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

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>
<thead>
<tr>
<th>Parameters</th>
<th>Definitions</th>
<th>How to achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D^S$</td>
<td>Data size of kernel</td>
<td>Kernel setup</td>
</tr>
<tr>
<td>$N^B$</td>
<td>Total number of blocks</td>
<td>Kernel setup</td>
</tr>
<tr>
<td>$D^W_{pb}$</td>
<td>Number of warps per block</td>
<td>Kernel setup</td>
</tr>
<tr>
<td>$N^w$</td>
<td>Total number of warps of kernel</td>
<td>Kernel setup</td>
</tr>
<tr>
<td>$N^{GST}$</td>
<td>Number of Global load/store transactions per warp</td>
<td>Nvidia profiler</td>
</tr>
</tbody>
</table>
Table 1. (Continue)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Definitions</th>
<th>How to achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^c$</td>
<td>Number of computation instructions of kernel</td>
<td>Nvidia profiler</td>
</tr>
<tr>
<td>$N^{aw}$</td>
<td>Number of active warps</td>
<td>Nvidia profiler</td>
</tr>
<tr>
<td>$N^{cw}$</td>
<td>Number of concurrent warps per SM</td>
<td>Nvidia profiler</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>L2 cache hit rate of all transactions from SM</td>
<td>Nvidia profiler</td>
</tr>
<tr>
<td>$L^a$</td>
<td>Time consumption of one DRAM access transactions</td>
<td>Formula derivation</td>
</tr>
<tr>
<td>$D^a$</td>
<td>Average data access time of global memory considering L2 cache hit rate</td>
<td>Formula derivation</td>
</tr>
<tr>
<td>$P^a$</td>
<td>Time consumption over the link from SM to DRAM (or vice verse) of one transaction</td>
<td>Equation</td>
</tr>
<tr>
<td>$T^{aw}$</td>
<td>Cycles for executing one round of active warps on a SM</td>
<td>Equation</td>
</tr>
<tr>
<td>$P^l$</td>
<td>Average access latency of global memory considering L2 cache hit rate</td>
<td>Equation</td>
</tr>
<tr>
<td>$L^{ac}$</td>
<td>Time consumption of one L2 cache access transaction</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$P^c$</td>
<td>Time consumption over the link from SM to L2 cache (or vice verse) of one transaction</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$D^c$</td>
<td>Data access time on L2 cache of one transaction</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$L^{sh}$</td>
<td>Latency of one shared memory transaction</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$T^i$</td>
<td>Latency for the SM instruction type i</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$T^m$</td>
<td>Total execution time of multiple global memory requests</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$P^d$</td>
<td>Depth of pipeline</td>
<td>Microbenchmarking</td>
</tr>
<tr>
<td>$P^{exe}$</td>
<td>Total execution time of a kernel</td>
<td>Equation</td>
</tr>
</tbody>
</table>

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.

\[
A[i] = \sum_{j=0}^{m} B[i][j]
\]

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
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

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

5.2. Results

We used two metrics, the mean absolute percentage error $\varepsilon$(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. $\varepsilon$ 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 $\varepsilon$ values for predicting each kernel, and the experimental results are shown in Table 2.
\[\varepsilon = 100\% \times \frac{1}{N} \sum_{i=1}^{N} \frac{|x_i - y_i|}{y_i} \]  
\[r = \frac{\sum_{i=1}^{N} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{N} (x_i - \overline{x})^2 \sum_{i=1}^{N} (y_i - \overline{y})^2}} \]

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 misses 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


Publisher’s note
Bio-Byword Scientific Publishing remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.