GPU Performance Modeling and Analysis Using Software and Hardware Characteristics

Xuan Wang[1] **and Alexia Lou**[2]

[1,2]New York University, Department of Computer Science, New York, 10003, U.S.A
[1,2]{xwang, al5105}@nyu.edu
[+]equally contributed to this work

ABSTRACT

Our work implements a performance model for GPU TITAN X architecture based on warp level computation and memory parallelism. We further take into account the hardware feature of L2 cache and show that the consideration of L2 cache provides a better prediction (error within 8%) than only of global memory. We do the parameter calculations without the need to run the application. Only information on hardware spec, program characteristics and microbenchmarks which are independent of the application are needed.

Keywords: GPU, Performance Modeling, CUDA, Nvidia TITAN X

1 Introduction

1.1 Motivation and Related Work

The massively parallel GPU, since 2002, has evolved from a graphics-specific accelerator to a general-purpose computing device. With the accompanied programming frameworks like CUDA[1] and OpenCL[2], an active research and development community has formed to develop high-performance general-purpose applications and algorithms for GPUs[3]. However, compared to the enormous amount of efforts devoted to application and algorithm development, little has been done in the reverse direction that is performance profiling and analysis. Commercial program profiling tools such as NVIDIA Parallel Nsight[5], along with academic GPU functional simulators[6, 7], are limited to collect program statistics but do not relate the statistics to program performance. And the GPU performance modeling is highly related to the actual architecture in use. Hence GPU performance modeling remains a difficult and but important task.

Hong and Kim[12] estimate the number of parallel memory requests (they call the memory warp parallelism) by considering the number of running threads and memory bandwidth. Based on the degree of memory warp parallelism, the model estimates the cost of memory requests, thereby estimating the overall performance of a program. Such analytical approaches are also taken by Baghsorkhi et al.[11] and Kim et al.[8], they both designed abstract performance models of both computation and memory at a high level. Instead of trying to build an analytical model based on an abstraction of GPU architecture and then verifying the model by microbenchmarks, Zhang and Owens[9] adopt the reverse strategy: first design microbenchmarks, observe the benchmark results, and then break down the performance of GPU into three major components: the instruction pipeline, shared memory access, and global memory access.

However, the previous works are mostly done on Tesla devices that there were few cache mechanisms implemented. With the introduction of newer generation devices, e.g. Maxwell, efficient L1 and L2 caches are present, which makes the older models not give the accurate predictions.

Recently there are also Machine Learning GPU performance models [14, 15, 16, 17, 18] , multiple methods are used and some obtain considerable accuracy. However, these works strongly rely on training data such as specific performance counters and kernel settings. Even they can reveal some correlations between the input parameters and execution time, they still lack of clear explanation of the clear factors of performance which are of more practical importance in development.

Our work holds the belief that in real production it will be best and most practical to use minimum effort to get good heuristics of the performance of the application, so our work focuses on the CUDA program itself and only uses static analysis (or statistics collected by running independent microbenchmarks[12]) without the need to actually run the application. We use disassembled assembly code by using cuobjdump[13], which gives more accurate information, than intermediate representation(e.g. PTX [10]), about how the CUDA program got run on device, in order to calculate instruction level latency statistics.

1.2 Outline of Our Work

In our work, we analyze the performance of GPU along two lines: memory-wise and computation-wise. We first model the two lines of performance in approximate formulas based the work of Hong and Kim[19] but include the consideration of L2 cache, and then do parameter estimation using characteristics collected from both software and hardware sides.

This paper is structured as follows:

section 2 introduces the performance modeling methodology;
section 3 describes the details of how to calculate and estimate model parameters;
section 4 presents experiments and analysis;
and section 5 concludes the work.

2 Performance Modeling

The performance modeling is based on the work Hong and Kim[19]. We extend the model to take into consideration of L2 cache, which turns out to be crucial in predicting the performance (see section 4).

Warp is the basic unit of scheduling in GPU. Each SM can schedule multiple warps in a time-sharing fashion. Therefore, when analyzing the performance of GPU, it is natural to estimate the parallelism on the level of warp. We introduce MWP and CWP as in [19].

MWP is the maximum number of warps that can simultaneously access memory, whereas CWP represents the number of warps that are able to execute during one memory access period plus one (the warp which issues the memory access).

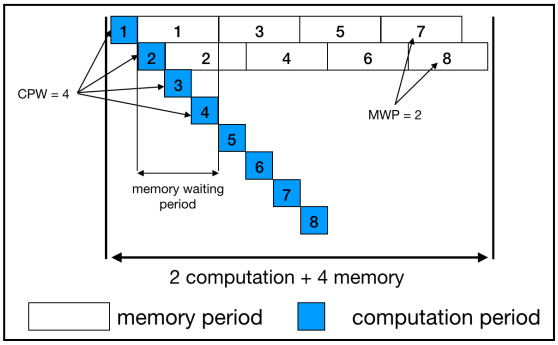

Figure 1. Execution Pattern $CWP >= MWP$

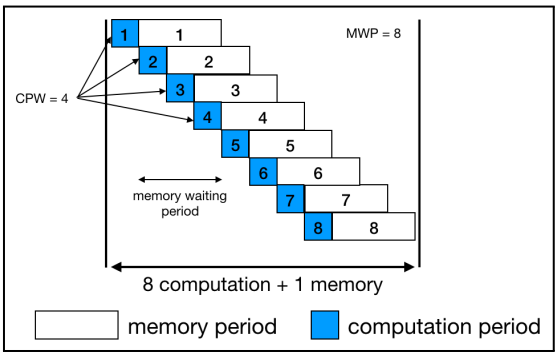

Figure 2. Execution Pattern $CWP < MWP$

2.1 When CWP is Greater Than MWP

In this case (Figure 1) the execution cost is dominated by memory access cost; all computation periods overlapped under memory waiting periods. Therefore,

$$Exec\_cycle = Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{\#Mem\_insts} \times MWP \tag{1}$$

Mem_cycles: Memory waiting cycles per warp (Equation 14).
$Comp\_cycles$: Computation cycles per warp (Equation 15).
$\#Mem\_insts$: Number of memory instructions per warp.
N: Number of active running warps per SM (Equation 16).

2.2 When MWP is Greater Than CWP

On the other hand, when $CWP < MWP$ (Figure 2), the execution cost is dominated by the computation cost; only part of the computation periods overlap with memory waiting period, resulting

$$Exec\_cycles = Mem\_L + Comp\_cycles \times N \tag{2}$$

2.3 Not Enough Warps

All above situation assume that we have enough warp running per SM. However, if an application does not have enough warps, the system cannot take advantage of all available zero-cost scheduling warp parallelism. In this case, both MWP and CWP cannot exceed N, the number of active warps per SM. Then

$$Exec\_cycles = Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_insts}(MWP - 1) \tag{4}$$

2.4 The Effect of L2 Cache

Hong and Kim's model is on earlier generations of GPU where there are no L2 cache. But in Maxwell, GPU are provided with efficient L2 cache. Only a tenth of latency is with L2 cache than with global memory. And when in the case of L2 cache hit, interesting situations can happen that the warp starts later may finish earlier because earlier warp's access to global memory help bring cache blocks to L2 cache.

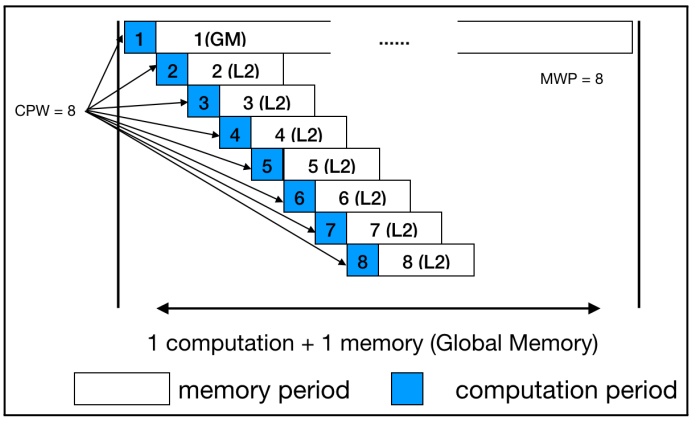

Figure 3. Execution Pattern when caching memory in L2 cache

2.5 The Whole Story

Assume each SM access its memory system and each active SM consume an equal amount of memory bandwidth. Let Mem_L be the latency of each memory warp, then MWP can be formulated as:

$$MWP = \min(\frac{Mem\_L}{Departure\_delay}, MWP_{peak\_bw}, N) \tag{5}$$

$$MWP_{peak\_bw} = \frac{Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM} \tag{6}$$

$$BW\_per\_warp = \frac{Freq \times Load\_bytes\_per\_warp}{Mem\_L} \tag{7}$$

Since the latency depends on whether the memory access coalesced, it is calculated as

$$Mem\_L = Mem\_L\_Uncoal \times Weight_{uncoal} + Mem\_L\_Coal \times Weight_{coal} \tag{8}$$

The parameters for above equation are calculated in Equation 11 - 15.
Calculating CWP is straightforward.

$$CWP = \min(CWP\_full, N) \tag{9}$$

$$CWP\_full = \frac{Mem\_cycle + Comp\_cycle}{Comp\_cycle} \tag{10}$$

Putting everything together, assuming the total number of warps scheduled to run for an application is $\#Rep$ (Equation 19), then the total execution cycle of an application can be estimated using equation 20, 21 and 22.

$$Mem\_L\_Uncoal = Mem\_LD + (\#Uncoal\_per\_mw - 1) \times Departure\_del\_uncoal \tag{11}$$

$$Mem\_L\_Uncoal = Mem\_LD \tag{12}$$

$$Weight_{uncoal} = \frac{\#Uncoal\_Mem\_insts}{\#Uncoas\_Mem\_insts + \#Coal\_Mem\_insts} \tag{13}$$

$$Weight_{coal} = \frac{\#Coal\_Mem\_insts}{\#Uncoas\_Mem\_insts + \#Coal\_Mem\_insts} \tag{14}$$

$$Departure\_delay = (Departure\_del\_uncoal \times \#Uncoal\_per\_mw) \times Weight_{uncoal} + Departure\_del\_coal \times Weight_{coal} \tag{15}$$

$$Mem\_cycles = Mem\_L\_Uncoal \times \#Uncoal\_Mem\_insts + Mem\_L\_Coal \times \#Coal\_Mem\_insts \tag{16}$$

$$Comp\_cycles = \#Issue\_cycles \times (\#total\_insts) \tag{17}$$

$$N = \#Active\_warps\_per\_SM \tag{18}$$

$$\#Rep = \frac{\#Blocks}{N \times \#Active\_SM} \tag{19}$$

if (MWP = N) and (CWP = N),

the first warp with a cold cache miss

$$First\_warp\_cycles = Mem\_cycles + Comp\_cycles$$

the last warp cycles in its best case where its memory access are L2 cache hit

$$Last\_warp\_cycles = First\_comp\_period\_insts \times \#Issue\_cycles \times (MWP-1) + Mem\_cycles \times Mem\_L2\_cache\_hit / Mem\_L2\_cache\_miss$$

$$Exec\_cycles\_app = max(Last\_warp\_cycles, First\_warp\_cycles) \times \#Rep \tag{20}$$

else if (CWP $\geq$ MWP) or (Comp_cycles $>$ Mem_cycles)

$$\#Cache\_hit\_mem = max(\frac{N}{MWP} - 1, 0)$$

$$\begin{aligned} Exec\_cycles\_app = (&Mem\_cycles + Mem\_cycles \times Mem\_L2\_cache\_hit / Mem\_L2\_cache\_miss \times \#Cache\_hit\_mem \\ &+ \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP-1)) \times \#Rep \end{aligned} \tag{21}$$

else

$$First\_warp\_cycles = Mem\_L + Comp\_cycles$$

$$Last\_warp\_cycles = First\_comp\_period\_insts \times \#Issue_cycles \times N + Mem\_l \times Mem\_L2\_cache\_hit / Mem\_L2\_cache\_miss$$

$$Exec\_cycles\_app = max(Last\_warp\_cycles, First\_warp\_cycles) \times \#Rep \tag{22}$$

3 Characteristics Collection and Parameter Estimation

3.1 Summary of Parameters

	Model Parameter	Definition	Obtained
1	#Threads_per_warp	Number of threads per warp	Hardware spec
2	Issue_cycles	Number of cycles to execute one instruction	Hardware spec[20]
3	Freq	Clock frequency of the SM processor	Hardware spec
4	Mem_Bandwidth	Bandwidth between the DRAM and GPU cores	Hardware spec
5	Departure_del_uncoal	Delay between two uncoalesced memory transactions	Hardware spec
6	Departure_del_coal	Delay between two coalesced memory transactions	Hardware spec
7	Load_bytes_per_warp	Number of bytes for each warp = 32 * 4 bytes = 128 bytes	Hardware spec
8	Mem_LD = Mem_L2_cache_miss	global memory access latency	Microbenchmark[12]
9	#Threads_per_block	Number of threads per block	Kernel setup
10	#Blocks	Total number of blocks in a program	Kernel setup
11	#Active_SMs	Number of active SMs	Kernel setup
12	#Active_blocks_per_SM	Number of concurrently running blocks on one SM	Kernel setup
13	#Active_warps_per_SM(N)	Number of concurrently running warps on one SM	Kernel setup
14	#Total_insts	(#Comp_insts + #Mem_insts)	Code analysis
15	#Comp_insts	Total dynamic number of computation instructions in one thread	Code analysis
16	#Mem_insts	Total dynamic number of memory instructions in one thread	Code analysis
17	#Uncoal_Mem_insts	Number of uncoalesced memory type instructions in one thread	Code analysis
18	#Coal_Mem_insts	Number of coalesced memory type instructions in one thread	Code analysis
19	#Coal_per_mw	Number of memory transactions per warp (coalesced access)	Code analysis
20	#Uncoal_per_mw	Number of memory transactions per warp (uncoalesced access)	Code analysis
21	First_comp_period_insts	Number of instructions before the first memory instruction	Code analysis

Table 1. Summary of Model Parameters

3.2 Hardware Related Parameters

These parameters can be easily obtained from hardware specs[21] or from simple calculations.

When memory uncoalesced, the departure delay depends how many 128-byte blocks the memory instruction will require. For example, consider the kernel when saxpy has an uncoalesced access of a stride of 4:

```
1 __global__ void saxpy(int n, float a, float *x, float *y)
2 {
3         int i = blockIdx.x * blockDim.x + threadIdx.x;
4         if (i < n) {
5                 y[i] = a * x[i*4] + y[i];
6         }
7 }
```

Then for a warp of size 32, the x[i*8] will span a memory block $sizeof(float) \times 32 * 4/128 = 4$, and thus it takes 4 memory transactions. And according the work of [22], the departure delay will be 4 * 1 = 4 cycles.

3.3 Code Analysis

We analyze the disassembliy code using cuobjdump. Let us look at an saxpy example code.

```
1           MOV R1, c[0x1][0x100];
2           S2R R0, SR_CTAID.X;                          // prepare block idx
3           S2R R2, SR_TID.X;                            // prepare thread idx
4           IMAD R0, R0, c[0x0][0x8], R2;
5           ISETP.GE.AND P0, PT, R0, c[0x0][0x20], PT;   // set branch predicate
6       @P0 BRA.U 0x78;
7      @!P0 MOV32I R5, 0x4;
8      @!P0 IMAD R2.CC, R0, R5, c[0x0][0x28];           // idx calculation
9      @!P0 IMAD.HI.X R3, R0, R5, c[0x0][0x2c];         // idx calculation
10     @!P0 IMAD R4.CC, R0, R5, c[0x0][0x30];           // idx calculation
11     @!P0 LD.E R2, [R2];                              // memory load
12     @!P0 IMAD.HI.X R5, R0, R5, c[0x0][0x34];         // integer multiplication
13     @!P0 LD.E R0, [R4];                              // memory load
14     @!P0 FFMA R0, R2, c[0x0][0x24], R0;              // fused float multiply and add
15     @!P0 ST.E [R4], R0;                              // memory store
```

So we can easily see that there are 3 memory instructions out of a total of 15 instructions. The first memory instruction is at line 11, so $First\_comp\_period\_insts = 10$. Other parameters are simple to get and their values are included in the params.yml file.

4 Experiments

Because the performance model does not make any assumption on the program, for simplicity but without the loss of generality we do experiments using the saxpy example on both coalesced and uncoalesced memory access patterns.

4.1 Coalesced

We can see from Figure 4 that the model performs quite well when memory is coalesced.

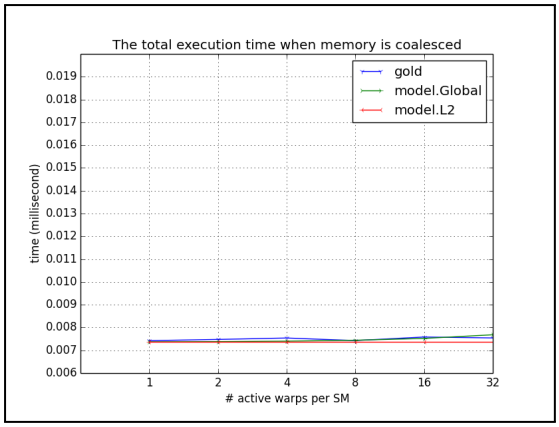

Figure 4. Coalesced

4.2 Uncoalesced

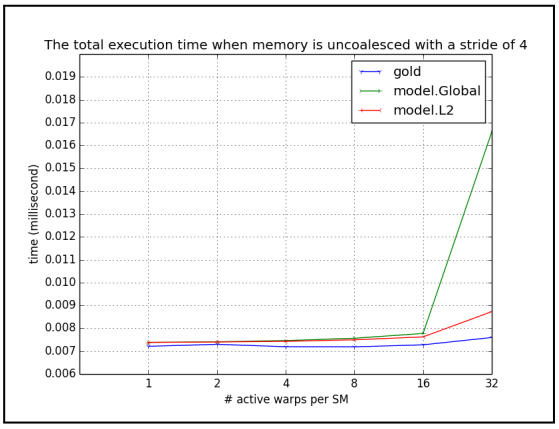

Figure 5. Uncoalesced with a stride of 4

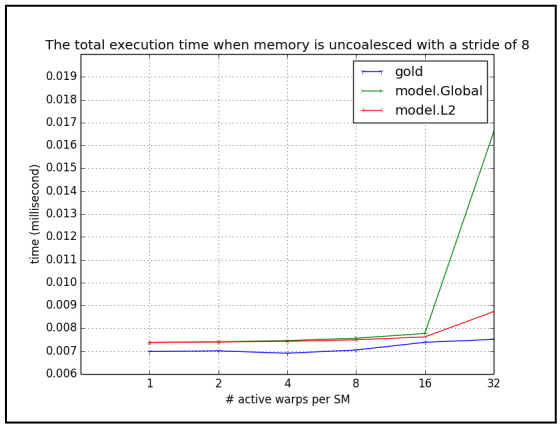

Figure 6. Uncoalesced with a stride of 8

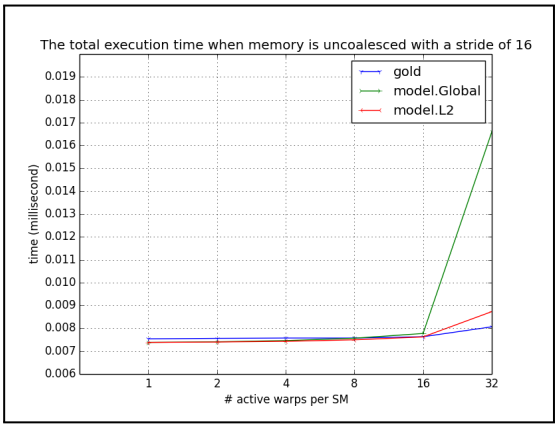

Figure 7. Uncoalesced with a stride of 16

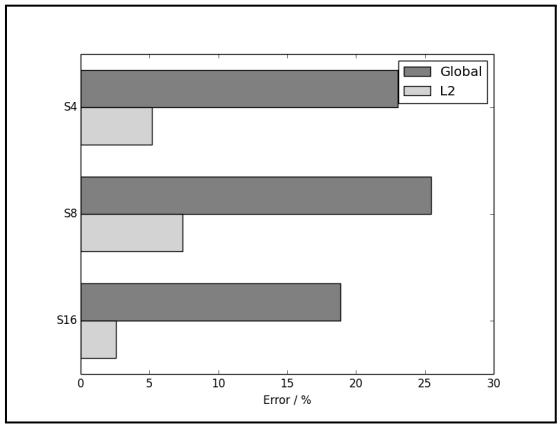

Figure 8. Relative prediction error when different uncoalesced strides

We can see that when memory is uncoalesced, if the number of active warps per SM is small, both model settings are good. But when the number of warps further approaches bigger, then only take into account of the global memory will result in a considerable error. But if we take into account the L2 cache, we still get quite good prediction.

This gives us insight that the L2 cache hit rate is very high in the TITAN X GPU. And this observation agrees with the recent Wang and Chu's work [22].

5 Conclusion

Our work implements a performance model for GPU TITAN X architecture based on warp level computation and memory parallelism. We further take into account the hardware feature of L2 cache and show that the consideration of L2 cache provides a better prediction (error within 8%) than only of global memory. We do the parameter calculations without the need to run the application. Only information on hardware spec, program characteristics and microbenchmarks which are independent of the application are needed.

Under the constraint of time, we only consider the most general and critical aspects of performance modeling. We make a few simplifications: 1) we do not consider synchronization effects; 2) we do not consider branch divergence. However, the performance modeling is straightforward to extend on the limitations as in the work of [19].

References

[1] "NVIDIA CUDA compute unified device architecture, programming guide," http://developer.nvidia.com/.
[2] "The OpenCL specification," http://www.khronos.org/ registry/cl/.
[3] "General-purpose computation using graphics hardware," http://www.gpgpu.org/.
[4] "ATI Stream Profiler," http://developer.amd.com.
[5] "NVIDIA Parallel Nsight," http://developer.nvidia. com.
[6] S. Collange, D. Defour, and D. Parello, "Barra, a parallel functional GPGPU simulator," Université de Perpignan, Tech. Rep. hal-00359342, Jun. 2009.
[7] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-01, 2009.
[8] Kim, Hyesoon, et al. "Performance analysis and tuning for general purpose graphics processing units (GPGPU)." Synthesis Lectures on Computer Architecture 7.2 (2012): 1-96.
[9] Zhang, Yao, and John D. Owens. "A quantitative performance analysis model for GPU architectures." High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on. IEEE, 2011.
[10] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009), Oct. 2009, pp. 3–12.
[11]Baghsorkhi, Sara S., et al. "An adaptive performance modeling tool for GPU architectures." ACM Sigplan Notices. Vol. 45. No. 5. ACM, 2010.
[12]Mei, Xinxin, and Xiaowen Chu. "Dissecting GPU memory hierarchy through microbenchmarking." IEEE Transactions on Parallel and Distributed Systems 28.1 (2017): 72-86.
[13]http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#cuobjdump
[14] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on. IEEE, 2015, pp. 564–576.
[15] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, "Power and performance characterization and modeling of gpu-accelerated systems," in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014, pp. 113–122.
[16] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013, pp. 673–686.
[17] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in Green Computing Conference, 2010 International. IEEE, 2010, pp. 115–122.
[18] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," Parallel and Distributed Systems, IEEE Transactions on, vol. 26, no. 7, pp. 1800–1813, 2015.
[19] Hong, Sunpyo, and Hyesoon Kim. "Memory-level and thread-level parallelism aware GPU architecture performance analytical model." (2009).
[20]www.cs.cornell.edu/courses/cs3410/2015sp/lecture/08-perf-pipeline-i.pptx
[21]https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications
[22]Wang, Qiang, and Xiaowen Chu. "GPGPU Performance Estimation with Core and Memory Frequency Scaling." arXiv preprint arXiv:1701.05308 (2017).