shared memory access latency and throughput, and pipeline depth. Experimental results showed that this method could provide relatively accurate time predictions for algorithm programs with different functionalities.

Disclosure statement

The author declares no conflict of interest.

References

[1] Hong S, Kim H, 2009. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. SIGARCH Comput. Archit. News, 37(3): 152–163.

[2] Wong H, Papadopoulou M-M, Sadooghi-Alvandi M, et al., 2010, Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), March 28–30, 2010: Demystifying GPU Microarchitecture Through Microbenchmarking. White Plains, 235–246.

[3] Mei X, Chu X, 2017, Dissecting GPU Memory Hierarchy Through Microbenchmarking. IEEE Transactions on Parallel and Distributed Systems, 28(1): 72–86.

[4] Amaris M, Cordeiro D, Goldman A, et al., 2015, Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), December 16–19: A Simple BSP-based Model to Predict Execution Time in GPU Applications. Bengaluru, 285–294.

[5] Liu G, Wang S, Bao Y, 2021, Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 26–29, 2021: SEER: A Time Prediction Model for CNNs from GPU Kernel's View, Atlanta, 173–185,

[6] Abdelkhalik H, Arafa Y, Santhi N, et al., 2022, Proceedings of 2022 IEEE High-Performance Extreme Computing Conference (HPEC), September 19–23, 2022: Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-Level Analysis. Waltham, 1–8.

[7] Wang Q, Chu X, 2020, GPGPU Performance Estimation with Core and Memory Frequency Scaling. IEEE Transactions on Parallel and Distributed Systems 31(12): 2865–2881.

[8] Restuccia F, Biondi A, 2021, Proceedings of the 2021 IEEE Real-Time Systems Symposium (RTSS), December 7–10, 2021: Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms, Dortmund, 441–454.

[9] Hong S, Kim H, 2010, Proceedings of the 37th Annual International Symposium on Computer Architecture, June 19–23, 2010: An Integrated GPU Power and Performance Model. Saint-Malo, 280–289.

[10] Song S, Su C, Rountree B, et al., 2013, Proceedings of the 2013 IEEE 27th International Symposium on Parallel and

Distributed Processing, May 20–24, 2013: A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures, Cambridge, 673–686.

[11] CUDA C++ Programming Guide, n.d., viewed September 9, 2023, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[12] Che S, Boyer M, Meng J, et al., 2009, Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), October 4–6, 2009: Rodinia: A Benchmark Suite for Heterogeneous Computing, 44–54.

[13] Preparing An Application for Profiling, n.d., viewed September 10, 2023, http://docs.nvidia.com/cuda/profiler-users-guide