A Performance Analysis Framework for Exploiting GPU Microarchitectural Capability

Keren Zhou Guangming Tan Xiuxia Zhang Chaowei Wang Ninghui Sun
Institute of Computing Technology, Chinese Academy of Sciences
{zhoukeren,tgm,zhangxiuxia,wangchaowei,snh}@ncic.ac.cn

ABSTRACT

GPUs are widely used in accelerating deep neural networks (DNNs) for their high bandwidth and parallelism. But tuning the performance of DNN computations is challenging, as it requires a thorough understanding of both underlying architectures and algorithm implementations. Traditional research, which focused on analyzing performance by CUDA C language or PTX instructions, has not combined hardware features tightly with source code. In this paper, we present a performance analysis framework at the assembly level. First, an *instruction parser* takes assembly source code, benchmark results, and hardware features as input to identify each instruction's efficiency and latency. Then, a *DAG constructor* builds a DAG that models instruction executions. Finally, a *performance advisor* incorporates block partitions, occupancy, and the generated DAG to predict running cycles of the source code and presents its potential bottlenecks. We demonstrate the effectiveness of our framework by optimizing DNNs' performance-critical kernels–GEMM and convolution. After taking steps to reduce bottlenecks, the experimental results show that our GEMM is 20% faster than cuBLAS, and our convolution outperforms cuDNN by 40%-60%. Because of the usage of assembly instructions, we can predict performance with an error as low as 2% in average.

CCS CONCEPTS

•Software and its engineering → Assembly languages;

KEYWORDS

GPUs, GEMM, Convolution, Performance Model

ACM Reference format:
Keren Zhou Guangming Tan Xiuxia Zhang Chaowei Wang Ninghui Sun. 2017. A Performance Analysis Framework for Exploiting GPU Microarchitectural Capability. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079083

1 INTRODUCTION

Recently, the spring up of DNN applications causes a surge of research on optimizing DNN computations on different architectures [3, 16, 17, 24, 25, 28]. Among these devices, GPUs have high memory bandwidth and a massive number of threads, satisfying a vast quantity of training data and parameters. Deep learning frameworks, such as Caffe [10], Torch [6], Theano [9], and cuDNN [4], embraced GPU-optimized implementations; previous research also focused on efficient algorithms [15, 17], parallelism approaches [12, 24], and memory efficiency [16] for DNN computations on GPUs. However, they have not built a comprehensive model to link architectural characteristics with algorithms and kernel designs. More than that, even the state-of-the-art libraries–cuDNN and cuBLAS only achieve about 60%-70% theoretical performance on Kepler GPU. Thus, it is urgent to build a model that helps programmers understand the performance bottlenecks with regard to underlying architectures and algorithms and direct them to optimize applications.

Although existing models on GPUs could analyze applications either at run-time [27] or in a static way [5, 14, 14, 22], they are insufficient to help programmers optimize source code under specific architectures at the development stage. The primary limiting factor is the usage of CUDA C language or PTX instructions [19] that are not feasible for programmers to do performance tuning for two reasons: (1) They do not contain control code that allows programmers to manipulate issue behavior, which is important for promoting the performance. For example, with the help of control code, we can activate the instruction dual-issue mechanism. (2) A PTX instruction or a C statement might map to many assembly instructions so that we cannot precisely measure its latency and count the number of instructions. Therefore, it is demanding for us to analyze programs at the assembly level.

To solve the aforementioned problems, we propose a performance analysis framework based on GPU assembly instructions, directing programmers or compilers to generate highly optimized code. The framework consists of three parts. First, we devise an *instruction parser* to extract instruction dependencies and calculate instruction efficiencies. The second component is a *DAG constructor* that links instructions by their dispatch order and dependencies to generate a DAG. Besides, it calculates block partitions to get the number of iterations the DAG executes and links hardware resource constraints with source code resource usage to derive occupancy on SMs. Third, a *performance advisor* estimates the running cycles for the source code by incorporating the DAG, block iterations, and occupancy. The advisor also indicates potential bottlenecks and in turn helps programmers tune their applications by modifying assembly instructions. To summarize, we have made the following contributions:

- We built a performance analysis framework at the assembly level, which benchmarks instruction characteristics, estimates running cycles, and points possible bottlenecks of given source code.

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079083

- We demonstrated the effectiveness of our framework by optimizing GEMM and convolution on Kepler GPU. Under the direction of performance advisor, we can eliminate each bottleneck step by step with register blocking, high bandwidth memory instruction, instruction scheduling, instruction dual-issue, and read-only cache.
- In the experiments, our GEMM achieved 88% peak efficiency, which outperforms cuBLAS with up to 20%; our convolution achieved 83% peaking efficiency, which is 40%-60% faster than cuDNN in a variety of network configurations. Our analyzer accurately predicted the running time. Comparisons between the actual results and prediction of the model show that average error is 2%.

We organize the rest of the paper as follows. Section 2 reviews GPU architectures and DNN computations. Section 3 describes our performance analysis framework. Section 4 illustrates the effectiveness of our model by optimizing two core computations in DNNs. Section 5 presents related works. Section 6 concludes and describes future research.

2 BACKGROUND

2.1 GPU Architectures

Modern GPUs are designed for compute-intensive applications, such as DNN computations. In this part, we review general designs for NVIDIA GPUs. A GPU is composed of a piece of global memory and many streaming multiprocessors (SMs). Each SM contains a variety of function units: streaming processors (SPs) that have floating point units (FPUs) and integer units (INTUs) inside, double precision units (DPUs), special function units (SFUs), and load store units (LDSTs). The SM also has a piece of low latency shared memory, on which users can explicitly allocate and access data. With the development of GPU architectures, NVIDIA incorporates cache hierarchy into its design. It has L1 cache and read-only cache on each SM and L2 cache shared by all SMs.

A warp is the minimum scheduling unit on GPUs. An SM may have multiple warp schedulers to manage warps. Recent generations of GPUs contain multiple dispatch units within a scheduler that issue instructions simultaneously. Whereas if two schedulers contend on the same function units, one of them has to wait for another to issue first. An SM holds multiple blocks that maintain several warps. The number of blocks resides on an SM concurrently is called active blocks, which is determined by the resource constraints such as available registers and shared memory.

2.2 DNN Computations

A neural network is composed of many neurons that serve as basic computational units. Each neuron is connected with others and may have a function which combines its inputs together. Besides, some neural networks organize neurons as multiple layers in which the latter layers take inputs from the formers. DNNs take independent samples, called batch, to apply the function. Suppose that the batch size is N, and a neuron is connected to every neuron in the previous layer, we derive Equation 1 which represents the fully connected layer. For simplicity, we use I to denote the input layer, O to denote the output layer, and W to denote the weight on links.

$$O[n][i] = \sum_{k=0}^{K} I[n][k] \times W[k][i] \tag{1}$$

$O[n][i]$: The i_{th} element in the n_{th} output sample.
$I[n][k]$: The k_{th} element in the n_{th} input sample.
$W[k][i]$: The weight on the link of k_{th} input element to the i_{th} output element.
We could apply a highly optimized GEMM routine for the above equation. Various kinds of DNN computations, such as multi-layer perceptron (MLP), recurrent neural network (RNN), and convolution neural network (CNN) follow the above computation patterns. Therefore, the core of optimizing DNN computations lies on boosting the performance of GEMM and hiding memory movement latency within floating point computations.

A convolution layer extracts features from input samples by applying a dot product with a filter, repeating the procedure stride by stride along each input channel, and combining the results into output channels. Input, filter, and output are 4-D tensors. To differentiate their dimensions, we adopt similar notations as [15]: N (batch size), C (input channels), H (input height), W (input width), K (output channels), P (output height), Q (output width), R (filter height), and S (filter width). Input samples have $NCHW$ dimensions, filters have $KCRS$ dimensions, and output samples have $NKPQ$ dimensions. We use Equation 2 to present the convolution computation.

$$O[n][k][i][j] = \sum_{c=0}^{C} \sum_{r=0}^{R} \sum_{s=0}^{S} I[n][c][i \times stride + r][j \times stride + s] \times F[k][c][r][s] \tag{2}$$

$O[n][k][i][j]$: The element on i_{th} row and j_{th} column in the k_{th} output channel of n_{th} output sample.
$F[k][c][r][s]$: The element on r_{th} row and s_{th} column in the c_{th} input channel and k_{th} output channel of the filter.
$I[n][c][i \times stride + r][j \times stride + s]$: The element on the corresponding position indexed by output row i, column j and $stride$ in the c_{th} input channel of n_{th} input sample.
We observe that the above equation could be viewed as a GEMM process after collapsing some dimensions. In particular, we collapse the last two dimensions of O, the last three dimensions of F, and the last three dimensions of I. We also record the offsets of the last three dimensions in I in an $index$ array.

$$O[n][k][ij] = \sum_{cij=0}^{C \times R \times S} I[n][index[cij]] \times F[cij][k] \tag{3}$$

In this way, each element of O is calculated via a GEMM kernel. We call this method direct convolution [11] since it is derived directly from the definition. Unlike FFT and Winograd methods [7, 15], the method is straightforward and does not involve constraints such as $stride = 1$. We will refer to this method as convolution in following sections.

3 PERFORMANCE ANALYSIS FRAMEWORK

We developed a performance analysis framework at the assembly level, shown in Figure 1. At the outset, we benchmark essential instruction features. We insert test instructions into a code snippet,

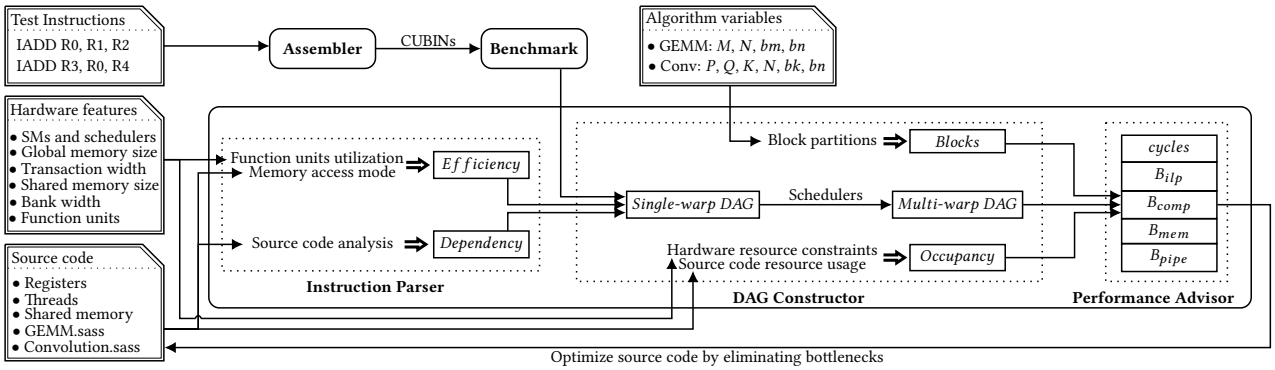


Figure 1: Assembly level performance analysis framework

wrapped by time measurement instructions. Then, we adopt existing GPU assemblers [26] according to specific GPU architectures. After compiling the snippet to CUBINs, our benchmark program invokes them and outputs instruction latencies.

The *instruction parser* associates the source code with hardware features to analyze each instruction's function unit utilization and memory access mode. It outputs instruction efficiencies and dependencies.

The *DAG constructor* starts by combing benchmark results, instruction efficiencies and dependencies to generate a DAG at the single-warp level. Then, the DAG is extended to multi-warp by summing up dispatches of all schedulers. We use hardware resource constraints and source code resource usage, such as shared memory and register usage, to get *occupancy* that decides the number of active warps that execute the DAG concurrently. We uses block partitions by algorithm variables and the kernel size to determine the number of iterations the DAG runs.

With the DAG, occupancy, and number of blocks, the *performance advisor* estimates the running cycles of the source code. More than that, the advisor proposes potential bottlenecks for programmers, including instruction level parallelism (B_{ilp}), compute efficiency (B_{comp}), memory efficiency (B_{mem}), and instruction pipeline (B_{pipe}). Last, by modifying assembly instructions in the source code, we can ultimately achieve low bottlenecks to gain extreme performance.

Algorithm 1 Instruction efficiency

```
1:  Input
2:  asmSet : a set of asm structures
3:  ▷ asm − ⟨name, control, modifier, dest, source1, source2...⟩
4:  Output
5:  insSet : a set of ins structures
6:  ▷ ins − ⟨asm1, asm2...⟩
7:  effMap : a map of asm → efficiency
8:  insSet ← 0
9:  effMap ← 0
10: i ← 0
11: while i < asmSet.size do
12:     if dual(asmSet[i].control) then
13:         insSet.push(⟨asmSet[i], asmSet[i + 1]⟩)
14:         i ← i + 2
15:     else
16:         insSet.push(⟨asmSet[i]⟩)
17:         i ← i + 1
18:     end if
19: end while
20: for i ← 0, insSet.size do
21:     for all asm ∈ insSet[i] do
22:         units ← typeof(asm.name)
23:         dispatches[units] ← dispatches[units] + 1
24:     end for
25:     for all asm ∈ insSet[i] do
26:         units ← typeof(asm.name)
27:         if units == mem then
28:             effMap[asm] ← mem_eff(asm, dispatches[units])
29:             ▷ Equation 5, 6
30:         else
31:             effMap[asm] ← comp_eff(asm, dispatches[units])
32:             ▷ Equation 4
33:         end if
34:     end for
35:     dispatches ← 0
36: end for
```

3.1 Instruction Parser

3.1.1 Function Units Utilization. We define *efficiency E* to be the reciprocal of cycles needed to dispatch a given number of instructions, and present major steps to calculate efficiency in Algorithm 1. Its input is an abstract set of the source code, consisting of instruction name, modifier, control code, and operands (*dest* and *sources*). Because some generations of GPUs support instruction dual-issue mechanism that a warp scheduler could issue two instructions simultaneously, we should examine the control code and put dual-issued instructions in the same entry (Line 11-19). For a compute instruction, Line 32 invokes Equation 4 to calculate its efficiency. Suppose that there are several function units in an SM, we use E_u to denote the efficiency for function unit u. And for $instruction_i$, we use u_i for its functions unit. As denoted in Equation 4, the efficiency depends on the ratio of requested function units to available function units.

$$E_{u_i} = \frac{1}{\lceil \frac{Dispatches_{u_i} \times Warp\_size}{Units_{u_i}} \rceil} \tag{4}$$

3.1.2 Memory Access Mode. Line 29 invokes a separate function for memory instructions. Unlike compute instructions, memory instructions' efficiency vary for their different access modes. A memory instruction is issued to LDST units, causing memory transactions that move a sequence of data between two regions of

Table 1: Summary of parameters

Name	Description	Source
S	Number of warp schedulers in an SM	Hardware features
U	Categories of different function units in an SM	Hardware features
$Units_{u_i}$, $Units_{ldst}$	Number of function units in an SM	Hardware features
$Warp\_size$	Number of threads in a warp	Hardware features
$Bank\_width$	Width of a shared memory bank	Hardware features
NSM	Number of SMs in a GPU	Hardware features
$SM\_threads$, $SM\_registers$, SM_shared	Size of threads, register, and shared memory in an SM	Hardware features
Max_width_i	Maximum bandwidth for $instruction_i$ by adjusting modifiers	Hardware features
$Latency_{global}$, $Latency_{amat}$, $Latency_{hit}$, $Latency_i$	Instruction latencies	Benchmark
N_i	Number of memory transactions for $instruction_i$	Memory access mode
Ins_width_i	Width for $instruction_i$	Memory access mode
$Dispatches_{u_i}$	Number of dispatches used for function units	Function units utilization
E_{u_i}, E_{shared}, E_{global}	Efficiencies for different types of function units	Function units utilization
I	Number of instructions	Source code analysis
I_u, I_{comp}, I_{mem}	Number of instructions for different function units	Source code analysis
I_{lat}	Number of edges in the critical path weighted by latency	DAG
C_w, C_b, C_k	Cycles of a warp, a block and a kernel	DAG
$C_{w_{eff}}$	Cycles of a warp without instruction latency	DAG
C_{w_u}	Cycles of a warp with E_u	DAG
$Block\_threads$, $Block\_registers$, $Block\_shared$	Size of threads, registers, and shared memory per block	Source code resource usage
$Interleave$	Average cycles a warp waits for scheduling	Source code resource usage
MBT, MBR, MBS, MBB	Active blocks restricted by threads, registers, shared memory, and blocks	Hardware resource constraints
$Active\_blocks$, $Active\_warps$	Active blocks and warps per SM	Occupancy
$Block\_iters$	Iterations each SM has to execute active blocks	Block partitions
B_{comp}, B_{mem}, B_{pipe}, B_{ilp}	Bottlenecks of compute efficiency, memory efficiency, instruction level parallelism and pipeline	Advisor

memory, either global memory and registers or shared memory and registers. We analyze global memory and shared memory accesses separately.

Shared memory has multiple banks, where the maximum bandwidth is bounded by the width per bank ($Bank\_width$). If the width of an instruction (Ins_width_i) is greater than $Bank\_width$, the instruction has to be issued multiple times. Besides, on shared memory, each 4-byte (or 8-byte) word is stored in adjacent banks. If multiple threads access different addresses in the same bank, it causes bank conflicts. Specifically, if there are N_i threads in a warp conflict, we call it an N_i-way conflict in which the instruction is issued N_i times. Hence, the efficiency of shared memory is reduced to $1/N_i$ as shown by Equation 5.

$$E_{shared} = \frac{1}{N_i \times \lceil \frac{Dispatches_{ldst} \times Warp\_size}{Units_{ldst}} \rceil \times \lceil \frac{Ins\_width_i}{Bank\_width} \rceil} \tag{5}$$

Global memory is accessed via 32, 64, or 128 bytes memory transactions. If a warp accesses global memory in a coalesced mode, only a single transaction is needed. Whereas if the warp accesses global memory in a stridden manner, or if the required bytes is greater than 128, multiple transactions are required. Let N_i be the number of transactions needed to access global memory, we derive Equation 6 to represent the efficiency of global memory.

$$E_{global} = \frac{1}{N_i \times \lceil \frac{Dispatches_{ldst} \times Warp\_size}{Units_{ldst}} \rceil} \tag{6}$$

In addition, we observe from our benchmark that if data are loaded from the read-only cache, the efficiency is higher than that calculated by Equation 6. So using the read-only cache results in higher throughput than loading data from global memory.

Algorithm 2 Instruction dependency

```
1:  Input
2:  asmSet : a set of asm structures
3:  ▷ asm − ⟨name, control, modifier, dest, source1, source2..⟩
4:  Output
5:  depSet : a list of dep structures
6:  ▷ dep − ⟨asm_from, asm_to, latency⟩
7:  depSet ← 0
8:  regAsm ← 0
9:  for i ← 0, asmSet.size do
10:     to ← asmSet[i]
11:     for all source ∈ to.sources do
12:         from ← regAsm[source]
13:         if typeof(from.name) == global then
14:             lat ← cachehit(from)                ▷ Equation 7
15:         else
16:             lat ← latency[from]
17:         end if
18:         depSet.push(⟨from, to, lat⟩)
19:     end for
20:     regAsm[to.dest] ← to
21: end for
```

3.1.3 Latency and Dependency. Algorithm 2 presents an iterative procedure to extract instruction dependencies and associates them with latencies. In each iteration, it reads previous instructions that have modified the current instruction's *source* registers at Line 12 and updates the *dest* register which the current instruction modifies at Line 20. Our benchmarks measure the latencies at Line 16, using the toolchain mentioned in Figure 1. We list some instructions' latencies on Kepler K20m in Table 2.

Unlike other instructions, we take cache miss into account for global memory instructions (Line 14). If data are hit in the cache, the latency is lower than accessing data from DRAM. We use $Latency_{amat}$ to denote the average latency of memory instructions.

$$Latency_{amat} = Latency_{global} \times Miss\_ratio + Latency_{hit} \tag{7}$$

In Equation 7, $Miss\_ratio$ is hard to obtain statically. With the increment of $Miss\_ratio$, $Latency_{global}$ determines the upper bound of $Latency_{amat}$. Thus, we consider the $Miss\_ratio$ as 1 by default and allow programmers to modify the value by themselves. NVIDIA employs L1, L2, and read-only caches on GPUs. For brevity, we omit to expand $Latency_{hit}$ on every cache level.

Table 2: Instruction latencies

Instruction	Type	Units	Latency
IMAD	compute	SP(INTU)	9 cycles
IMUL	compute	SP(INTU)	9 cycles
IADD	compute	SP(INTU)	9 cycles
FFMA	compute	SP(FPU)	9 cycles
RCP	compute	SFU	9 cycles
STS.32	shared	LDST	9 cycles
LD.32	global	LDST	190 cycles

3.2 DAG Constructor

Algorithm 3 DAG constructor

```
1: Input
2: insSet : a set of ins structures
3: ▷ ins − ⟨asm1, asm2, ...⟩
4: depSet : a set of dep structures
5: ▷ dep − ⟨asm_from, asm_to, latency⟩
6: effMap : a map of asm → efficiency
7: Output
8: dag : a dag linked by dispatch orders and dependencies
9: C_w : the length of the critical path of dag
10: C_w ← 0
11: dag ← 0
12: for i ← 1, insSet.size do
13:     for all from ∈ insSet[i − 1] do
14:         for all to ∈ insSet[i] do
15:             dag[from][to] ← 1/effMap[from]
16:         end for
17:     end for
18: end for
19: for i ← 0, depSet.size do
20:     from ← depSet[i].asm_from
21:     to ← depSet[i].asm_to
22:     dag[from][to] ← 1/effMap[from] + dep[i].latency
23: end for
24: for i ← 0, insSet.size do
25:     for all from ∈ insSet[i] do
26:         for all to ∈ dag[from][to] ≠ 0 do
27:             dag[to] ← Max{dag[to], dag[from][to] + dag[from]}
28:         end for
29:     end for
30: end for
31: for i ← 0, dag.size do
32:     C_w ← Max{C_w, dag[i]}
33: end for
```

3.2.1 Single-warp DAG. Algorithm 3 shows the steps to construct the instruction DAG for a single warp unit. Its inputs are two sets and a map: *insSet* that is composed of instructions by dispatch order, *depSet* that consists of instruction pairs that have dependencies, and $effMap$ that stores instruction efficiencies. It outputs the generated DAG and C_w that indicates the execution cycles.

The algorithm begins by presenting each instruction as a node in the graph, linking them with directed edges in the dispatching order, and assigning a weight for each edge (Line 12-18). Because a warp scheduler can issue two instructions simultaneously, we should loop all *asms*. It then associates instructions that have dependencies and adds a directed edge between them (Line 19-23), weighted with the sum of latency and the reciprocal of efficiency. Finally, it calculates every instruction's running cycles $dag[i], 0 \le i \le I-1$ (Line 24-30) and extracts their maximum value to be the length of the critical path (Line 31- 33).

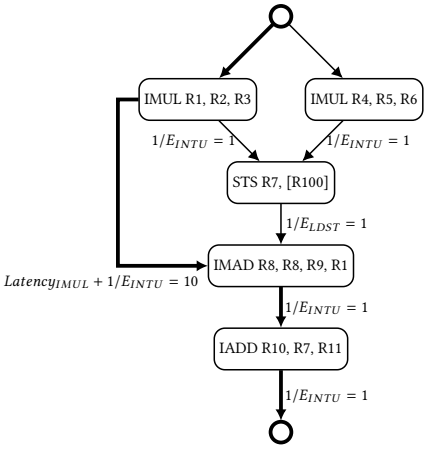


Figure 2: A DAG example

Figure 2 illustrates a DAG example. Among all the paths, the length of the critical path (bold lines in the figure) is 12 cycles, where the IMUL instruction's latency contributes 9 cycles.

3.2.2 Multi-warp DAG. An SM consists of multiple warp schedulers that issue instructions concurrently. To represent a DAG for multiple schedulers, we expand $Dispatches_u$ as the sum of $Dispatches_u^s$ from each warp scheduler.

$$Dispatches_u = \Sigma_{s=0}^{S-1} Dispatches_u^s \quad (8)$$

If warp schedulers conflict on shared resources, some of them will wait until the resources are free. Consider the following example in Figure 3. Suppose there are four warp schedulers with eight dispatches, but every two schedulers share $3 \times Warp\_size$ SPs. Then, two schedulers will be idle if all dispatches tend to issue IMUL instructions. Since only four dispatches are utilized, $E_{SP} = 1/2$ by Equation 4. So two cycles are needed to issue eight IMULs.

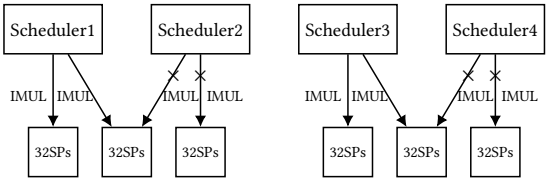


Figure 3: Resource conflicts by warp parallelism

With the increment of active warps, especially when it exceeds the number of schedulers, it is hard to predict the execution because of warp scheduling. We generate the average scheduling interval for prediction. Let *Interleave* be the division of *Active_warps* by *Schedulers*, we obtain the running time of a block by multiplying

Interleave with C_w (Equation 10). C_b might be longer than the actual running time since warp scheduling could hide some latencies. So C_b is considered to be an upper bound estimation. But since we focus on compute-intensive applications like DNN computations, where instruction scheduling can hide most of the latencies, C_b will be close to the actual value. We provide an analysis of the effect of hiding latencies in Section 3.3.

$$Interleave = \frac{Active\_warps}{Schedulers} \tag{9}$$

$$C_b = C_w \times Interleave \tag{10}$$

3.2.3 Occupancy and Blocks. An SM can run multiple blocks concurrently, depending on both hardware constraints and kernel resource usage. NVIDIA presents the concept of *occupancy* to describe resource utilization rate. We use Equation 11-15 to demonstrate how the maximum number of blocks is constrained by the maximum size of threads (MBT), registers (MBR), shared memory (MBS), and hardware specified block number (MBB).

$$MBT = \lfloor \frac{SM\_threads}{Block\_threads} \rfloor \tag{11}$$

$$MBR = \lfloor \frac{SM\_registers}{Block\_registers} \rfloor \tag{12}$$

$$MBM = \lfloor \frac{SM\_shared}{Block\_shared} \rfloor \tag{13}$$

$$Active\_blocks = min\{MBT, MBR, MBM, MBB\} \tag{14}$$

$$Active\_warps = Active\_blocks \times \lceil \frac{Block\_threads}{Warp\_size} \rceil \tag{15}$$

For a GPU that has NSM SMs, it handles $NSM \times Active\_blocks$ blocks concurrently. Assuming that workloads are distributed evenly among all SMs, we derive the following equations to estimate the running cycles C_k of a computation kernel:

$$Block\_iters = \lceil \frac{Blocks}{Active\_blocks \times NSM} \rceil \tag{16}$$

$$C_k = C_b \times Block\_iters \tag{17}$$

In Equation 16, *Blocks* is decided by algorithm variables and kernel partitions. If the remainder of *Blocks* and $Active\_blocks \times NSM$ is large, it causes the long tail phenomenon that puts a cap on the performance.

3.3 Performance Advisor

We propose four potential bottlenecks to quantify the performance from different prospectives. Specifically, we analyze B_{ilp}, B_{comp}, B_{mem}, and B_{pipe} of which the best values are 0 and worst values are 1. Programmers could mitigate these bottlenecks step by step to promote the performance. We elaborate details of these metrics in the following.

B_{ilp} exhibits the bottleneck of instruction level parallelism, indicating whether different types of instructions are properly dual-issued or not. C_{w_u} is the length of the critical path in a graph that extracts E_u from the DAG in Section 3.2. The longest path among all C_{w_u} is considered to be the running cycles with the maximum parallelism level. Then, it compares the value to the length of the critical path without latencies ($C_{w_{eff}}$) to generate B_{ilp}.

$$B_{ilp} = 1 - \frac{Max_{u=0}^{|U|-1} C_{w_u}}{C_{w_{eff}}} \tag{18}$$

B_{comp} presents the bottlenecks of compute instructions that use SFUs or SPs, by dividing the utilized units to the total amount of units. Programmers could use all schedulers and dual-issue instructions of the same type to decrease B_{comp}.

$$B_{comp} = 1 - \frac{\sum_{i=0}^{I_{comp}-1} E_{u_i} \times Dispatches_{u_i} \times Warp\_size}{\sum_{i=0}^{I_{comp}-1} Units_{u_i}} \tag{19}$$

B_{mem} is devised for memory instructions that use LDST units. It compares the achieved bandwidth with the maximum bandwidth (Max_width_i) for an instruction. For instance, *LDS.32* that reads 32-bit for each thread from shared memory has the same efficiency as *LDS.64*, while two-folds *LDS.32* are needed to load the same amount of data. So B_{mem} of using *LDS.32* is higher than that of using *LDS.64*. For simplicity, we combine the presentation of global memory and shared memory bottlenecks, but we adopt different techniques to eliminate them. On shared memory, we choose instructions that match bank width and avoid bank conflicts. On global memory, we coalesce memory accesses and put the read-only cache into use.

$$B_{mem} = 1 - \frac{\sum_{i=0}^{I_{mem}-1} E_{u_i} \times Inst\_width_i \times Warp\_size}{\sum_{i=0}^{I_{mem}-1} Max\_width_i} \tag{20}$$

B_{pipe} shows the performance potential by hiding latencies. It sums up all latencies in the critical path, dividing the result by the running cycles without latencies ($C_{w_{eff}}$). Programmers could either reorder instructions in the DAG or increase the number of $Active\_warps$ ($Active\_blocks$) to cut down the bottleneck.

$$B_{pipe} = \frac{\sum_{i=0}^{I_{lat}-1} Latency_i}{C_{w_{eff}} \times Interleave} \tag{21}$$

If a given code snippet is highly optimized, all metrics will be close to 0. Comparing with traditional models, ours explains how the performance relates with assembly instructions and hardware features, and what optimization strategies have effects.

4 APPLYING TO DNN

In this section, we apply our framework to optimize two important DNN computations: GEMM and convolution. Starting from a baseline implementation, we demonstrate the effectiveness of optimization strategies by showing the process of mitigating bottlenecks step by step and promoting the performance. To evaluate the optimization effects, we compare the achieved performance with the predicted performance upper bound. Also, we compare our results with the state-of-the-art implementations, including cuBLAS-8.0, cuDNN-5.0, and cuDNN-6.0.

All experiments were run on Kepler K20m with computability 3.5. It has 13 SMs, and each of them has four warp schedulers, 192 SPs, 64 DPs, 32 LDSTs, and 32 SFUs. A warp scheduler can dispatch two instructions in a cycle. We adopt *nvprof* to investigate run-time metrics and present their median numbers in figures.

4.1 GEMM

The baseline GEMM implementation is shown in Algorithm 4, where A is a $K \times M$ matrix, B is a $K \times N$ matrix, and C is a $M \times N$ matrix. It first uses blocking at the shared memory level. Each block reads $\langle b_k, b_m \rangle$ elements from A to sm_A and $\langle b_k, b_n \rangle$ elements from

B to sm_B and computes a $\langle b_m, b_n \rangle$ sub-matrix in the result matrix C. sm'_A and sm'_B are switched with sm_A and sm_B at the end of every iteration, hiding global memory latency. It also adopts blocking at the register level. Every thread reads r_m elements from sm_A and r_n elements from sm_B, accumulating a $\langle r_m, r_n \rangle$ matrix along the K dimension.

Algorithm 4 Double buffering GEMM TN implementation ($\alpha = 1.0$ and $\beta = 0.0$): $C = C + A^T \times B$

```
1:  sm_A ← a b_k × b_m block of A on shared memory
2:  sm_B ← a b_k × b_n block of B on shared memory
3:  r_A ← a 1 × r_m row of sm_A on consecutive registers
4:  r_B ← a 1 × r_n row of sm_B on consecutive registers
5:  k ← 0
6:  do
7:      sm'_A ← a b_k × b_m block of A on shared memory
8:      sm'_B ← a b_k × b_n block of B on shared memory
9:      for i ← 0, b_k − 1 do
10:         accum ← accum + r_A^T × r_B
11:         r_A ← a row of sm_A
12:         r_B ← a row of sm_B
13:     end for
14:     sm_A ↔ sm'_A
15:     sm_B ↔ sm'_B
16:     k ← k + b_k
17:     sync
18: while k < K
19: accumulate accum with b_m × b_n block of C on global memory
```

4.1.1 Analysis. We focus on the most expensive phase of Algorithm 1 (Line 6-18). In this part, most of the operations are multiply-and-add instructions (Line 10) that accumulate along the K dimension and load instructions that read data from shared memory (Line 11-12).

$\boldsymbol{B_{ilp}}$: Because compute instructions and memory instructions use different function units, we can dual-issue load instructions (LDS) with multiply-and-add instructions (FFMA) to reduce B_{ilp}.

$\boldsymbol{B_{comp}}$: On K20m, four schedulers cannot utilize all 192 SPs with single-issued FFMAs. Whereas if all schedulers dual-issue FFMAs, resource conflicts happen. Therefore, we should design a proper pattern to utilize all 192 SPs without conflicts.

$\boldsymbol{B_{mem}}$: Apart from issuing LDS with FFMA, we have to choose appropriate shared memory instructions that fit bank width and avoid bank conflicts to lower B_{shared}. For sm'_A and sm'_B that are read from global memory, we decrease B_{global} by leveraging the read-only cache.

$\boldsymbol{B_{pipe}}$: By Equation 22, a larger register blocking size leads to higher arithmetic intensity.

$$\frac{2 \times r_m \times r_n}{4 \times (r_m + r_n)} \tag{22}$$

It indicates that the ratio of FFMA instructions to LDS instructions is increasing with the growth of r_m and r_n. In this way memory instruction latency is hidden, rendering low B_{pipe}. But increasing the number of registers per thread might reduce the number of active blocks (Equation 12), which in turn increases B_{pipe}. Hence, we should make a balance between parallelism and the blocking size.

4.1.2 Optimizations. **Register Blocking (RB)**: We can launch 128, 256, 384, or 512 threads per block to let threads evenly distributed on four warp schedulers. We choose 256 threads that allow 256 registers per block (maximum on Kepler). As mentioned before, we have to make a balance between parallelism and the blocking size. We choose the 12×12 blocking size, leading to very low $\sum_{i=0}^{I_{lat}-1} Latency_i$ in B_{pipe} (Equation 21). A 13×13 blocking size is not suitable for vector memory instructions, and a 14×14 blocking size is too large that the total amount of registers is greater than 256. Notice that some registers are already occupied for other purposes like offset calculation and double buffering. In this way $b_m = 192$ and $b_n = 192$.

Utilizing Bank Width (BW): K20m has 64-bit width shared memory banks. Based on Equation 20, B_{shared} of using *LDS.32* will be greater than *LDS.64*, as two-folds *LDS.32* are needed to load the same amount of data comparing with *LDS.64*. Using *LDS.128*, though matches 64-bit bank width, leading to bank conflicts. Thereby, we use *LDS.64* to load data from shared memory.

Dual-issue (DUAL): If we single-issue FFMAs, only $\frac{2}{3}$ SPs are utilized; if all schedulers dual-issue FFMAs, we encounter resource conflicts. Hence, we eliminate B_{comp} with a pattern that mixes dual-issued instructions and single-issued instructions. On K20m, a control instruction is followed by seven normal instructions. Assuming that the warp scheduling algorithm could choose the best strategy to fill instruction pipelines, all patterns that have half dual-issued FFMAs and half single-issued FFMAs can make full use of SPs and eliminate B_{comp}. We devise the "1-2-2-1" pattern. That is, in the first cycle the scheduler issues an FFMA, following four FFMAs in the next two cycles, and the last FFMA is dual-issued with another instruction like *LDS*, which also reduces B_{ilp}. Comparing with other patterns, it consumes only four cycles and uses the operand collector mechanism [26] to avoid register bank conflicts.

Instruction Scheduling (IS): We schedule instructions carefully to cut down B_{pipe}. In the "1-2-2-1" pattern, we have an empty slot left to dual-issue an instruction with the last FFMA instruction. Thus, apart from the mentioned LDS instruction, we also insert other instructions, such as offsets calculation and texture barriers into these empty slots. We use the DAG specified in Section 3.2 to reorder instructions, aiming to obtain a shorter length of the critical path (C_w).

Read-only Cache (ROC): We adopt *LDG* instruction, which reads data from the read-only cache and has a higher throughput than *LD* instruction that loads data from global memory. To keep data consistency, We insert a texture barrier into the loop (Line 6-18), thus hiding at most 200 cycles DRAM latency on K20m (Table 2). We use 128-bit modifier–*LDG.128* with the highest throughput.

4.1.3 Optimization Steps. Figure 4 shows GEMM's optimization steps, in which we present B_{global} and B_{shared} separately.

We start from mitigating B_{shared} by leveraging *LDS.64* to fit shared memory bank width (**+BW**). B_{ilp} is also reduced due to the decrement of $C_{w_{eff}}$ in Equation 18. Then, we eliminate B_{comp} by dual-issuing FFMA (**+DUAL**), improving the performance by 50%. Since shared memory instructions are dual-issued with the last FFMA instruction in the "1-2-2-1" mode, the optimization lowers B_{ilp} significantly. Instruction scheduling (**+IS**) mitigates B_{pipe} by organizing instructions which are dual-issued with the last FFMA.

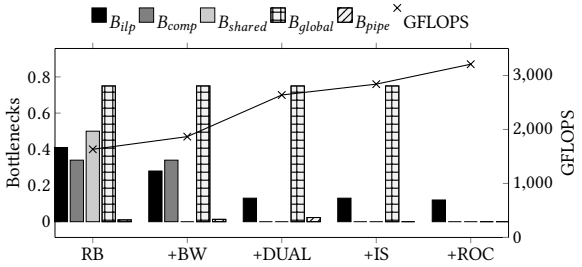


Figure 4: GEMM optimization steps

Last, we adopt *LDG.128* that reads data from the read-only cache (**+ROC**) to reduce B_{global}. We achieve 115% speedup with step-by-step optimizations finally.

4.1.4 Performance Prediction and Evaluation. Figure 5 shows the results of GEMM performance by square matrices $M = N = K$ from 384 to 5760. Model-predict is the estimated running time of the whole program. When the matrix size is small (less than 1000), the total number of blocks is small so that SMs are not fully utilized. Therefore, both cuBLAS and our ASM render low performance, and ASM performs worse than cuBLAS because cuBLAS automatically uses small size kernels to avoid the long tail phenomenon illustrated by Equation 16. With the increment of shape size, ASM beats cuBLAS by about 20%, utilizing about 88% floating-point resources. Our model could precisely predict the performance of ASM and guide optimizations. The average 2% predict error is caused by the neglect of register bank conflicts and synchronization effects.

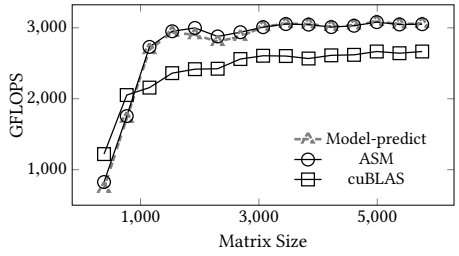


Figure 5: GEMM performance

We capture six metrics at run-time to reason the performance benefits compared with cuBLAS in Table 3. In GEMM's compute phase, two types of instruction dominate: LDS and FFMA. ASM achieves 951GB/s shared memory load throughput, three times of cuBLAS's 384GB/s, which illustrates the effectiveness of optimizing B_{mem}. ASM shows 86% floating point efficiency, while cuBLAS only has 72%; ASM also has 6.1 IPC (Instructions per cycle), but cuBLAS only has 5.3. These indicate that ASM has reduced B_{comp} and B_{ilp} by enabling the dual-issue mechanism. Moreover, we reorder instructions to hide instruction latencies. But as cuBLAS is also highly optimized, both implementations have very low memory instruction stall and execution instruction stall, indicating low B_{pipe}.

Table 3: Comparison between assembly GEMM and cuBLAS metrics (M=N=K=3840)

Metrics	cuBLAS	ASM
Shared memory throughput	384GB/s	951GB/s
Execution stall	2.6%	3.2%
Memory stall	0%	0%
IPC	5.3	6.1
Occupancy	12%	12%
SP efficiency	72%	86%

4.2 Convolution

Equation 2 indicates that the convolution process partitions on K (b_k) and N (b_n) dimensions and incorporates a similar calculation to GEMM, despite that it has to generate an *index* array to retrieve offsets. We use a single warp to calculate the *index* array and let others wait until it finishes. This benefits from using intra-warp bit operators like *VOTE* and *POPC* and avoiding communication across different warps. We also focus on the compute phase as it plays the most expensive part in most configurations.

4.2.1 Analysis. ***Pad_reduce***: Different from GEMM, the compute phase fetches data on the *CHW* dimension of the input using the *index* array (Equation 3). In some configurations (Table 5), we pad elements on the input. Because all the padding elements are zeros, we can avoid fetching them without loss of accuracy.

$$P = (H + 2 \times pad - R)/stride + 1 \tag{23}$$

$$Q = (W + 2 \times pad - S)/stride + 1 \tag{24}$$

$$Pad\_reduce = \frac{\sum_{i=0}^{P \times Q - 1} Actual\_comp_i}{2 \times P \times Q \times R \times S} \tag{25}$$

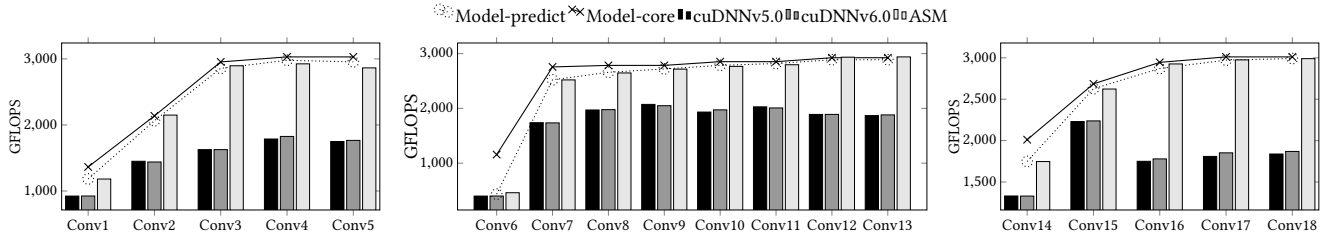
Equation 25 expresses an approximation of the number of computations. Intuitively, it equals to the number of computations without *pad* over the computations with *pad*. We estimate the running cycles as $C_k \times Pad\_reduce$.

$\boldsymbol{B_{ilp}, B_{comp}, B_{mem}, B_{pipe}}$: The compute phase of convolution is similar to GEMM in which the most instructions are FFMA and LDS. Hence, we use the "1-2-2-1" pattern (**DUAL**) to dual-issue LDS with FFMA, mitigating B_{comp} and B_{ilp}. Likewise, B_{mem} is eliminated with suitable bank width (**BW**) and the read-only cache (**ROC**). However, comparing with GEMM, it has to generate the offset of the *index* array and read data from it. Because of the growth of extra instructions, we cannot hide latencies only by instruction scheduling (**IS**). Under the circumstance, we add the number of active warps to diminish B_{pipe} (Equation 21).

4.2.2 Optimizations. As discussed before, we apply **DUAL**, **BW**, **IS**, and **ROC** in the same way as GEMM. Two different optimization steps are listed as following:

Register Blocking (RB): In GEMM, we use a 12×12 register blocking size, requiring $b_k = 192$ and $b_n = 192$. But it is too large for K and N as shown in Table 5 such that some threads remain idle in the compute phase. So we use a 8×8 register blocking size for $b_k = 128$ and $b_n = 128$ to fit for most configurations.

Register Reuse (RR): As mentioned in Section 4.2.1, we should increase the number of active warps to hide latencies. During the compute phase, we can carefully reuse some registers, rendering 128 registers per thread so that each SM holds two active blocks



(a) Alexnet performance **(b) VGG performance** **(c) Overfeat performance**

Figure 6: Convolution performance on different networks

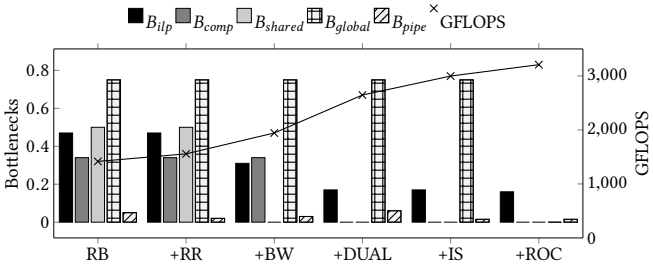


Figure 7: Convolution optimization steps

instead of one. In this way, we increase $Interleave$ in Equation 21 and thus reduce B_{pipe}.

4.2.3 Optimization Steps. We examine optimization effects in Figure 7. For the baseline kernel, B_{pipe} is high because of instruction latencies. So we reuse registers (**+RR**) to hide latencies. To lower B_{shared}, we adopt *LDS.64* (**+BW**) to utilize shared memory bank width. It also reduces B_{ilp} as the number of LDS instructions is diminished. B_{comp} and B_{ilp} are eliminated by dual-issuing FFMA and LDS instructions (**+DUAL**), which improves the performance by 50%. Interestingly, because the running cycles are reduced, B_{pipe} increases. We then lower B_{pipe} by reordering instructions (+**IS**). Finally, we use the read-only cache (+**ROC**) to achieve higher throughput, eliminating B_{global}.

4.2.4 Performance Prediction and Evaluation. Figure 6 displays the experimental results of three classical convolution networks in Table 5. Model-core is the predicted running time of the core loop which we optimized, which could be regarded as the prediction of performance upper bound. Our ASM achieves 83% floating point efficiency at maximum, and it outperforms cuDNN's GEMM based convolution in all layers, with up to 60%. Conv1, Conv2, and Conv14 show relatively low performance but close to Model-core, as their Ks are not a multiple of $b_k = 128$ so that some threads remain idle. The problem can be fixed by using smaller size kernels. Conv6 in VGG differs from the mentioned three layers, in which Model-core is greatly higher than ASM. Two reasons explain the phenomenon: (1) Its C, R, and S are small so that other phases, such as the generation of the *index* array, play a significant part. (2) $C \times R \times S$ is less than 8×8, the register blocking size, and thereby some iterations are not needed. The problems can be solved by reducing the number of blocks to increase each block's compute iterations. Our model can precisely predict most of the configurations by incorporating the Pad_reduce parameter in Equation 25.

Table 4: Comparison between assembly convolution and cuDNN metrics (Conv13)

Metrics	cuDNN	ASM
Shared memory throughput	492GB/s	1000GB/s
Execution stall	3.5%	0%
Memory stall	8.1%	0%
IPC	4.5	5.6
Occupancy	24%	24%
SP efficiency	59%	76%

To investigate micro-architecture level advantages over cuDNN, we also benchmark six metrics. Because cuDNN consists of several kernel calls, we only display the one takes the most time. Like the GEMM comparison, our ASM has a higher shared memory throughput, SP efficiency, and IPC, indicating that we have reduced B_{mem}, B_{comp} and B_{ilp}. More than that, ASM's instruction execution stall and memory stall are lower than cuDNN, which illustrates the effectiveness of eliminating B_{pipe}.

5 RELATED WORKS

This section reviews previous research about performance analysis on GPUs and accelerations on deep neural networks.

The performance model on GPU is an effective tool for directing programmers to tune their applications. [8] provided a model based on MWP (memory warp parallelism) and CWP (compute warp parallelism). But their work was built on PTX instructions, and they regarded shared memory instructions the same as compute instructions. [22] extended the MWP-CWP model. It analyzed applications in both dynamic and static ways, taking bank conflicts into account. However, they still used PTX instructions for performance prediction. [27] presented a quantitative study based on assembly instructions to inspect run-time program features. [14] and [5] built performance models for specific applications. None of the previous works have developed a thorough framework to analyze program bottlenecks and guide programmers choosing assembly instructions.

Since deep neural networks [13, 21, 23] have achieved significant successes in various applications, there has been an emergence of approaches to utilize efficient hardware [1–3] to accelerate different networks. Among these acceleration methods, GPUs have the highest compute efficiency and memory bandwidth. Hence,

DNN libraries [4, 9, 11, 18] for GPUs are widely used for developing neural networks. Recent research has dealt with architectural effects that impact DNN performance on GPUs. [17] investigated different convolution libraries by comparing run-time metrics like global memory efficiency and warp efficiency. [16] studied different storage formats of input data and provided automatic transform strategies to gain the highest performance. But they have not built models to associate algorithms with GPU features. Although vendor products like cuDNN [4] and cuBLAS [20] provide some efficient GPU implementations, their source code is not open source to meet the the growth of DNN computation patterns. It is necessary to provide developers a complete toolchain, helping them quickly instantiate fast kernels under specific architectures.

Table 5: Classic CNN networks

Layer	N	K	H&W	R&S	C	pad	stride
Alexnet							
Conv1	128	64	224	11	3	3	4
Conv2	128	192	27	5	64	2	1
Conv3	128	384	13	3	192	1	1
Conv4	128	256	13	3	384	1	1
Conv5	128	256	13	3	256	1	1
VGG							
Conv6	128	64	224	3	3	1	1
Conv7	128	128	112	3	64	1	1
Conv8	128	256	56	3	128	1	1
Conv9	128	256	56	3	256	1	1
Conv10	128	512	28	3	256	1	1
Conv11	128	512	28	3	512	1	1
Conv12	128	512	14	3	512	1	1
Conv13	128	512	14	3	512	1	1
Overfeat							
Conv14	128	96	231	11	3	0	4
Conv15	128	256	28	5	96	0	1
Conv16	128	512	12	3	256	1	1
Conv17	128	1024	12	3	512	1	1
Conv18	128	1024	12	3	1024	1	1

6 CONCLUSION

This paper presents a GPU performance analysis framework at the assembly level. It analyzes assembly instructions, generates a DAG to model instruction behavior, and incorporates occupancy and block partitions for accurate prediction. The framework also advises potential bottlenecks of compute efficiency, memory efficiency, instruction level parallelism and instruction pipeline to guide programmers for performance tuning. By mitigating these bottlenecks step by step, we succeed in promoting two vital DNN computations: GEMM and convolution. The limitation of the methodology is the availability of a tool to handle assembly instructions. We plan to extend the *instruction parser* for various GPU architectures.

7 ACKNOWLEDGEMENT

We would like to express our gratitude to all reviewer's constructive comments for helping us polishing this paper. This work is supported by The National Key Research and Development Program of China (2016YFB0200803, 2016YFB0200300, 2016YFB0200504, 2016YFB0201305), National 863 Foundation of China (2015AA01A301) and National Natural Science Foundation of China, under grant no. (61521092, 91430218, 31327901, 61472395, 61272134, 61432018).

REFERENCES

[1] Bryan Catanzaro. 2013. Deep learning with COTS HPC systems. (2013).
[2] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 247–257.
[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, Vol. 49. ACM, 269–284.
[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[5] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM Sigplan Notices*, Vol. 45. ACM, 115–126.
[6] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.
[7] Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation* 9, 3 (1990), 251–280.
[8] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 152–163.
[9] Bergstra James, Breuleux Olivier, Bastien Frédéric, Lamblin Pascal, and Pascanu Razvan. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.
[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.
[11] Alex Krizhevsky. 2014. cuda-convnet2. (2014).
[12] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).
[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[14] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 1–10.
[15] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.
[16] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 54.
[17] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance Analysis of GPU-Based Convolutional Neural Networks. In *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 67–76.
[18] NervanaSystems. 2016. Neon. https://github.com/NervanaSystems/neon. (2016).
[19] NVIDIA. 2016. NVIDIA Compute PTX : Parallel Thread Execution. (2016).
[20] CUDA Nvidia. 2015. Cublas library. *NVIDIA Corporation, Santa Clara, California* (2015).
[21] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2013. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229* (2013).
[22] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 11–22.
[23] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[24] Marc Gonzalez Tallada. 2016. Coarse Grain Parallelization of Deep Neural Networks. *SIGPLAN Not.* 51, 8, Article 1 (Feb. 2016), 12 pages. DOI: http://dx.doi.org/10.1145/3016078.2851158
[25] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014).
[26] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. (2017).
[27] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 382–393.
[28] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. 2016. ZNN – A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *IEEE International Parallel and Distributed Processing Symposium*. 801–811.