Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning

Xiuxia Zhang[1,2] Guangming Tan[1,2] Shuangbai Xue[1,2] Jiajia Li[3] Keren Zhou[1,2] Mingyu Chen[1,2]

[1] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2] University of Chinese Academy of Sciences
[3] Georgia Institute of Technology

{zhangxiuxia,tgm,xueshuangbai,cmy}@ict.ac.cn jiajiali@gatech.edu zhoukeren@ncic.ac.cn

Abstract

In this paper, we present a methodology to understand GPU microarchitectural features and improve performance for compute-intensive kernels. The methodology relies on a reverse engineering approach to crack the GPU ISA encodings in order to build a GPU assembler. An assembly microbenchmark suite correlates microarchitectural features with their performance factors to uncover instruction-level and memory hierarchy preferences. We use SGEMM as a running example to show the ways to achieve bare-metal performance tuning. The performance boost is achieved by tuning `FFMA` throughput by activating dual-issue, eliminating register bank conflicts, adding non-`FFMA` instructions with little penalty, and choosing proper width of global/shared load instructions. On NVIDIA Kepler K20m, we develop a faster SGEMM with 3.1Tflop/s performance and 88% efficiency; the performance is 15% higher than cuBLAS7.0. Applying these optimizations to convolution, the implementation gains 39%-62% performance improvement compared with cuDNN4.0. The toolchain is an attempt to automatically crack different GPU ISA encodings and build an assembler adaptively for the purpose of performance enhancements to applications on GPUs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming---parallel programming; D.3.2 [*Programming Languages*]: Language Classifications---assembly languages

General Terms Performance, Benchmarking

Keywords SGEMM, Assembler, GPU, Performance, Convolution, Reverse-engineering GPU ISA encoding

PPoPP '17, February 04–08, 2017, Austin, TX, USA.
©2017ACM. ISBN 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018755

1. Introduction

As GPUs provide higher peak floating-point performance and memory bandwidth than CPUs, researchers tend to adopt GPUs to accelerate compute-intensive data-parallel programs. Hardware vendors provide performance-critical libraries tuned on their specific processors, such as Intel MKL [14] and AMD ACML [3] for multicore x86 CPUs and NVIDIA cuBLAS [14] and AMD clMath [2] for GPUs. However, third-party implementations consistently outperform these vendor libraries. For example, OpenBLAS [30], based on the hand-tuned assembly, achieves better performance than AMD ACML on AMD Piledriver processors and Intel MKL on Intel Sandy Bridge processors. Although a comprehensive OpenBLAS-like library is not available for GPUs, several efforts [6, 11, 16, 25, 29] were invested to achieve better performance than cuBLAS's GEMM by tuning assembly on NVIDIA GPUs. However, these works leave **two open questions** to be addressed for diligent performance tuning on the microarchitecture of every GPU generation. Since single-precision general matrix multiply (SGEMM) is extensively used in scientific applications and deep learning related domains, we use SGEMM as a running example for a concise presentation throughout this paper.

- *A toolchain that can identify GPU microarchitectural features and guide performance tuning is lacking.* Unlike the general-purpose CPU community, where a series of toolchains is available to tune performance in a bare-metal way, only the CUDA model is encouraged in the GPU community. NVIDIA engineers hand tune their supported libraries in GPU assembly language; thus, this leaves other unsupported algorithms poorly optimized. Fortunately, researchers have made some initial progress on performance tuning tools, including benchmarking [18, 29, 31] and the design of assemblers on particular GPU architectures [5, 10, 12, 27]. In this paper, we develop a methodology to systematically identify mi-

croarchitectural features by automatically cracking ISA encoding, building a GPU assembler, and benchmarking.

- *There is a lack of comprehensive understanding of the computational kernels' performance behavior in terms of the low-level GPU microarchitecture.* Most SGEMM analyses are circumscribed at the CUDA or PTX level due to the shortage of GPU bare-metal tools. Therefore, these studies cannot directly diagnose compiler deficiencies or hardware defects. By observing the disassembled codes of CUDA SGEMM, we find out that control codes generated by the CUDA compiler NVCC are inefficient in exploiting the FP32 Fused Multiply Add (`FFMA`) dual-issue feature on a Kepler GPU (details in Section 4.2). This implies that the binary codes generated by NVCC cannot utilize GPU cores efficiently. In addition, the compiler deficiencies can lead to a biased estimation of the performance bound.

In this paper, we automate the ISA-to-binary mapping and build a Kepler GPU assembler. Due to the compatible syntax with disassembly generated by `cuobjdump` [21], we can use NVCC to compile CUDA codes to a cubin file and then disassemble it to generate assembly. This approach allows users to optimize any code segment based on the generated assembly instead of coding from scratch. With this assembler, a microbenchmark suite is designed to understand and analyze a plenty of GPU microarchitectural features such as instruction throughput, warp scheduling, and register bank distribution. These features are helpful to understand and optimize the performance of a computational program. More specifically, we make the following contributions:

- We propose a GPU ISA encoding solver to crack ISA encodings of diverse GPU microarchitectures automatically by feeding disassembly codes. A Kepler GPU assembler is developed to tune the assembly codes generated by the CUDA compiler directly.
- We design a microbenchmark suite to explore the undocumented microarchitecture features of NVIDIA GPUs, such as control codes regulating `FFMA` instruction dual-issue and register bank indices influencing instruction throughput. These features are necessary to understand and tune GPU programs.
- We implement both SGEMM and convolution kernels on an NVIDIA Kepler GPU by applying the microarchitecture-level optimizations. The optimized SGEMM achieves up to 88% of the machine's peak floating-point performance, which is 15% higher than cuBLAS7.0. The optimized convolution kernel gains 39%-62% performance improvement compared with cuDNN4.0.

Our methodology is suitable for other NVIDIA GPU architectures with minor adjustments of the instruction solver, assembler, and benchmarking. The experience of exploring bare-metal optimizations is helpful for compiler development and performance tuning [33].

The rest of this paper is organized as follows. Section 2 introduces the SGEMM algorithm on a GPU and the CUDA binary utilities. Section 3 presents the instruction solver algorithms and microbenchmarking insights. In Section 4, a series of microarchitectural optimizations are applied to SGEMM. We report the experimental results in Section 5. Section 6 discusses the generality and portability of our methodology for different GPUs and diverse algorithms. Section 7 summarizes the related work. Finally, Section 8 concludes this work.

2. Background

This section first highlights tunable factors that determine SGEMM performance on the GPU architecture; it then introduces some CUDA binary utilities related to SGEMM optimizations at the assembly level for self-containment.

2.1 Performance Factors of SGEMM

The state-of-the-art SGEMM implementations on GPUs [11, 16, 19, 25] use two-level blocking, register and shared memory blockings, to exploit data reuse through the GPU memory hierarchy. The operation of SGEMM is defined as $C = beta*C + alpha*AB$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ matrices, respectively, while $alpha$ and $beta$ are scalar constants.

Algorithm 1 shows the skeleton of the blocked SGEMM algorithm. Task partitioning is based on the result matrix C. Every $t_x \times t_y$ thread block is responsible for computing a $b_m \times b_n$ matrix block of C by reading a $b_m \times b_k$ block of A and a $b_k \times b_n$ block of B, where b_k is the unrolling factor. Therefore, A, B, and C are divided into $M * K$, $K * N$ and $M * N$ grids with $b_m \times b_k$, $b_k \times b_n$, and $b_m \times b_n$ blocks, where $M = \left\lfloor \frac{m+b_m-1}{b_m} \right\rfloor$, $K = \left\lfloor \frac{k+b_k-1}{b_k} \right\rfloor$, and $N = \left\lfloor \frac{n+b_n-1}{b_n} \right\rfloor$.

Algorithm 1 has $2 \times r_x \times r_y \times b_k$ floating-point operations (flops) on each thread for one iteration of the while loop. Table 1 estimates the data movement volume through registers, shared memory, and global memory correspondingly. These parameters demonstrate that SGEMM performance is determined by the hierarchical memory blocking and the unrolling factor of the inner for loop. Tuning these factors relies on GPU memory bandwidth and latency. According to previous work [19, 25], tuning proper parameters to overcome memory bandwidth limits is relatively easy with the estimates in Table 1. Performance tuning for memory latency is much more difficult since latency is closely coupled with particular microarchitectural features such as the instruction sequence and instruction types, which depends on a GPU toolchain and an assembler to identify.

2.2 CUDA Binary Utilities

Though no official GPU assembler is publicly available, NVIDIA provides a PTX assembly compiler `PTXAS` and

Algorithm 1 Blocked SGEMM algorithm of a $t_x \times t_y$ thread block. b_m, b_n, b_k are shared memory blocking sizes, r_x and r_y are register blocking sizes.

▷ Shared memory variables: sm_A, sm_B, sm'_A, sm'_B
▷ Registers variables: $accum[r_x \times r_y]$, $r_A[r_x]$, $r_B[r_y]$, $r'_A[r_x]$, $r'_B[r_y]$.
$sm_A \leftarrow$ a $b_m \times b_k$ block of A
$sm_B \leftarrow$ a $b_k \times b_n$ block of B
$r_A \leftarrow$ a column of sm_A
$r_B \leftarrow$ a row of sm_B
do
  sync
  ▷ Shared memory double buffering
  $sm'_A \leftarrow$ another $b_m \times b_k$ block of A
  $sm'_B \leftarrow$ another $b_k \times b_n$ block of B
  ▷ Thread-local computation
  for $i \leftarrow 0, \ldots b_k - 1$ **do** ▷ loop unrolling
    $r'_A \leftarrow$ a column of sm_A
    $r'_B \leftarrow$ a row of sm_B
    if i is even **then** ▷ Register double buffering
      $accum \leftarrow accum + r_A \circ r_B$ ▷ $\circ$: outer-product
    else
      $accum \leftarrow accum + r'_A \circ r'_B$
    end if
  end for
  $sm_A \leftrightarrow sm'_A$; $sm_B \leftrightarrow sm'_B$ ▷ swap shared memory pointers
while one more valid $b_k \times b_n$ block of B exists
store $accum$ with $b_m \times b_n$ block of C to global memory

Table 1: Data movement volume of each thread for one while loop iteration.

Data path	Volume (in words)
global⇒register (`LDG`)	$\frac{r_x \times r_y \times b_k}{b_m} + \frac{r_x \times r_y \times b_k}{b_n}$
register⇒shared (`STS`)	$\frac{r_x \times r_y \times b_k}{b_m} + \frac{r_x \times r_y \times b_k}{b_n}$
shared⇒register (`LDS`)	$r_x \times b_k + r_y \times b_k$

disassembly tools `cuobjdump` and `nvdisasm`, which make it possible to build a GPU third-party assembler. Both `cuobjdump` and `nvdisasm` disassemble *cubin* files to *sass* files, which are a human-readable disassembly for examining potential performance issues in CUDA programs. Every instruction in the disassembly is composed of an instruction address in hexadecimal, instruction content, and the 64-bit instruction encoding in hexadecimal. For example, an `IADD` instruction in disassembly is recorded as follows:

$$/*0048*/\ IADD\ R0,\ R2,\ R0; /*0x4800000000201c03*/ \quad (1)$$

Unfortunately, no tools are available to modify the assembly directly for performance tuning; the CUDA codes can only be refined and the newly generated disassembly is reexamine iteratively.

NVIDIA provides its pseudo-assembly language PTX and its compiler `PTXAS`, where PTX targets a stable machine-independent ISA that could span multiple GPU generations. Its machine-independent property is convenient for compiler design, but also brings several performance drawbacks. First, PTX uses pseudo-registers; programmers cannot control its register allocation, so register bank conflicts cannot be avoided. Second, PTX has no control code information which is used in Kepler and Maxwell GPUs to optimize warp scheduling. Thus, dual-issue on Kepler and warp scheduling on both Kepler and Maxwell optimizations cannot be controlled by the PTX programmer. Therefore, an assembler is required to manipulate the assembly. Although NVIDIA hasn't released their internal assembler and instruction encoding format, the disassembled codes in *sass* files and the pseudo-assembly references [22] provide clues to crack its instruction format in order to build our own assembler (details in Section 3). This paper uses disassembly generated from PTX as the input of the GPU ISA encoding solver.

3. Identifying Microarchitectural Features

We first illustrate the methodology to understand microarchitectural features and then explain the three major components: a GPU ISA encoding solver, an assembler, and benchmarks.

3.1 Methodology

We propose a methodology to understand GPU microarchitectural features and correlate them with microbenchmark performance. The workflow in Figure 1 consists of three major components: a GPU ISA encoding solver, a GPU assembler, and the microbenchmarks. The sample programs in Figure 1 are synthetic PTX files which generate specific instructions as the input of the instruction solver. Microbenchmarks are designed in assembly to figure out the correlation between microarchitecture and performance.

We write a simple generator to produce PTX instructions with various modifiers as the PTX samples. The generated PTX files are compiled to `cubin` files by `PTXAS`, and then disassembled by `cuobjdump` to generate native disassembly (around 2300 instruction variants). The ISA encoding solver takes the disassembly *sass* file as the input to decode each field of 64-bit instructions. A set of algorithms are designed to solve all the fields of a binary instruction. The fields include *operands*, *opcodes*, and *modifiers*. `cuobjdump` or `nvdisasm` is used to disassemble the binaries generated from the solver algorithms. Our assembler translates every instruction field to obtain 64-bit binaries and then encapsulates them with an ELF header to generate an executable cubin file. In the benchmarking workflow (dashed arrows in Figure 1), assembly microbenchmarks are tuned to explore GPU microarchitectural features such as register banking, instruction throughput, control codes, and load/store width.

3.2 GPU ISA Encoding Solver

The ISA encoding information that is necessary to build an assembler is not released by GPU vendors. Previous third-party work on building GPU assemblers [10, 12, 27] also involves cracking GPU ISA encodings, whereas no cracking process is explicitly reported. Our solver is proposed to make cracking GPU ISA encodings simple, automatic, and portable to future GPUs.

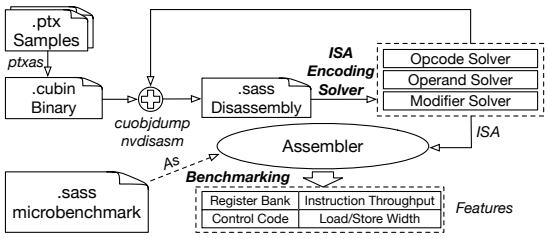


Figure 1: A schematic diagram in which GPU microarchitectural features are demystified by leveraging CUDA binary tools. The solid arrows represent the workflow of the instruction solver, while the dashed arrow represents benchmarking to determine the correlation between microarchitectures and performance.

Operands could be registers (e.g., `R5`), global memory (e.g., `[R6+0x20]`), constant memory (e.g., `C[0x2][0x40]`), shared memory (e.g., `[0x50]`), immediate values (e.g., `0x9` and `1.5`), or predicate registers (e.g., `P3`) [22]. We observe that an operand name always includes a number; thus, an operand can be inferred by its name. Since a number's encoding can be exactly explained, the operand encoding is relatively easy to recognize. For example, the register operand `R5` can be inferred from 101 in binary format, while the immediate value `0x9` is 1001. Opcodes and modifiers are mnemonic symbols, whose encodings cannot be literally explained; a heuristic algorithm is designed to crack their positions. In addition, modifiers are instruction-specific. Modifiers of the same type may have different encodings for different instructions. As an example, masks of the type-size modifier for instructions `LD` and `LDG` are in different binary positions. Hence, the modifier of every instruction needs to be processed independently.

3.2.1 Opcode and Modifier Solvers

We design a heuristic algorithm (Algorithm 2) to crack `opcode` encodings by detecting their bit positions. The opcode solver takes N disassembly instructions generated from PTX samples as the input and outputs all opcode bit positions as $opBits$. Each disassembly instruction $inst$ integrates its 64-bit binary code, name, operand types, and operand values as a structure. Take the `IADD` instruction in Equation 1 as an example; its binary code is $0x4800000000201c03$; the name is `IADD`; the types of operands are represented as a string "RRR"; and the values of operands are represented as a tuple $\{0, 2, 0\}$. By flipping the 64 bits one by one, we determine if each bit belongs to opcodes. Each flip generates a new binary code $\widehat{enc}$, which is disassembled by nvdisasm to a new instruction $\widehat{inst}$. If $\widehat{inst}$ is valid, and its instruction name is different from that of $inst$, this bit is in the opcode field.

From our results on a NVIDIA Tesla K20m, we find that the highest 10 bits and the lowest 2 bits represent opcodes of GPU Kepler architecture. The time complexity of the opcode solver is $O(64N)$, where N is the number of instructions in the disassembly file generated from PTX samples ($N \approx 2300$). This algorithm narrows the opcode enumeration space from 2^{64} to 2^{12}. We only enumerate possible opcodes in the pruned 12 bits and identify all the valid ones.

A modifier defines a specific behavior of an instruction. For example, `LD` instruction has type-size modifiers, such as `.u8`, `.s8`, `.u16`, `.32`, `.64` and `.128`, and cache operation modifiers, such as `.CS` (cache streaming) and `.CG` (cache at global level). We use a similar solver by flipping bit-by-bit to observe whether the bit controls an instruction modifier.

Algorithm 2 Opcode Solver

Input: instSet, N disassembly instructions generated from PTX samples
Onput: opBits, opcode positions in the 64-bit encoding
for $i \leftarrow 0, \ldots, N-1$ **do**
  ▷ $inst$ structure: {enc64, name, oprdType, oprd}
  $inst$ = instSet[i]
  ▷ the 64-bit encoding of the instruction
  $enc \leftarrow$ inst.enc64
  ▷ check each bit of enc for opcode
  for $j \leftarrow 0, \ldots, 63$ **do**
    ▷ flip the j^{th} bit of the encoding
    $\widehat{enc}$ = xorAt(enc, j)
    ▷ disassemble new encoding $\widehat{enc}$
    $\widehat{inst}$=`nvdisasm`($\widehat{enc}$)
    ▷ bit-j is in the opcode field if $\widehat{inst}$ is a different instruction.
    if isvalid($\widehat{inst}$) and inst.$name \neq \widehat{inst}.name$ **then**
      put(j, $opBits$)
    end if
  end for
end for
Return opBits

3.2.2 Operand Solver

Algorithm 3 shows our algorithm of cracking operand encodings. Since each instruction may have different numbers and types of operands, we identify operand positions as a tuple of its name and operand type. The operand types of an instruction are represented as a string. For example, oprdType of the instruction `IADD` R1, R2, C[0x0][0x40] is "RRC" and oprdType of `IADD` R1, R2, 0x8 is "RRI", where R, C, and I represent registers, constant memory and immediates, respectively. For each instruction with the known opcode, we flip each bit to obtain a new instruction $\widehat{inst}$. We compare $inst$ with $\widehat{inst}$ using a simple function "whichChange". This function confirms which operand value is changed by its position, if any; otherwise, it returns a negative integer. An entry in the visited dictionary is marked as 1 to avoid duplicated probing. The time complexity of Algorithm 3 is $O(52N)$, where N is the number of disassembly instructions with different opcodes.

3.3 Benchmarking

We design a microbenchmark suite to explore NVIDIA GPU microarchitectural features. As shown in Figure 1, we write GPU kernel codes in assembly and compile them into `cubin` by our assembler. The assembly instructions inside the kernel are unrolled without exceeding the L1 instruction cache

Algorithm 3 Operand Solver

Input: instSet, N disassembly instructions with different opcodes
Onput: pos, operand positions for each opcode, a 2D array.
visited={} ▷ A dictionary to record the visited information of an operand bit, key:⟨name, oprdType⟩
for $i \leftarrow 0, \ldots, N-1$ **do**
  ▷ $inst$ structure: {enc64, name, oprdType, oprd}
  inst = instSet[i]
  ▷ check the rest bits of enc for operand by excluding opcode bits
  for $j \leftarrow 2, \ldots, 53$ **do**
    ▷ flip j^{th} bit of enc
    $\widehat{enc}$ = xorAt(enc, j)
    ▷ disassemble new encoding $\widehat{enc}$
    $\widehat{inst}$=nvdisasm($\widehat{enc}$)
    ▷ If bit-j is an operand bit, return the operand position.
    p = whichChange($inst$,$\widehat{inst}$)
    if $p > 0$ **then**
      put(j, pos[⟨inst.name,inst.oprdType⟩][p])
      ▷ Mark ⟨name, oprdType⟩ as visited
      visited[⟨inst.name,inst.oprdType⟩] = 1
    end if
  end for
end for
Return pos

size to cover the branch overhead. The kernel code is tested hundreds of times, and the average time is used for accuracy. Timing ticks are recorded by the `S2R R0, SR_CLOCKLO` instruction, in which `S2R` moves the value from the special clock register `SR_CLOCKLO` to a general purpose register. Time is calculated in the general register, and then written into the global memory. Two metrics are benchmarked: latency and throughput. Latency is tested by a succession of *dependent* instructions by launching only one thread block with 32 threads. Throughput is tested by a succession of *independent* instructions by launching enough threads to make full use of the CUDA cores and memory bandwidth.

In this benchmarking process, we correlate microarchitectural features with the benchmark performance and obtain some meaningful observations in four microarchitectural features: register bank, control code, arithmetic throughput, and memory operation. These observations are useful for optimizing program snippets of real-world applications.

Observation 1–[Register Bank]*: Source register combinations may cause register bank conflicts that degrade instruction throughput.*

Shared memory bank conflicts are recognized as an important performance factor for CUDA programming. Recent research [16] has observed that register bank conflicts are also non-negligible. To probe register bank distribution, our microbenchmarks measure the instruction throughput for different combinations of `FFMA` register operands. Table 2 shows that different register combinations result in various levels of efficiency and throughput. The rightmost column represents the number of register bank conflicts recorded in our experiments. This experiment is conducted in single-issue mode by setting the control code to 0x20. Theoretically, without dual-issue, the peak efficiency is $4 \times 32/192 = 66.67\%$, in which 4 is the number of warp schedulers and 32 is the number of threads in a warp. In fact, we observe that the single-issue and dual-issue modes produce similar throughput behavior with bank conflicts. From our experiments on the Kepler architecture, we observe that:

- The destination register does not contribute to bank conflicts, so they are free to be assigned to any bank.
- When source registers have 2-way or 3-way register bank conflicts, the throughput of float instructions drops by 2.33% and 17.17% respectively, in single-issue mode.
- A proper register distribution is found through benchmarking to eliminate bank conflicts. Table 3 lists partial registers and their corresponding banks on NVIDIA Tesla K20m, which is consistent with the distribution on GTX680 [16]. Although they have different maximal numbers of registers per thread and instruction encodings.

Table 2: The efficiency of instruction throughput varies with different register combinations. Inst: instruction, Th/SM: instruction throughput per SM, Eff: throughput efficiency, Conf: register bank conflicts.

Inst	Th/SM	Eff	Conf
`FFMA R5,R4,R1,R0`	127.50	66.40%	0
`FFMA R2,R4,R1,R0`	127.50	66.40%	0
`FFMA R5,R2,R1,R0`	119.18	62.07%	2-way
`FFMA R3,R2,R1,R0`	119.18	62.07%	2-way
`FFMA R5,R9,R3,R1`	94.52	49.23%	3-way
`FFMA R11,R9,R3,R1`	94.52	49.23%	3-way
`FMUL R4,R1,R0`	127.50	66.40%	0
`FMUL R4,R2,R0`	119.17	62.06%	2-way

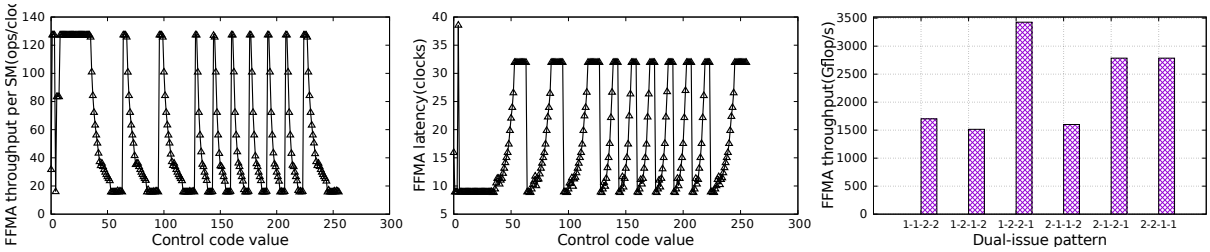
Table 3: Partial register index to register bank mapping.

Bank0	R0	R2	R8	R10	R16	R18	R24	R26
Bank1	R1	R3	R9	R11	R17	R19	R25	R27
Bank2	R4	R6	R12	R14	R20	R22	R28	R30
Bank3	R5	R7	R13	R15	R21	R23	R29	R31

Observation 2–[Control Code]*: Warp scheduling and issue mode are tunable by modifying control codes that regulate instruction issue.*

Starting with the Kepler architecture, NVIDIA has moved some control logics off the chip and into the kernel instructions to save power [10, 16]. This evolution provides programmers with the opportunity to make globally optimal scheduling decisions and other control optimizations if an assembler is available. The disassembly codes from sample programs indicate that a 64-bit binary control code controls 7 instructions is shown in Figure 7. We determine that the highest 6 bits and the lowest 2 bits are the *opcode* field of the scheduling instructions, and the middle 56 bits are used to control the execution of the following 7 instructions, each of which is assigned to an 8-bit control code.

We figure out the control code meanings by benchmarking the instruction sequences of different control codes. Bit 4, 5, and 7 represent shared memory, global memory, and the texture cache dependency barrier, respectively. We crack



(a) (b) (c)

Figure 2: Different control codes impact on performance(subfigure(c), 1→single-issue, 2→dual-issue).

the meanings of bits 0-3 by testing the latency of FFMA instructions. Figure 2(a) and Figure 2(b) show FFMA's throughput and latency when its control code varies from 0 to 255. After 0x20, both the throughput and latency show obvious periodicity. For each period, by increasing the value of bits 0-3, FFMA throughput drops, and its latency increases at different rates. This phenomenon implies that bits 0-3 indicate the number of stall cycles before issuing the next instruction. Our microbenchmarking reveals some specific patterns of control codes:

- When the control code is set to 0x00, the scheduler suspends a warp of instructions for 16 cycles.
- 0x2n means a warp is suspended for n cycles before issuing the next instruction, where $n = 0, 1, \ldots, 15$.
- 0x20 means the single-issue mode, while 0x04 means the dual-issue mode. When two consecutive instructions are controlled by 0x04 and 0x05, the throughput could reach its maximum.

Observation 3–[Arithmetic Throughput]*: With a proper control code pattern and register allocation,* FFMA *instruction throughput could approach its theoretical peak in dual-issue mode.*

It's very intricate to tune the efficiency of FFMA throughput on Kepler due to the dual-issue mode. Previous work [16] reports the maximal FFMA throughput per SM as 132, which is only 68.75% of the theoretical throughput. Our microbenchmarks reveal several key points of tuning FFMA throughput to 97% efficiency. First, the control codes must be set properly to dual issue adjacent instructions. Second, the ratio and interval of dual issued FFMA instructions must be tuned into a specific pattern. Third, the first instruction of the core loop needs to be aligned by 8 instructions (seven normal instructions plus one scheduling instruction). This restriction is caused by the control code pattern in the seven normal instruction sequences. Finally, each FFMA requires three source registers; thus, in dual-issue mode, two FFMAs require six source registers. However, a Kepler GPU only provides four register banks. The instruction order must be adjusted to use Kepler's operand collector mechanism [7, 26] to avoid register bank conflicts.

Figure 3 illustrates the mapping of FFMA instructions to CUDA cores in dual-issue mode on one SM. There are $32 \times 6 = 192$ cores on one SM; among them, 32 cores are shared by two warp schedulers, and four warp schedulers are available for each SM. In single-issue mode, each warp scheduler can issue one float instruction to 32 cores per cycle, which yields $4 \times 32/192 = 66.67\%$ float computation efficiency. In dual-issue mode, two warp schedulers must use the shared cores alternately to avoid resource conflicts. The jagged white and gray blocks in Figure 3 show a proper phase shift between the executing pace of two warps to get access to the shared computing units in turn. The theoretical optimal ratio of dual-issue to single-issue is $2:2$, $\binom{4}{2} = 6$ combinations for mixed single-issue and dual-issue pattern inside a 7-instruction scheduling block (Figure 2(c)). We choose the best 1-2-2-1 pattern for our SGEMM implementation. As shown in Table 4, these optimizations together improve FFMA throughput to 186 ops/cycle, which is very close to the theoretical peak 192 ops/cycle.

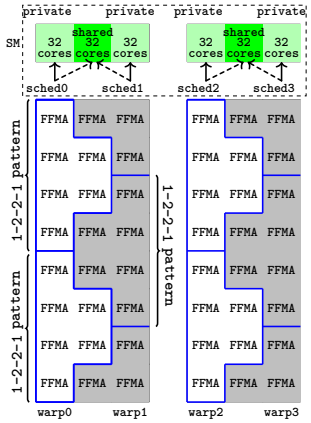


Figure 3: An illustration of FFMA dual-issue on one SM to achieve peak throughput

Table 4: Floating-point instruction throughput on Kepler

Inst	operation	single-issue	dual-issue
FFMA	c=a*b+c	127.52	186.35
FMUL	c=a*b	127.52	186.35
FADD	c=a+b	127.52	186.50

Observation 4–[Memory]*: For high memory bandwidth, shared memory prefers* 64*-bit load instruction* `LDS.64` *while global memory prefers texture cached* 128*-bit load instruction* `LDG.128`.

Regarding the GPU memory hierarchy, we focus on the programmer-controllable memory resources, shared memory and global memory. Different memory access widths (32-, 64-, or 128-bit) and paths (through L2 or texture cache) exist on NVIDIA GPUs. Intuitively, a wider load operation should achieve greater bandwidth. We test the bandwidth of shared memory instructions with different widths, i.e., `LDS.32`, `LDS.64`, and `LDS.128`. The instructions are arranged to avoid shared memory bank conflicts. Figure 4 compares the sequential memory access bandwidth of the three instructions by increasing the data volume. `LDS.64` achieves the highest bandwidth 137GB/s, which is about 76% of the peak shared memory bandwidth[1].

Two paths are used to load data from global memory, through the L2 cache by a `LD` instruction or the texture cache by a `LDG` instruction. We launch 26 thread blocks each with 512 threads, and specify that each thread accesses four words in a stride of $4 \times blockDim.x \times gridDim.x$. The total accessed global memory is 256MB. We benchmark that a `LDG.128` achieves 131GB/s, while `LD.128` only achieves 76GB/s.

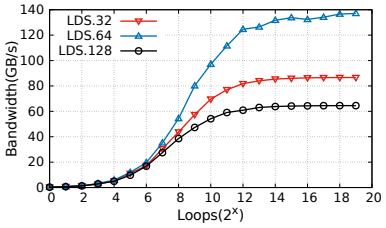


Figure 4: The bandwidth of `LDS.32`, `LDS.64`, and `LDS.128`

4. Optimizing SGEMM

The understanding of GPU microarchitectural features provides us with a larger tuning space for computational kernels. We apply a series of incremental optimizations to improve SGEMM performance on the Kepler architecture. The optimization strategies go through the architectural hierarchy from CUDA cores and registers to memory. All the optimization strategies are inspired by the observations from the benchmarking in Section 3.3.

- At the GPU core level, we promote `FFMA` instruction throughput to approach its peak through dual-issue by setting proper control codes.
- At the register level, we meticulously map operands to registers to avoid bank conflicts in the inner loop of Algorithm 1.
- At the memory level, we select the appropriate shared memory load/store width and global memory data path to achieve high bandwidth.

4.1 Register Allocation

To allocate registers for A column, B row, and C matrix block as in Algorithm 1, we have three objectives: correctness, no bank conflict, and tight register indices. `LDS.128` restricts four-word alignment for registers. Since the NVIDIA GPU does not have a 128-bit register, four consecutive 32-bit registers `RN`, `RN+1`, `RN+2`, and `RN+3` will be an equivalence for a 128-bit register, where N is a register index. We discover an undocumented restriction that N must be divisible by 4; otherwise, illegal instruction errors will be reported. The four-word alignment restriction for `LDS.128` simplifies hardware logic and cuts down power. Since we use `LDS.128` to load A and B, there are two bank allocation choices under this restriction and Kepler's bank distribution (Table 3). We use four colors to represent the four banks and show the register allocation when computing a 12×12 block of C in Figure 5. We assume allocating banks of A as $\begin{bmatrix}0\\1\end{bmatrix}$, banks of B as $\begin{bmatrix}2 & 3\end{bmatrix}$ as in Figure 5, and two choices remain for C, $\begin{bmatrix}1 & 2\\3 & 0\end{bmatrix}$ and $\begin{bmatrix}3 & 1\\0 & 2\end{bmatrix}$. The $2 \times 2 = 4$ bank patterns of SGEMM are equivalent in performance. Then, we arbitrarily choose $\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}2 & 3\end{bmatrix}$ $\begin{bmatrix}1 & 2\\3 & 0\end{bmatrix}$ for A, B, and C, respectively. For actual register index allocation, we choose continuous register indices to avoid exceeding 255. Every matrix block C_{ij}, column A_i, and row B_j have different banks in Figure 5; thus, register bank conflicts are completely avoided.

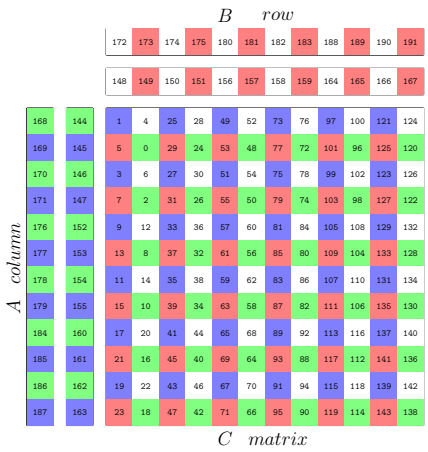


Figure 5: Register allocation in SGEMM. Every number in a cell is a register index. Different colors denote the mapping of register banks: green→bank0, blue→bank1, gray→bank2, red→bank3.

4.2 FFMA Dual-Issue

It is unrealistic to keep warp schedulers dual issuing the same kind of arithmetic instructions (e.g., `FFMA`) all the time.

[1] The theoretical shared memory bandwidth of an SM is calculated as $Bandwidth = f_{core} \times Width \times Warpsize$ in bytes, where f_{core} is the core frequency, $Width$ is the width of each shared memory bank.

0	4	3	71	68	67	72	76	75	143	140	139
1	2	5	69	79	66	73	74	77	141	142	138
11	8	7	60	64	63	83	80	79	132	136	135
9	10	6	61	62	65	81	82	78	133	134	137
12	16	15	59	56	55	84	88	87	131	128	127
13	14	17	57	58	54	85	86	89	129	130	126
23	20	19	48	52	51	95	92	91	120	124	123
21	23	18	49	50	53	93	94	90	121	122	125
24	28	27	47	44	43	96	100	99	119	116	115
25	26	29	45	46	42	97	98	101	117	118	114
35	32	31	36	40	39	107	104	103	108	112	111
33	34	30	37	38	41	105	106	102	109	110	113

Figure 6: FFMAs instruction scheduling to compute a 12×12 block of C. The numbers in cells denote the FFMA execution order. Dashed ellipses across two adjacent cells indicate that two adjacent FFMA instructions are dual issued in one clock cycle.

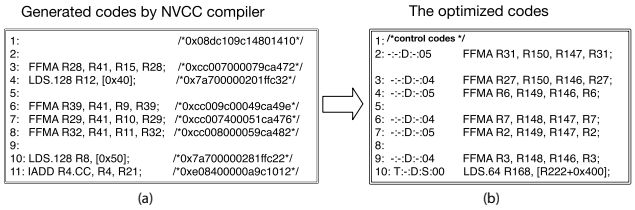


Figure 7: A comparison of compiler generated codes and our tuned assembly codes.

On a Kepler GPU, each warp is privately assigned 32 cores, and four warp schedulers consume 128 cores. The remaining $192 - 128 = 64$ cores are divided into two 32-core groups; each group of 32 cores is shared by two warp schedulers. Two warp schedulers need to negotiate for the extra 32 shared cores to avoid resource conflicts. As noted from *Observation 3* in Section 3.3, the best pattern of FFMA instructions block is a sequence of one single-issue (1 FFMA), two dual-issues (4 FFMAs), and another single-issue (1 FFMA). As shown in Figure 7, the instructions in lines 3-4 and lines 6-7 are dual issued separately. The other two instructions in line 8 and line 11 are two single-issues in terms of floating-point instruction execution. In contrast, most of the FFMAs are single issued in the CUDA compiler generated codes.

By extending the basic 7-instruction block (Section 3.3), we depict the scheduling pattern when computing a 12×12 block of matrix C in Figure 6. The FFMA to calculate C_{00} is issued first, then two FFMAs to compute C_{10} and C_{11} are simultaneously issued.

Another advantage of this execution order is less register pressure due to register reuse by facilitating the operand collector mechanism [7]. The operand collector allows operands to be cached and reused in subsequent instructions. The assembly code in Figure 7 lists the instructions calculating $C_{32}, C_{22}, C_{21}, C_{30}, C_{31}, C_{20}$ which correspond to the orders of $6, 7, 8, 9, 10, 11$ in Figure 5. With the elaborately designed computing order and register allocation, the reuse happens as follows. The FFMA in Line 3 uses cached operand R150 of Line 2, while Line 3 and Line 4 share R146. Thus, in the dual-issue mode, FFMAs of Line 3 and 4 need to read four registers R146, R27, R149, R6 instead of six. The corresponding banks of these registers are $0, 1, 3, 2$ based on Table 3, so no bank conflicts occur. Similarly, Line 7 uses the cached operand R149 from Line 4. In dual-issue mode, two FFMAs of Line 6 and Line 7 need to read 4 registers R148, R147, R7 and R2.

4.3 Schedule non-FFMA Instructions

After setting the order of FFMAs, non-FFMA instructions need to be inserted in proper positions to ensure correctness without losing performance. To tolerate instruction latency, the distance of dependent instructions needs to be larger than their latency. The distance is approximated as

$$distance = \frac{4 \times \#instructions}{7}. \quad (2)$$

A 7-instruction scheduling block costs four clock cycles in dual-issue mode. Therefore, given the distance L of two interleaved instructions, at least $\frac{L*7}{4}$ instructions are needed. In addition, the number of the rest slots to insert these non-FFMA instructions is estimated as

$$\#slots = \frac{r_x \times r_y \times b_k}{\#ffmas\_in\_schedule\_block} = \frac{12 \times 12 \times 4}{6} = 24 \times 4.$$

$r_x \times r_y \times b_k$ yields the total number of FFMAs for one thread inside the innermost loop in Algorithm 1, where r_x and r_y are register blocking sizes and b_k is the unrolling factor. $\#ffmas\_in\_schedule\_block$ is the number of FFMA instructions of one scheduling block, which is six by the 1-2-2-1 dual-issue pattern in Section 3.3. According to these principles, we first arrange LDS, STS, LDG because of their long latencies. The schedule slots are illustrated in a table. Note that we use double-buffering to hide the LDG latency from global memory, which is around 120 clock cycles. Every four loops require two LDGs to load data from global memory to registers, and four STSs to store data from registers to shared memory. A read after write (RAW) dependency exists between LDG and STS. From Equation 2, $\frac{120 \times 7}{4} = 210$ instructions are needed between them. We put LDG and STS in position $P[77][3]$ and $P[65][2]$, respectively, in Table 5. Thus, the in-between $143 - 77 + 144 \times 2 + 65 = 419$ (> 210) instructions are enough to hide the latency of LDGs.

The arrangement of LDS instructions, which load data from shared memory for double buffering A and B, follows the same approach with LDGs. The LDS latency is 28 clock cycles; thus, $\frac{28 \times 7}{4} = 49$ instructions are needed to interleave a LDS and an FFMA. In Table 5, LDS in $P[11][3]$ reads data from STS in $P[65][2]$; the distance between them is more than 28 cycles. At the end, a BAR.SYNC is inserted after STS but before LDS to make sure that data in shared memory is ready. Other instructions such as XOR, IADD, and ISETP are

inserted according to data dependency; they do not influence the performance because of their short latencies.

Table 5: The position table of `non-FFMA` instructions. The inner-loop is unrolled four times. The first column records slot numbers, and the first row represents iteration numbers.

unroll / slot	0	1	2	3
5	ISET P0	IADD A0		XOR smB
11	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
17	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
23	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
29	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
35	IADD K, -4	IADD A1	TEXDEPBAR	
41	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
47	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
53	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
59	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
65			STS.64 writeS	ISETP P2
71				
77		IADD B0		LDG A
83	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
89	ISETP P3			
95	LDS.64 smA	LDS.64 smA	LDS.64 smA	LDS.64 smA
101			STS.64 loadB0	LDG B
107			STS.64 loadB2	XOR writeS
113				
119	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
125			XOR smA	
131	LDS.64 smB	LDS.64 smB	LDS.64 smB	LDS.64 smB
137				
143		IADD B1	BAR.SYNC	BAR Loop

4.4 Memory Movement

According to our benchmarking observations, we use `LDG.128` to load data from texture cached global memory and `LDS.64` to load data from shared memory. Additional reasons to adopt them in SGEMM kernel are as follows: First, `LDG.128` reduces load instructions, and hence reduces `non-FFMA`s. In the inner loop of Algorithm 1, we need three `LDG.128`s instead of twelve `LDG.32`s to read twelve words from a column of A . Second, the shared memory transaction size of a warp is at most 256 bytes, which forces `LDS.128` memory requests to be split into multiple transactions. As we analyzed in Section 5, `LDS.128` has a lower bandwidth than `LDS.64`, which bounds the SGEMM performance.

5. Evaluation

In this section, we compare the performance of our optimized SGEMM with NVIDIA cuBLAS. Then, we present quantitative analysis on the effect of each optimization strategy and an estimation of the performance upper bound using a roofline model.

The experiments are conducted on an NVIDIA Tesla K20m GPU (refer to [20] for detailed specifications). We compare cuBLAS from CUDA 7.0 by testing both square matrices and rectangle matrices. The size of the square matrices varies from 384 to 12288 with a step 384. The smallest dimension of the rectangle matrices varies from 192 to 3072 with a step 192.

5.1 Overall Performance

We use $[M, K, N]$ to denote size of matrix A to be $M \times K$ and size of matrix B to be $K \times N$. Figure 8 shows the SGEMM performance of five different matrix shapes, $[W, W, W]$, $[W, 2W, 4W]$, $[W, 4W, W]$, $[4W, W, 4W]$, and $[4W, 2W, W]$, where the values of W are shown on the x-axis. The results demonstrate that our 12×12 and 8×8 register blocking implementations are better than cuBLAS in most cases. While 12×12 is the best one when the matrix size is greater than 1000, and 8×8 blocking is better when the matrix size is small. The performance of both cuBLAS and our SGEMM for the matrix shape $[W, 4W, W]$ fluctuates more seriously than other shapes because its parallelism is coarser, and each thread iteration is much longer ($K = 4W$). In contrast, $[4W, W, 4W]$, with the finest parallelism of the four shapes, has the smoothest results. When the matrix size is 12288×12288, the optimized SGEMM achieves 3.1 Tflop/s with 88% efficiency, while cuBLAS gets 2.7 Tflop/s with 76% efficiency. Our SGEMM achieves a $1.15\times$ performance speedup over cuBLAS. SGEMM performance increases with matrix size. One reason is that a larger matrix has a higher ratio of floating-point operations to the store operations of matrix C.Another reason is that a larger matrix has a better load balance on GPU because of the increase in the workload of each CUDA core. The number of thread blocks ranges from $2 \times 2, 8 \times 8, \ldots, 64 \times 64$ from left to right in Figure 8. Since Kepler has 13 SMs, matrix 384×384 suffers more from load imbalance because there are only 4 thread blocks. This explains the significant performance improvement when the size grows from 384 to 1536. With respect to the performance improvement over cuBLAS, our optimizations benefit more from larger matrices. The higher arithmetic intensity of larger matrices makes their performance increasingly bounded by the GPU microarchitecture rather than its memory hierarchy. Therefore, our microarchitecture-level optimization plays an important role in tuning SGEMM performance.

5.2 Performance Analysis

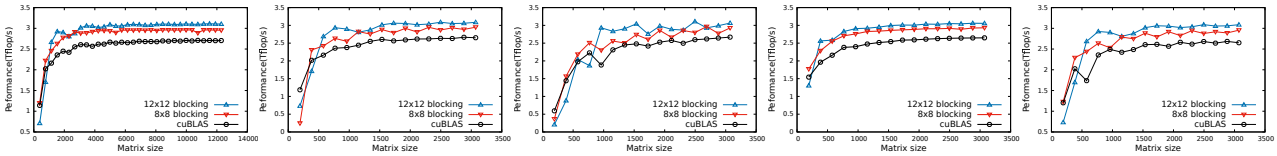
5.2.1 Influence of Register Blocking Sizes

Table 1 summarizes the data movement volume of the blocked SGEMM algorithm. The volume of data moving from shared memory to register is $r_x \times b_k + r_y \times b_k$, and the floating-point computation volume is $r_x \times r_y \times b_k$. The shared memory arithmetic intensity (sAI) is defined as the ratio between them:

$$sAI = \frac{r_x \times r_y \times b_k}{r_x \times b_k + r_y \times b_k} = \frac{1}{\frac{1}{r_x} + \frac{1}{r_y}}. \quad (3)$$

Register blocking sizes r_x and r_y are limited by the register counts per thread. Each thread needs $r_x \times r_y$, $2 \times r_x$ and $2 \times r_y$ registers to store the result of a matrix block of C, two columns of A (double-buffering), and two rows of B (double-buffering), respectively, in one single iteration of the innermost loop in Algorithm 1. Since the total number of registers must be less than the maximum register counts per thread (256 on Kepler), we have

$$r_x \times r_y + 2r_x + 2r_y < 256. \quad (4)$$



(a) [W,W,W] (b) [W, 2W, 4W] (c) [W, 4W, W] (d) [4W, W, 4W] (e) [4W, 2W, W]

Figure 8: Performance comparison of cuBLAS and the optimized SGEMM on different matrices

Equation 3 implies that larger register blocking yields a higher shared memory arithmetic intensity, and the optimal solution lies at $r_x = r_y$ with restriction by Equation 4. We use 8×8 and 12×12 blockings as a case study. In Figure 8, 12×12 blocking has better performance than 8×8 blocking because 12×12 blocking has higher sAI and instruction level parallelism. With respect to instruction scheduling optimization in Table 5, it has more slots to insert `non-FFMA` instructions, and latency can be better hidden. $12 * 12 + 4 * 12 = 192$ registers are used by `LD` and `FFMA` instructions, and a total of 236 registers with other address indices. The register counts per SM restricts us to launch up to 256 threads/block, which is in our implementation. Each thread block computes a matrix block $C_{192,192}$ by multiplying $A_{192,4}$ and $B_{4,192}$, where 4 is the unrolling factor.

Only one thread block per SM is active due to register limitation; thus, the thread occupancy is $256/2048 = 12.5\%$. With higher instruction level parallelism by using larger register blocking, the thread parallelism becomes low. However, the high performance of our SGEMM confirms that instruction level parallelism also plays an important role in GPU performance. A similar conclusion is offered by Volkov in [28].

5.2.2 Profiling Microarchitectural Optimizations

To examine the performance gains of different optimization strategies, we construct several intermediate implementations by incrementally applying our microarchitectural optimizations.

Baseline: The baseline has adopted conventional optimizations including register blocking, global and shared memory double bufferings, and unrolling, although without the assembly level optimizations. For example, the baseline uses default 32-bit `LD` rather than 128-bit `LDG` instructions to load data from global memory. Registers are allocated first for C from 0 to 143, then for A and B. In this case, `FFMA`s have $368/(144 * 4) = 63.89\%$ 2-way bank conflicts and $64/(144 * 4) = 11.11\%$ 3-way bank conflicts. In addition, the baseline cannot apply dual-issue optimization either.

+Reg: The register allocation pattern described in Section 4.1 is applied to eliminate register bank conflicts.

+LD128: Use wide global load instruction `LD.128` with L2-cached. Although Kepler has an L1 data cache, it is designed for local rather than global memory access [20].

+LDG128: Use the faster texture cached `LDG` instead of L2-cached `LD` along with `TEXDEPBAR` before data access due to the weak consistency of the texture cache [17].

+Dual: Use the dual-issue control fully enabled by utilizing the pattern described in Section 3. In dual-issue mode, `NOP` may be inserted for 7-instruction alignment.

Figure 9 shows the performance gains of each optimization strategy. As long as the number of instructions changes, instruction rescheduling is necessary to achieve good performance. Compared with the baseline implementation, SGEMM can achieve a $2.6\times$ speedup by applying all the optimizations. The elimination of register bank conflicts leads to performance improvement of approximately 10%. Wide load instruction `LD.128` contributes 27%-35% to performance, and texture-cached load instruction `LDG.128` leads to an improvement of 5%-12%. Dual-issue achieves the highest performance improvement with 84%-106%.

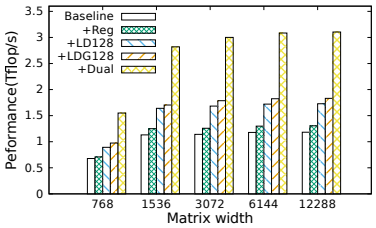


Figure 9: Evaluation of incremental optimizations.

5.2.3 Bounding Factor Analysis

We estimate the upper bound factors: shared memory, global memory, and computation power. In the innermost loop of Algorithm 1, each thread block computes $b_m \times b_n \times b_k$ operations and reads $b_m \times b_k + b_n \times b_k$ words. The upper bound of SGEMM by global memory bandwidth can be modeled as:

$$\frac{2 \times b_m \times b_n \times b_k}{4(b_m \times b_k + b_n \times b_k)} \geq \frac{PeakFlops}{Bandwidth_{global}} \quad (5)$$

According to the parameters in our SGEMM implementation, $b_m = b_n = 192$, $b_k = 4$, $PeakFlops = 3520$Gflop/s, so 73GB/s is the minimum requirement for the global memory bandwidth to achieve the theoretical peak of 3520Gflop/s. Inside each loop, $(r_x \times b_k + r_y \times b_k) \times t_x \times t_y$ words are read from shared memory, where t_x and t_y are thread block sizes, and r_x and r_y are register blocking sizes. The computation flops is $b_m \times b_n \times b_k$; thus, the ratio of computation to shared

memory access is

$$\frac{2 \times b_m \times b_k \times b_n}{4t_x \times t_y \times (r_x \times b_k + r_y \times b_k)} \geq \frac{PeakFlops}{Bandwidth_{shared}} \quad (6)$$

The minimum shared memory bandwidth requirement of SGEMM is 1173GB/s on the Tesla K20m, which is $1173/13 = 90$GB/s for each SM. Theoretically, hardware provides 200GB/s of global memory bandwidth and 2349GB/s of shared memory bandwidth, both of which are higher than the requirements; hence, neither of them are the bottleneck.

SGEMM has a different computation to memory access ratio in terms of global memory and shared memory, so we demonstrate two roofline models in Figure 10. The slant line in Figure 10(a) represents attainable performance with different arithmetic intensities when bounded by global bandwidth. The x-axis represents the flops:bytes ratios, which are 48 for 192×192 and 32 for 128×128 shared memory blocking of SGEMM based on Equation 5. Horizontal lines show the machine's peak performance and the performance of our optimized SGEMM with or without dual-issue. This figure demonstrates that SGEMM is not bounded by global memory bandwidth for either 128×128 or 192×192 shared memory blockings. The slant line in Figure 10(b) represents attainable performance with different arithmetic intensities when bounded by shared memory bandwidth. The x-axis represents the flops:bytes ratios, which are 3 for 12×12 and 2 for 8×8 register blocking of SGEMM based on Equation 6. This figure shows SGEMM is not bounded by shared memory through our `LDS` optimization. However, if we use `LDS.128` instead of `LDS.64`, SGEMM will be bounded by shared memory even for 12×12 blocking.

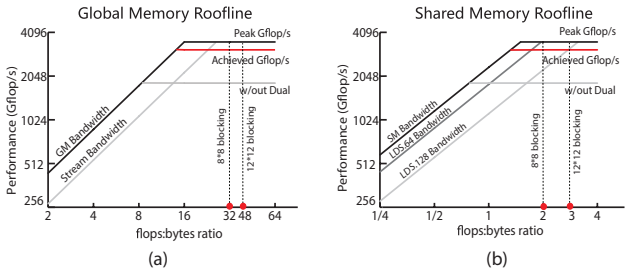


Figure 10: Memory roofline model using log-log scale. "GM/SM Bandwidth" indicates global/shared memory's theoretical bandwidth.

The loss of peak performance can be explained as follows. As we have shown in Section 3, `FFMA` throughput can achieve 97.67% efficiency. The loss is about 2.33%, which may be the result of overhead of the warp scheduler in `FFMA` dual-issue mode. The double-buffering algorithm can amortize the latency of `LDS`. With 12×12 register blocking and unrolling of 4, there are $144 * 4 = 576$ `FFMA`s in the loop. With our designed `FFMA` dual-issue pattern, every six `FFMA`s need four clock cycles in the pipeline. The innermost loop needs $4 \times 144 \times 4/6 = 384$ cycles for each thread, and two `LDG.128`s are also needed. We observe that each `LDG` has 10 cycles of penalty; the total `LDG` causes a $2 \times 10/384 = 5.2\%$ loss. Other penalties may be caused by synchronization and writing C matrix in the block.

6. Generality

Although this paper examines the methodology on GPU Kepler architecture and shows performance optimizations for SGEMM, we argue that the developed toolchains can be easily extended to other NVIDIA GPUs and that the explored optimization strategies are applicable to other floating-point computation-intensive applications.

Portability of Toolchains: For a particular GPU architecture, users only need to regenerate disassembly codes from the PTX samples and microbenchmarks with CUDA binary utilities and then feed them to the ISA encoding solver. The opcode, modifier, and operand solvers are portable among GPU architectures. We have validated the solver functionalities on the Fermi, Kepler, and Maxwell GPUs [33]. A corresponding assembler can be obtained by modifying the instruction grammar definition decoded by the solver, which could be done automatically with an assembler template [4].

Generality of Optimization Methods: The optimization strategies include FFMA dual-issue, register allocation, memory load/store width, and instruction scheduling. Note that the bare-metal tunings are more microarchitectural specific than application specific. With the support of assembly, our scheduling optimization is totally derived from instruction dependency and latency, which is not specific to SGEMM. With a proper blocking algorithm, multiple float computation instructions reside in a single loop iteration; then, register allocation and dual-issue optimization can be used to improve floating-point computation throughput.

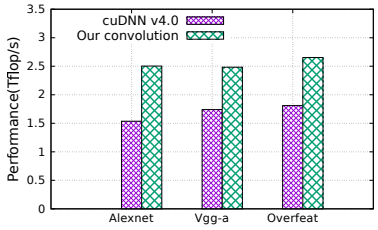


Figure 11: Performance comparison of cuDNN and our optimized convolution on GPU Tesla K20m (batch size is 128).

Applying Optimizations to Convolution Algorithm: Our convolution algorithm implementation uses 128×128 shared memory blocking and 8×8 register blocking. We optimize it at the assembly level by using the method described in section 4. We benchmark three popular convolutional neural networks: Alexnet [15], Vgg [24], and Overfeat [23]. The configuration of each convolutional layer can be found in corresponding papers. Figure 11 presents the performance of our optimized convolution and cuDNN by comparing the

average performance of all the convolutional layers in a neural network. Our optimized convolution implementations are 39%, 46%, and 62% higher than cuDNN V4.0 for the three CNN configurations respectively on Tesla K20m.

7. Related Work

This section briefly discusses related work in reverse engineering ISA encoding, GPU assembler, microbenchmarking, and SGEMM optimization.

ISA Encoding Solver and GPU Assembler: CPU ISA encoding reverse engineering work [8, 13] broadly exists by using a MIPS, SPARC, Alpha, PowerPC, ARM, or x86 assembler to crack CPU instruction sets and extract bit-level instruction encoding information. These works output C declarations that can be used by binary tools. However, NVIDIA does not have an available GPU assembler, but a disassembler instead. The lack of a GPU assembler for public use makes our reverse engineering on GPUs more meaningful.

The absence of a GPU assembler motivates a series of works that develop toolchains to facilitate tuning codes at the assembly level. For the early architecture G80, Decuda [27] demonstrated the feasibility of operating assembly instructions. After that, for almost every generation of GPU architecture, there are several efforts to develop assembly toolchains. Asfermi [12], and cudaasm-qhasm [5] are assemblers for the Fermi architecture, and MaxAs [10] assembler is for the Maxwell architecture. Envytools [1] supports the translation of PTX instructions to 64-bit binaries for several GPU architectures; it is not able to generate a compatible `cubin` that can be directly used by the CUDA driver APIs. Although all their work involves reverse engineering GPU ISA encodings, detailed descriptions are not provided nor is a general tool developed to crack a new GPU ISA encoding. We provide a general toolchain to automatically crack different GPU ISA encodings and an assembler for the Kepler architecture.

Benchmarking: Fog [9] did sound work on benchmarking instruction latency and throughput of several generations of Intel and AMD CPUs, and Intel Xeon Phi accelerators. Wong et al. [31] performed a comprehensive benchmarking study on GT200 and provided pipeline latency data and memory features. Mei et al. [18] benchmarked memory hierarchy including cache, and shared memory on Fermi, Kepler, and Maxwell GPUs. However, they did not benchmark the dual-issue mode of arithmetic instructions or vectorized load instructions such as `LD` and `LDS`, which leaves non-negligible unexplored performance space. We leverage the complete assembler to crack control codes that reveal more microarchitecture details for tuning application performance.

Matrix Multiplication Tuning: With respect to GEMM optimization in the microarchitectural level, some work inspires our GPU implementations. Tan et al. [25] implemented a fast DGEMM by using assembly-level optimization, such as software pipelining, vector memory operations, and instruction scheduling. Lai et al. [16] presented performance analysis and optimization work of SGEMM on both Fermi and GTX680 GPUs. Gray [11] presented SGEMM optimization in assembly on a Maxwell GPU. We adopt the proved effective optimization techniques such as shared memory/register blocking and double-buffering [25, 29]. However, it is difficult to tune all possible microarchitectural optimizations well without a useful toolchain. In this work, we present a complete case study of applying microarchitectural features by combining register allocation, dual issuing, memory access paths, and instruction scheduling by using our toolchains.

8. Conclusion

We presented a methodology to understand GPU microarchitectural features and demonstrated its application on tuning SGEMM and convolution algorithms. The methodology includes a GPU ISA solver to crack its ISA encoding, a GPU assembler, and microbenchmarks at the assembly level to correlate architectural features with performance. Based on the disclosed insights, we implemented the fastest SGEMM, 3.1Tflop/s with 88% efficiency on an NVIDIA Tesla K20m, which is 15% faster than cuBLAS. We also build roofline models to analyze its performance upper bound. Applying these optimizations on convolution, the optimized implementations outperform cuDNN4.0 by 39%-62% on the Tesla K20m. Our work brings GPU optimizations to a deeper native machine level, which may shed light on developments in compilers or other performance-critical kernels. In the future, we will apply our methodology to the latest Pascal GPU to reveal its microarchitectural features. We also intend to apply our GPU microarchitectural insights and assembly optimization to optimize code generation of the open source GPU compiler GPUCC [32].

Acknowledgments

We would like to thank Prof. Mary Hall and other reviewers for the very useful comments and suggestions which help us improve the quality of our paper. This work is supported by the National Key Research and Development Program of China (2016YFB0201305, 2016YFB0200504, 2016YFB0200803, 2016YFB1000400), National 863 Foundation of China (2015AA01A301, 2015AA015303), National Natural Science Foundation of China, under grant no. (91430218, 31327901, 61472395, 61272134, 61432018, 61521092), the joint deep learning lab of Institute of Computing Technology and Sugon, and CAS Holdings.

References

[1] Envytools. `https://github.com/envytools/envytools`.

[2] AMD. clMath. `https://github.com/clMathLibraries`.

[3] ACML AMD. AMD Core Math Library (ACML), 2014.

[4] Alexandro Baldassin, Paulo Cesar Centoducatte, and Sandro Rigo. Extending the archc language for automatic generation of assemblers. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pages 60–67. IEEE, 2005.

[5] Daniel J Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. Usable assembly language for GPUs: a success story. *IACR Cryptology ePrint Archive*, 2012:137, 2012.

[6] Lung-Sheng Chien. Hand tuned SGEMM on GT200 gpu. Technical report, Tech. rep., Department of Mathematics, Tsing Hua University, Taiwan, 2010.

[7] Jack Hilaire Choquette, Manuel Olivier Gautho, and John Erik Lindholm. Methods and apparatus for source operand collector caching, January 28 2014. US Patent 8,639,882.

[8] Christian S Collberg. Reverse interpretation+ mutation analysis= automatic retargeting. In *ACM SIGPLAN Notices*, volume 32, pages 57–70. ACM, 1997.

[9] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Denmark (Lyngby): Technical University of Denmark*, 2012.

[10] Scott Gray. MaxAs. `https://github.com/NervanaSystems/maxas`.

[11] Scott Gray. NervanaGPU. `https://github.com/NervanaSystems/maxas/wiki/SGEMM`.

[12] Yunqing Hou. AsFermi. `https://code.google.com/archive/p/asfermi/wikis`.

[13] Wilson C Hsieh, Dawson R Engler, and Godmar Back. Reverse-engineering instruction encodings. In *USENIX Annual Technical Conference, General Track*, pages 133–145, 2001.

[14] MKL Intel. Intel Math Kernel Library, 2007.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Junjie Lai and André Seznec. 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*, pages 1–10. IEEE, 2013.

[17] M. Lukyanov, B. Beylin, R.S. Glanville, and A. Grosul. Efficient placement of texture barrier instructions, February 20 2014. US Patent App. 13/590,075.

[18] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. Benchmarking the memory hierarchy of modern GPUs. In *Network and Parallel Computing*, pages 144–156. Springer, 2014.

[19] R Nath, S Tomov, and J Dongarra. An improved MAGMA GEMM for Fermi GPUs, university of tennessee computer science technical report. Technical report, UTCS-10-655, July, 2010.

[20] Nvidia. Nvidias next generation CUDA compute architecture: Kepler GK110, the fastest, most efficient HPC architecture ever built. White Paper, 2012.

[21] NVidia. CUDA binary utilities. September 2015.

[22] NVIDIA. Parallel thread execution ISA v4.3. `http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz42f7ftJVy`, September 2015.

[23] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2011.

[26] D. Tarjan and K. Skadron. Policy based allocation of register file cache to threads in multi-threaded processor, June 12 2012. US Patent 8,200,949.

[27] Wladimir J. van der Laan. Decuda. `https://github.com/laanwj/decuda`.

[28] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[29] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[30] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

[31] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[32] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: an open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 105–116. ACM, 2016.

[33] Xiuxia Zhang. KeplerAs. `https://github.com/PAA-NCIC/PPoPP2017_artifact`.