Hand-Tuned SGEMM on GT200 GPU

Lung-Sheng Chien
Department of Mathematics, Tsing Hua university, R.O.C. (Taiwan)
d947207@oz.nthu.edu.tw

February 2010, version 1.1

Abstract: in this work, we improve SGEMM of Volkov's code [1]. $C = AB$ is tested on square matrices A, B and C with dimension N. On TeslaC1060 with CUDA 2.3, compared with Volkov's code for large N ($N > 1500$), we have

(1) N = multiple of 64: ~ 20% improvement,

(2) N = multiple of 16: > 15% improvement and

(3) N = multiple of 8: > 10% improvement.

Averagely speaking, we have 10% improvement for large N. As far as peak performance is concerned, our method reaches 440Gflop/s on TeslaC1060, which is 70% of peak performance of single precision without dual issue. Volkov's code reaches 346 Gflop/s, which is 55.45% of peak performance.

Moreover out-of-array bound checker is added into our method and the same performance improvement is achieved on TeslaC1060, but about 5% performance will be lost when adding out-of-array bound checker on game card, GTX285. However we can guarantee 10% improvement on game card for large dimension even out-of-array bound checker is added.

Basic idea: we replace MAD with shared memory operand, "MAD dest, [smem], src2, src3", by two operations. First one is movement from shared memory to register, "MOV reg, [smem]", and the other is MAD without shared memory operand, "MAD dest, src1, src2, src3".

However compiler ***nvcc*** cannot accomplish such simple idea, we use package **decuda/cudasm** to modify binary code. Thanks to Wladimir J. van der Laan and Sylvain Collange, we can manipulate ***cudasm*** to act on some parts of our source code to do the correct translation.

In this work, we also try to build a cost model as guide of optimization, however we have better result than expectation of the model. So far no good model to explain high performance of our method. Moreover we release binary code which can be loaded into application by driver API. However such binary code is not compatible with future architecture, Fermi. We must re-do whole work for Fermi again, hope that official **decuda/cudasm** is ready at that time.

Introduction

Matrix-multiplication $C = AB$ is a basic operation in linear algebra and NVIDIA provides an example in SDK to explain how to utilize shared memory to avoid overhead of memory latency. However matrix-multiplication example in SDK with shared memory to store sub-matrix of A and B is not the fastest method, Vasily Volkov and James W. Demmel provide another faster algorithm in 2008 [1]. Volkov's method stores sub-matrix of B to shared memory but uses registers to hold sub-matrix of A. This modification saves one movement from shared memory to register per MAD ($c = a \times b + c$) operation, that is why Volkov's code has better performance than SDK example. Volkov's code can apply to both single precision (SGEMM) and double precision (DGEMM). Under GT200 architecture, performance of double precision is 1/8 of single precision, so DGEMM reaches 70Gflop/s which is 94% of peak performance. It is impossible to improve DGEMM anymore. However SGEMM only reaches 55.45% of peak performance, we still have a chance to accelerate it.

In this work, we focus on $C = \alpha AB + \beta C$ where size of *A* is $m \times k$, size of *B* is $k \times n$, and size of *C* is $m \times n$. Three matrices are stored by column-major and have leading dimension lda, ldb, ldc. Moreover matrices are allocated in linear memory, no texture memory is used. In other words, we focus on how to decrease overhead of MAD, not overhead of memory latency.

Basic operation in SGEMM is $c = a \times b + c$ where *c* can be reside in register and *a, b* must come from matrix *A* and *B*. Volkov's method keeps *a* in register and fetches *b* from shared memory but use only one MAD operation to execute $c = a \times b + c$. Such MAD operation in Volkov's code has one shared memory operand *b*, say "MAD *c, b, a, c*". We decompose MAD in Volkov's code into two operations.

(1) move shared memory to register: MOV src1, *b*

(2) MAD without shared memory operand: MOV *c, src1, a, c*

Our basic idea is to reduce number of data transfer from shared memory to register such that cost of $c = a \times b + c$ approaches to cost of " MOV dest, src1, src2, src3" where dest, src1, src2 and src3 are registers. Note that the best performance of $c = a \times b + c$ occurs when *a, b* and *c* are all registers, so it is reasonable to reduce operation " MOV src1, *b* " such that " MOV dest, src1, src2, src3" dominates computation, which is the best way we can expect.

The remaining sections are organized as follows: some preliminaries are introduced in section 1, including hardware category and algorithm category. Hardware category lists SPEC of GT200 (TeslaC1060, GTX285 and GTX295). Also we calibrate pipeline latency and throughput of MAD operation, latency and throughput of global DRAM. Algorithm category describes block version of matrix-multiplication and two representations of matrix-multiplication, one is inner-product based, the other is outer-product based. Then we review structure of Volkov's code and compare it with SGEMM in CUBLAS (CUDA 2.3) in section 2. Our idea is introduced in section 3, after discussion of theoretical profile of Volkov's code, we give a intuition why we replace "MAD dest, [smem], src2, src3" by "MAD dest, src1, src2, src3". Also difficulty arises because compiler **nvcc** cannot implement our idea. In section 4, we mention method 1 (first proposed method), and its variant, method1_variant. We need package **decuda/cudasm** to modify binary code of method1_variant. In this section, we spend much effort to explain how to use **decuda/cudasm** to modify binary code. Structure and performance of method 2 and method 4 are shown in section 5 and 6. In section 7, we provide a workaround to avoid the obstacle, " Volkov's code does not work for general dimension " if one want to use method 1 on arbitrary dimension. In section 8, out-of-array bound check is added into method 1and the experimental result is good. Moreover we propose method 8 in section 9, which achieves uniform performance on TeslaC1060 but sacrifices performance a little bit on game card. Finally we have some conclusions in section 10.

Remark 1: in this work, we use Volkov's code as baseline and call it as algorithm *volkov*, however we abbreviate it as *volkov*, or *volkov* method.

Table of Contents

1 preliminary

1.1 Hardware issue (GT200 architecture)

1.1.1 SPEC of GT200

In this work we use three GPUs listed in Table 1 to measure performance of SGEMM. All three GPUs belong to GT200 series but GTX brand does overclocking core frequency and memory speed. Under dual issue [5], one SP can deliver one MAD ($c = a \times b + c$) operation and one MUL ($c = a \times b$) operation every clock (in fact, SM can issue one MAD and one MUL in 4 cycles per warp). So peak performance is three flops per clock since MAD is combination of multiplication and addition (its flop count is two) and flop count of MUL is one.

Single precision peak performance = $240(core) \times 1.3(core\ freq.) \times 3(flop\ count)$

However 8 SPs share one 64-bit FMAD module and have no dual issue, so performance of double precision is $\frac{1}{12}$ of single precision with dual issue or $\frac{1}{8}$ of single precision without dual issue.

Double precision performance = $\frac{1}{8} \times 240(core) \times 1.3(core\ freq.) \times 2(flop\ count)$

Main cost in SGEMM is MAD and dual issue can be neglected since it is unlikely to merge MAD and MUL in flight due to few MUL operations in SGEMM. Hence we also report Single precision performance without dual issue which is actual limit of SGEMM that we can pursue.

Single precision performance without dual issue = $240(core) \times 1.3(core\ freq.) \times 2(flop\ count)$

Second, GPUs typically address this granularity issue by dividing their memory interfaces into multiple channels. Each channel can serve one read or write request at a time, so an interface with multiple channels can serve multiple requests simultaneously [3]. GT200 series has 64-bit per channel, so 448-bit has 7 channels and 512-bit has 8 channels. Access pattern of channels in global memory can play an important role in matrix transpose [6], however it is not significant in SGEMM.

	GTX295[1]	GTX285	TeslaC1060
# of Streaming Processor	240	240	240
Core Frequency	1242MHz	1476 MHz	1.3 GHz
Memory Speed	999MHz	1242 MHz	800 MHz
Memory Interface	448-bit (7 channel)	512-bit (8 channel)	512-bit (8 channel)
Memory Bandwidth (GB/s)	112	159	102
SP, peak (Gflop/s)	894	1063	933
SP without dual issue	596.2	708.5	624
DP, peak (Gflop/s)	74.5	88.6	78
DRAM (MByte)	896	1024	4096

Table 1: The list of the GPUs in this paper. SP is single precision performance and DP is double precision

[1] Although GTX925 has two GPU units (assembly of two GTX275), proposed SGEMM is executed in single GPU such that we only repost SPEC of one GPU.

performance.

1.1.2 latency and throughput of MAD

There are two kinds of MAD when operands are "float",

(1) "MAD dest, src1, src2, src3" corresponds to $dest = src1 \times src2 + src3$ where dest, src1, src2 and src3 are all registers.

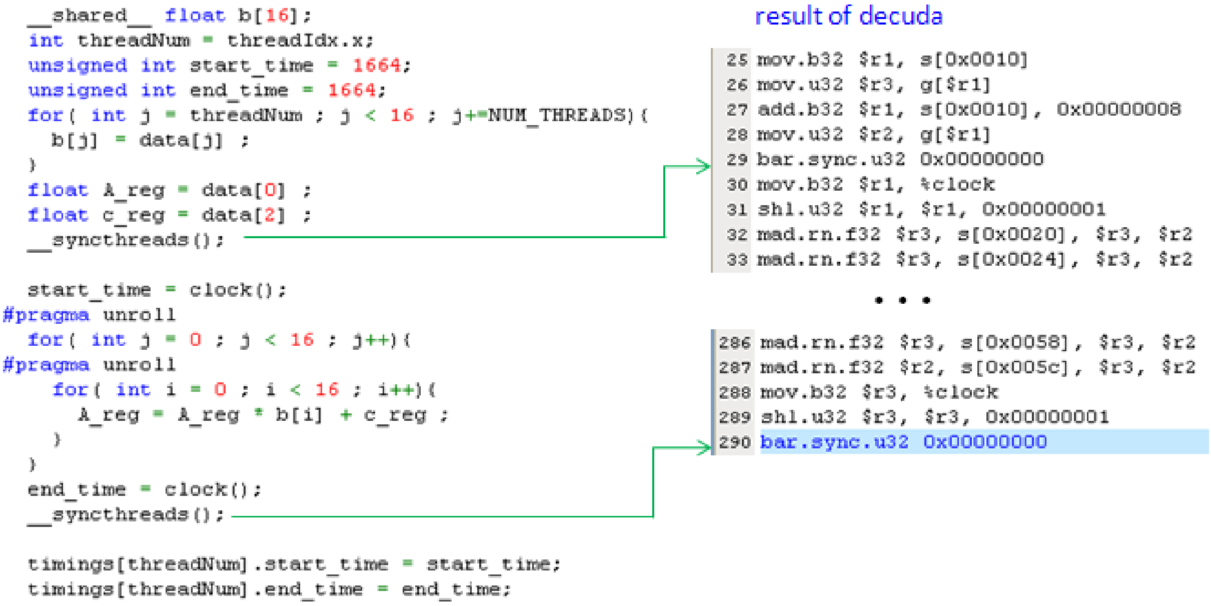
(2) "MAD dest, [smem], src2, src3" corresponds to $dest = [smem] \times src2 + src3$ where [smem] denotes shared memory.

These two MAD operations have different pipeline latency and throughput. In our method, we change the later in Volkov's code to the former to improve performance.

1.1.2.1 "MAD dest, [smem], src2, src3"

To evaluate pipeline latency of MAD operation, we execute $a = a \times b[i] + c$ 256 times where $a = a \times b[i] + c$ has read-after-write hazard, then do timing profile for each thread. Source code is listed in Figure 1 and corresponding result of ***decuda*** shows that ***nvcc*** translates all $a = a \times b[i] + c$ into "MAD dest, [smem], src2, src3".

To calibrate "MAD dest, [smem], src2, src3", we use one thread block in one SM (stream multiprocessor). So execution configuration is dim3 grid(1,1,1) and dim3 block(NUM_THREADS,1,1). We control parameter NUM_THREADS to setup how many threads in a block.



```
  __shared__ float b[16];
  int threadNum = threadIdx.x;
  unsigned int start_time = 1664;
  unsigned int end_time = 1664;
  for( int j = threadNum ; j < 16 ; j+=NUM_THREADS){
    b[j] = data[j] ;
  }
  float A_reg = data[0] ;
  float c_reg = data[2] ;
  __syncthreads();

  start_time = clock();
#pragma unroll
  for( int j = 0 ; j < 16 ; j++){
#pragma unroll
    for( int i = 0 ; i < 16 ; i++){
      A_reg = A_reg * b[i] + c_reg ;
    }
  }
  end_time = clock();
  __syncthreads();

  timings[threadNum].start_time = start_time;
  timings[threadNum].end_time = end_time;
```

result of decuda

```
mov.b32 $r1, s[0x0010]
mov.u32 $r3, g[$r1]
add.b32 $r1, s[0x0010], 0x00000008
mov.u32 $r2, g[$r1]
bar.sync.u32 0x00000000
mov.b32 $r1, %clock
shl.u32 $r1, $r1, 0x00000001
mad.rn.f32 $r3, s[0x0020], $r3, $r2
mad.rn.f32 $r3, s[0x0024], $r3, $r2
```

• • •

```
mad.rn.f32 $r3, s[0x0058], $r3, $r2
mad.rn.f32 $r2, s[0x005c], $r3, $r2
mov.b32 $r3, %clock
shl.u32 $r3, $r3, 0x00000001
bar.sync.u32 0x00000000
```

Figure 1: code to calibrate "MAD dest, [smem], src2, src3"

Each thread executes 256 MAD operations, we measure average time per MAD by $\frac{end\_time - start\_time}{256}(cycle / MAD)$ and report two numbers in Table 2, one is minimum time among all threads, the other is maximum time among all threads. Clearly, result of one thread reveals pipeline latency of "MAD dest, [smem], src2, src3", which is 34.6 cycle. Throughput can be calculated via result of 512 threads, say $\frac{96.1(cycle)}{32(warp)} \sim \frac{96.9(cycle)}{32(warp)} \sim 6(cycle/warp)$. The value of throughput matches result of Volkov's paper [1].

NUM_THREADS	1	64	128	192	224	256	288	320	384	512
Minimum time	34.6	34.6	34.7	38.6	42.4	46.6	54.0	60.2	72.1	96.1
Maximum time	34.6	34.6	34.8	39.3	43.2	49.5	54.7	60.6	72.7	96.9
Total time for one a = a*b_smem +c	34.6	34.6	34.6	36	42	48	54	60	72	96

Table 2: average number of cycles per "MAD dest, [smem], src2, src3" on TeslaC1060. Pipeline latency is 34.6 cycle and throughput is 6 cycle /warp.

Six active warps per SM are required to hide pipeline latency of "MAD dest, [smem], src2, src3" under throughput is 6 cycle/warp. This is understandable if warp scheduling is round-robin. For example, Figure 2 shows Gatt chart of "MAD dest, [smem], src2, src3" under 6 active warps per SM. Moreover if we invoke more than 6 warps in a SM, then total time of one "*a* = *a* * *b_smem* + *c*" is (6 cycle) *x* (number of warps). Otherwise, total time is determined by pipeline latency, see fourth row of Table 2.

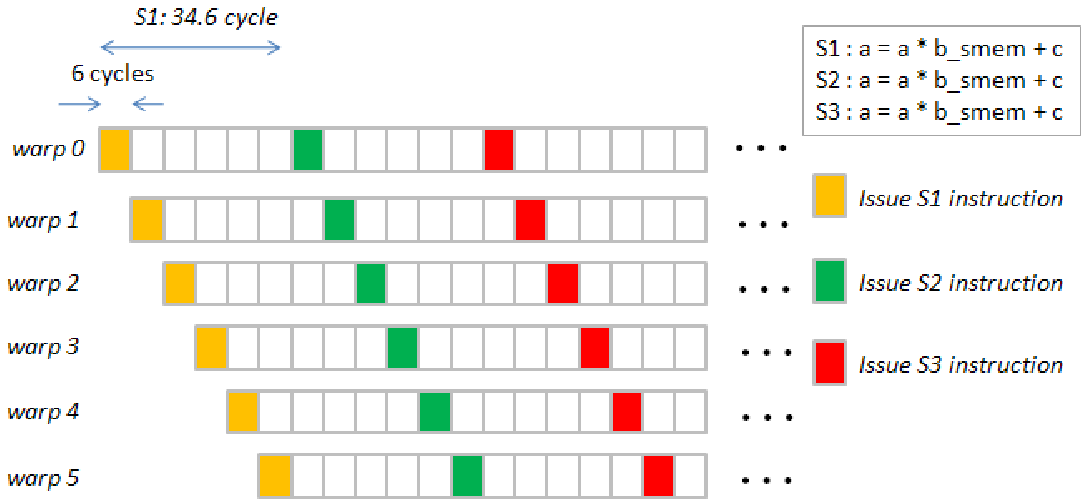
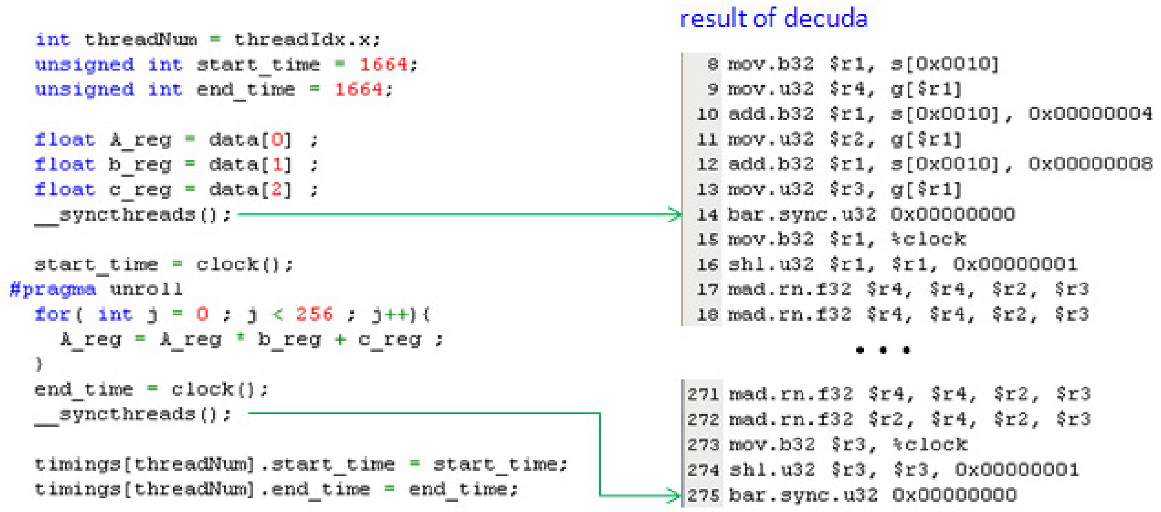


Figure 2: Gatt chart of "MAD dest, [smem], src2, src3" when 6 active warps in a SM.

1.1.2.2 "MAD dest, src1, src2, src3"

Similarly we execute $a = a \times b + c$ 256 times, then do timing profile for each thread. Source code is listed in Figure 3 and corresponding result of ***decuda*** shows that ***nvcc*** translates all $a = a \times b + c$ into "MAD dest, src1, src2, src3".



```
  int threadNum = threadIdx.x;
  unsigned int start_time = 1664;
  unsigned int end_time = 1664;

  float A_reg = data[0] ;
  float b_reg = data[1] ;
  float c_reg = data[2] ;
  __syncthreads();

  start_time = clock();
#pragma unroll
  for( int j = 0 ; j < 256 ; j++){
    A_reg = A_reg * b_reg + c_reg ;
  }
  end_time = clock();
  __syncthreads();

  timings[threadNum].start_time = start_time;
  timings[threadNum].end_time = end_time;
```

```
mov.b32 $r1, s[0x0010]
mov.u32 $r4, g[$r1]
add.b32 $r1, s[0x0010], 0x00000004
mov.u32 $r2, g[$r1]
add.b32 $r1, s[0x0010], 0x00000008
mov.u32 $r3, g[$r1]
bar.sync.u32 0x00000000
mov.b32 $r1, %clock
shl.u32 $r1, $r1, 0x00000001
mad.rn.f32 $r4, $r4, $r2, $r3
mad.rn.f32 $r4, $r4, $r2, $r3

          • • •

mad.rn.f32 $r4, $r4, $r2, $r3
mad.rn.f32 $r2, $r4, $r2, $r3
mov.b32 $r3, %clock
shl.u32 $r3, $r3, 0x00000001
bar.sync.u32 0x00000000
```

Figure 3: driver to calibrate "MAD dest, src1, src2, src3"

From Table 3, pipeline latency of "MAD dest, src1, src2, src3" is 31.5 cycle but we have a little confused how to obtain throughput since minimum time is much different from maximum time, this is impossible under round-robin assumption. We estimate throughput by $\frac{time}{\text{number of warps}}$ and report this in second number of Table 3 when NUM_THREADS is greater than 256. Pessimistically we accept estimated throughput from maximum time and then throughput of "MAD dest, src1, src2, src3" is about 4 cycle per warp.

NUM_THREADS	1	64	128	192	224	256	320	384	512
Minimum time	31.5	31.5	31.4	31.5	34.9	33.9 4.2	38.8 3.9	24.4 2.03	29.7 1.86
Maximum time	31.5	31.5	31.5	31.7	35.5	37.0 4.6	42.6 4.3	50.5 4.21	66.6 4.16

Table 3: first number is average number of cycles per "MAD dest, src1, src2, src3" on TeslaC1060. Second number is estimate of Pipeline latency is 31.5 cycle and throughput is about 4 cycle /warp.

1.1.3 latency and throughput of global memory

1.1.3.1 Latency of global memory

Latency of global DRAM is not determined by a single factor, on NVIDIA forum [7], Sylvain Collange lists possible reasons , including

(1) Virtual address calculation

(2) On-chip crossbar interconnect traversal,

(3) Virtual to physical address translation,

(4) Physical to raw address translation (includes a division/modulo to accommodate non-power-of-two numbers of

partitions),

(5) Reordering from a deep buffer. Memory controllers aggressively reorder accesses to minimize DRAM page switching and read/write turnaround overheads, trading latency for throughput,

(6) The DRAM read cycle itself,

(7) Going back through the interconnect,

(8) Going through texture filtering units, if there is no shortcut datapath,

So far, we have two official results about latency of global memory

(*I*) from programming guide 5.1.1.3, it says "Throughput of memory operations is 8 operations per clock cycle. When accessing local or global memory, there are, in addition, 400 to 600 clock cycles of memory latency.

(*II*) Misel-Myrto reports ~441 cycle latency in [8].

However in our experiment, memory latency is about 550 cycle on TeslaC1060.

Next we want to build a latency model used in this work on TeslaC1060.

Data rate of memory bus in TeslaC1060 is 800MHz but core frequency is 1.3GHz, so 1 memory cycle = 1.625 core cycle. Second TeslaC1060 has 512-bit memory interface, which is divided into 8 channel s (64-bit per channel) and each channel occupies 256 byte width (64-float) [6]. If one 16-float read transaction falls into one channel, then it requires 16*4*8/64 = 8 memory cycle or say 8 * 1.625 = 13 core cycle.

In Figure 4, one read transaction of 16-float is considered, half-warp needs two cycles to issue a read command, then we assume fixed cost is 500 core cycle after read command. Such fixed cost is due to all reasons except data transfer through data bus. Data transfer of 16-float needs 13 core cycle. Hence total cost of one transaction is 2 + 500 + 13 = 515 core cycle.

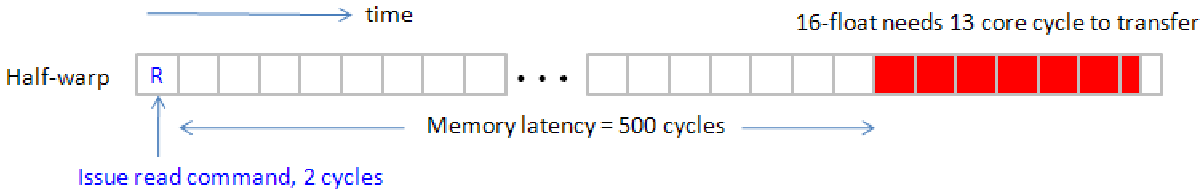

time
16-float needs 13 core cycle to transfer
Half-warp
R
Memory latency = 500 cycles
Issue read command, 2 cycles

Figure 4: cost of one memory transaction of 16 floats.

However it is better to consider cost of read transaction per warp since throughput of arithmetic operations is calibrated in terms of warp, not half-warp. We combine read transaction of two consecutive half-warp in Figure 5. It needs 4 cycle to issue a read command and 26 cycle to transfer 32-float through data bus, hence memory latency of a warp is 4 + 500 + 26 = 530 core cycle.

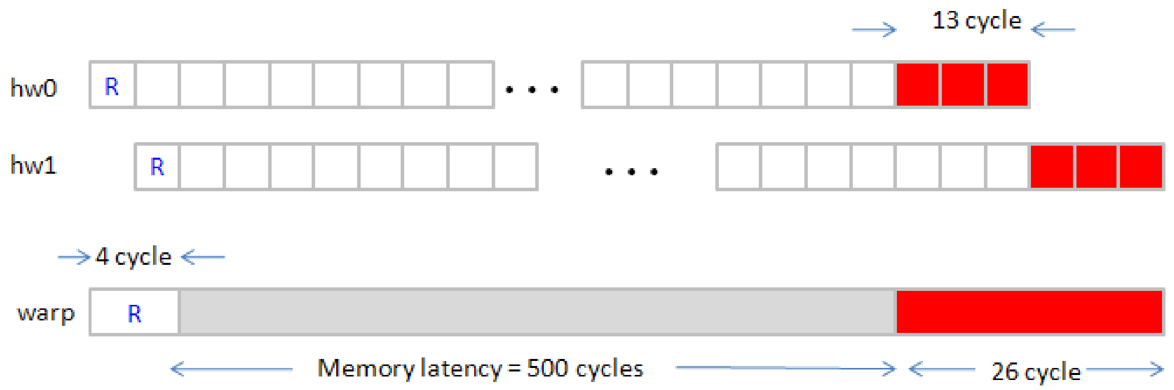


Figure 5: cost of one transaction of 32-float in a warp.

1.1.3.2 Throughput of global memory

It is intuitive that throughput of memory depends on number of active threads per SM. The more active threads are, the more latency can be hidden. We take operation "*b[inx][iny+i]* = *B[i*ldb]*" as an example to show memory throughput. This operation loads data in global memory *B* to shared memory *b*, and would be seen in Volkov's code later. GPU does not support direct transfer from global DRAM to on-chip shared memory, we use register as a relay to connect these two different memory types. Suppose that occupancy is 50% (512 active threads per SM) and decompose "*b[inx][iny+i]* = *B[i*ldb]*" as

Reg ← *B[i*ldb]*

b[inx][iny+i] ← Reg.

Furthermore, latency of global memory is 530 cycle and latency of shared memory is 36 cycle [1] (our experiment shows 34 cycle on TeslaC1060 [9]). From Gatt chart in Figure 6, under round-robin assumption of warp scheduling, it is clear that 512 threads (16 warps) cannot hide 530 cycle memory latency, after 64 cycle, all 16 warps are in waiting queue, no one obtains data from DRAM till 530-th cycle. Hence warp scheduler issues second command " *b[inx][iny+i]* ← Reg " of warp 0 at 503-th cycle. That is to say, when we talk about memory cost of one thread, we must do averaging, Throughput of global memory is $\frac{530(cycle)}{512(thread)} = 1.16(cycle/thread)$, however if one SM has only N_T active threads, then Throughput of global memory is $\frac{530(cycle)}{N_T(thread)}$.

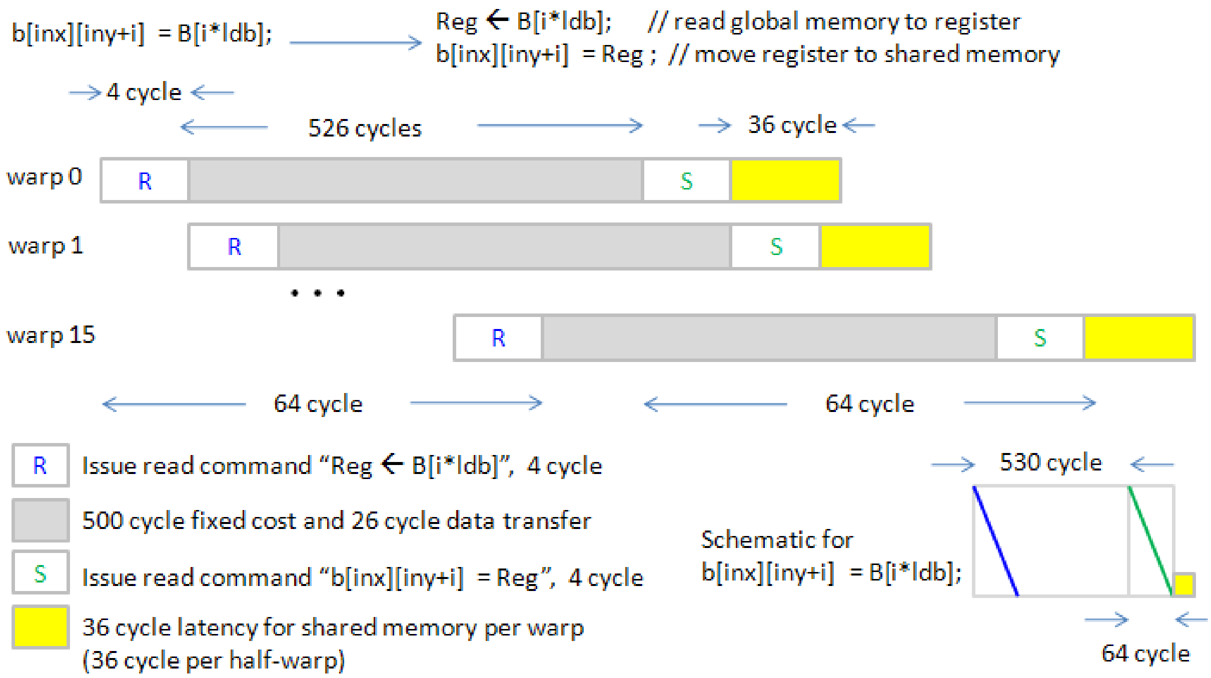


Figure 6: Gatt chart of global memory access followed by shared memory access. Latency of global memory is 530 cycle and latency of shared memory is 36 cycle.

1.2 algorithm issue

1.2.1 notation of matrices and partition of grid, block

Under notations in Volkov's paper [1], in Figure 7, we assume that *A*, *B* and *C* are $m \times k$, $k \times n$ and $m \times n$ matrices respectively. Partition these matrices into $M \times K$, $K \times N$ and $M \times N$ grids of $bm \times bk$, $bk \times bn$ and $bm \times bn$ blocks. Formally $M = \left\lfloor \frac{m + b_M - 1}{b_M} \right\rfloor$, $K = \left\lfloor \frac{k + b_K - 1}{b_K} \right\rfloor$ and $N = \left\lfloor \frac{n + b_N - 1}{b_N} \right\rfloor$. We use register file or on-chip shared memory to store $A_{bm,bk}$ (sub-block of matrix *A*) and $B_{bk,bn}$ (sub-block of matrix *B*), also always use registers to store $C_{bm,bn}$ (sub-block of matrix *C*, should keep $C_{bm,bn}$ in registers since it is destination operand of MAD operation), then all four kinds of SGEMM, including $C = \alpha AB + \beta C$, $C = \alpha A^T B + \beta C$, $C = \alpha AB^T + \beta C$, $C = \alpha A^T B^T + \beta C$ require two-steps computation:

Step 1: fetch *K* blocks of matrices *A* and *B* into $A_{bm,bk}$ and $B_{bk,bn}$ respectively, then compute $C_{bm,bn} = \sum_k A_{bm,bk} B_{bk,bn}$,

$C_{bm,bn} = \sum_k A^T_{bm,bk} B_{bk,bn}$, $C_{bm,bn} = \sum_k A_{bm,bk} B^T_{bk,bn}$ or $C_{bm,bn} = \sum_k A^T_{bm,bk} B^T_{bk,bn}$

Step 2: update $C|_{(bm,bn)}$ which is global matrix *C* at block index (bm, bk) by $C|_{(bm,bn)} = \alpha C_{bm,bn} + \beta C|_{(bm,bn)}$.

We can summarize complexity of data transfer and float-point computation in SGEMM as

(1) read/write *C*: $MN \times b_M b_N = mn$

(2) read *A* to register/shared memory: $MN \times K \times b_M b_K = mnk \frac{1}{b_N}$, independent of dimension *k*.

(3) read *B* to register/shared memory: $MN \times K \times b_K b_N = mnk \frac{1}{b_M}$, independent of dimension *k*.

(4) number of MAD ($c = a \cdot b + c$): $MNK \times b_K b_M b_N = mnk$ independent of grid dimension.

Moreover in this work, we focus discussion on $C = AB$ but deliver source code to deal with $C = \alpha AB + \beta C$. One can make little modification to do remaining three forms of SGEMM.

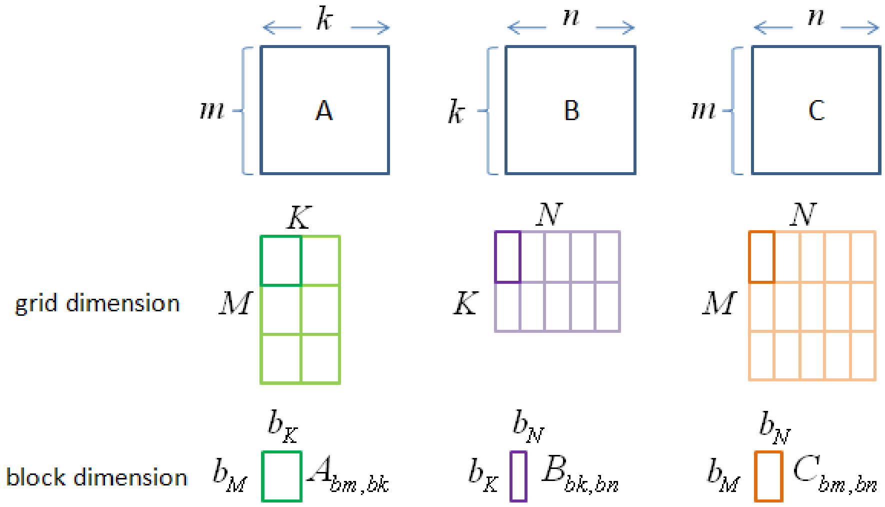


Figure 7: dimension of matrices A, B, C and execution configuration.

1.2.2 inner-product based algorithm

In NVIDIA SDK example, inner-product based matrix-multiplication is demonstrated. $A_{bm,bk}$ and $B_{bk,bn}$ have dimension 16×16 and are stored in shared memory whereas register $C_{bm,bn}$ also has dimension 16×16 per 256 threads. This is fine-grain parallelism since each thread deals with one element of matrix *C*.

for each sub-block $C_{bm,bn}$, we sweep K blocks of *A* and *B* to do matrix-multiplication by following code

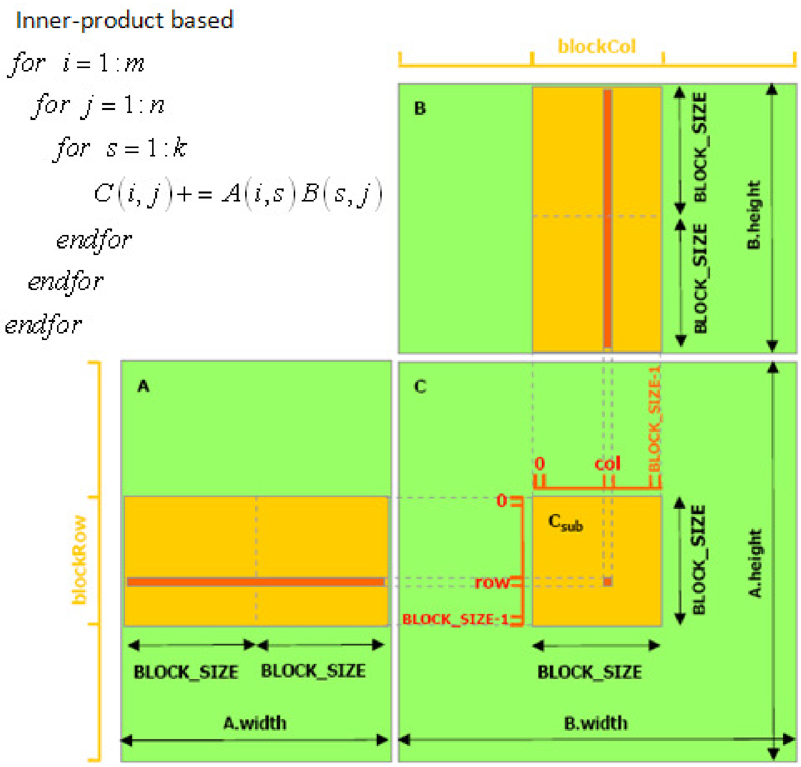
```
for each $k$
    load global matrix $A$ to shared memory $A_{bm,bk}$
    load global matrix $B$ to shared memory $B_{bk,bn}$
    synchronization
    $C_{bm,bn} += A_{bm,bk} B_{bk,bn}$, each thread computes one element of $C_{bm,bn}$
endfor
```

and how to update $C_{bm,bn}$ is depicted in Figure 8.

However such shared-memory based algorithm is not the fast version. Volkov provides another view, called outer-product based algorithm, and this formulation beats shared-memory based algorithm.

1.2.3 outer-product based algorithm

The relationship between inner-product formulation and outer-product formulation is shown in Figure 9, we change loop-order (i,j,s) of inner-product based algorithm to (s,j,i) and expand index i to obtain vector form such that $\vec{C}(j)=C(1:m,j)$ is a column vector. In this formulation, only $B_{bk,bn}$ is stored in shared memory, we can use registers to store $A_{bm,bk}$ column-by-column.

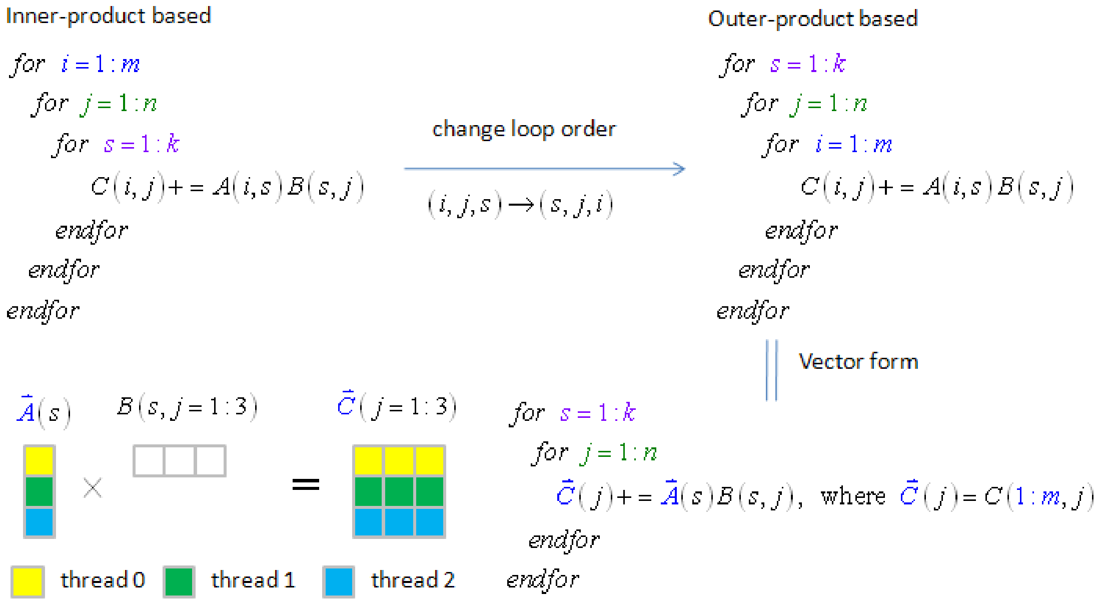


```
Inner-product based
for $i = 1:m$
  for $j = 1:n$
    for $s = 1:k$
      $C(i,j) += A(i,s)B(s,j)$
    endfor
  endfor
endfor
```

Figure 8: algorithm of inner-product based matrix-multiplication, the graph is copied from figure 3.2 in NVIDIA_CUDA_Programming_Guide_2.3.pdf [4]

We take 3×3 matrix-multiplication as an example to show that $B_{bk,bn}$ must be stored into shared memory. Suppose $\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$ and we do loop-unrolling for index j such that outer-product based algorithm can be described as $\left\{\begin{array}{l} \text{for } s=1:3 \\ \quad C(1:3,1) += A(1:3,s)B(s,1) \\ \quad C(1:3,2) += A(1:3,s)B(s,2) \\ \quad C(1:3,3) += A(1:3,s)B(s,3) \\ \text{endfor} \end{array}\right.$.

Figure 10 shows results of executing $s=1$, which does rank-1 update $\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} & b_{13} \end{pmatrix}$ by using first column of *A* and first row of *B*.



Inner-product based

```
for $i=1:m$
  for $j=1:n$
    for $s=1:k$
      $C(i,j) += A(i,s)B(s,j)$
    endfor
  endfor
endfor
```

Outer-product based

```
for $s=1:k$
  for $j=1:n$
    for $i=1:m$
      $C(i,j) += A(i,s)B(s,j)$
    endfor
  endfor
endfor
```

```
for $s=1:k$
  for $j=1:n$
    $\vec{C}(j) += \vec{A}(s)B(s,j)$, where $\vec{C}(j)=C(1:m,j)$
  endfor
endfor
```

Figure 9: relationship between inner-product based formulation and outer-product based formulation. This kind of outer-product formulation is good when matrix A is column-major.

Whole spirit of Volkov's code on rank-1 update is

(1) use three threads to update 3×3 matrix $C_{bm,bn}$, each thread computes one row of $C_{bm,bn}$. In other words, thread j has register array $C_{reg}(1:3)$ satisfying $C_{reg}(1:3)=C_{bm,bn}(j,1:3)$. This is not fine-grain since one thread deals with 3 elements of matrix *C*. However only one column of registers is enough to store $A_{bm,bk}$. The consequence is one $reg \leftarrow smem$ is saved when do MAD operation, $C_{ij}=A_{ik}B_{kj}+C_{ij}$. This is why Volkov's code can be much faster than shared-memory based algorithm in CUBLAS 1.1

(2) $B_{bk,bn}$ must be stored into shared memory since all threads would access the same element of $B_{bk,bn}$ during rank-1 update. Fortunately such access pattern of $B_{bk,bn}$ can activate broadcasting mechanism since all threads access the same element of $B_{bk,bn}$, and broadcasting achieves highest performance in access of shared memory.

Remaining $s=2$ and $s=3$ are shown in Figure 11. They have the same pattern as $s=1$ except different column of $A_{bm,bk}$ and row of $B_{bk,bn}$.

Remark 2: outer-product formulation $$\left\{\begin{array}{l} \text{for } s=1:k \\ \quad \text{for } j=1:n \\ \qquad C(j) += A(s)B(s,j) \\ \quad \text{endfor} \\ \text{endfor} \end{array}\right.$$ fetches one column of matrix *A* into registers. If matrix *A* is stored as column-major, then such access pattern is coalesced. Hence the algorithm is good for $C=\alpha AB+\beta C$ and $C=\alpha AB^T+\beta C$ where *A*, *B* and *C* are column-major. We have another outer-product representation which is good for row-major. In Figure 12. we change loop-order (i,j,s) of inner-product based formulation to (s,i,j) and expand index j. This formulation requires storing $A_{bm,bk}$ in shared memory and storing $B_{bk,bn}$ in registers (just change role of $A_{bm,bk}$ and $B_{bk,bn}$ of former outer-product formulation). This is good when matrix B is stored as row-major. Of course it is also good for $C = \alpha A^T B^T + \beta C$ when A, B and C are column-major.

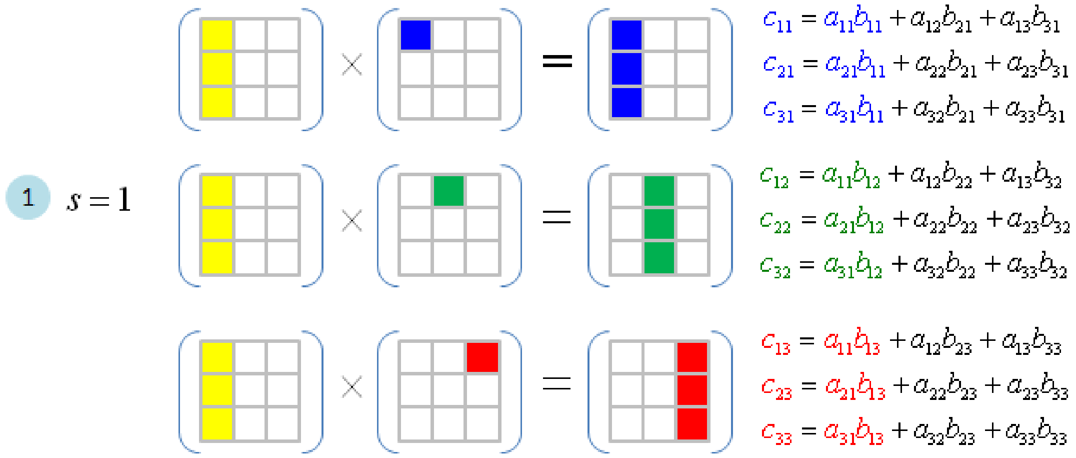

1
$s=1$
$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}$
$c_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}$
$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$
$c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}$
$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}$
$c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33}$
$c_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}$

Figure 10: rank-1 update of first column of A and first row of B

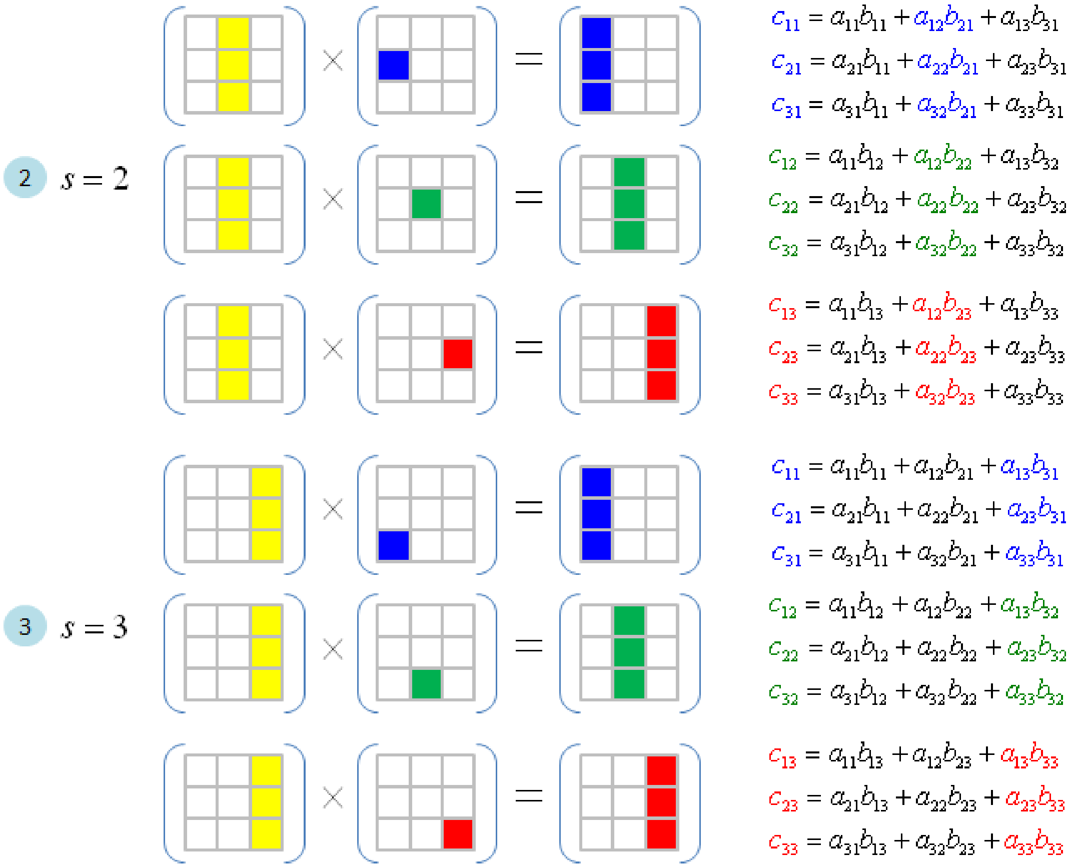
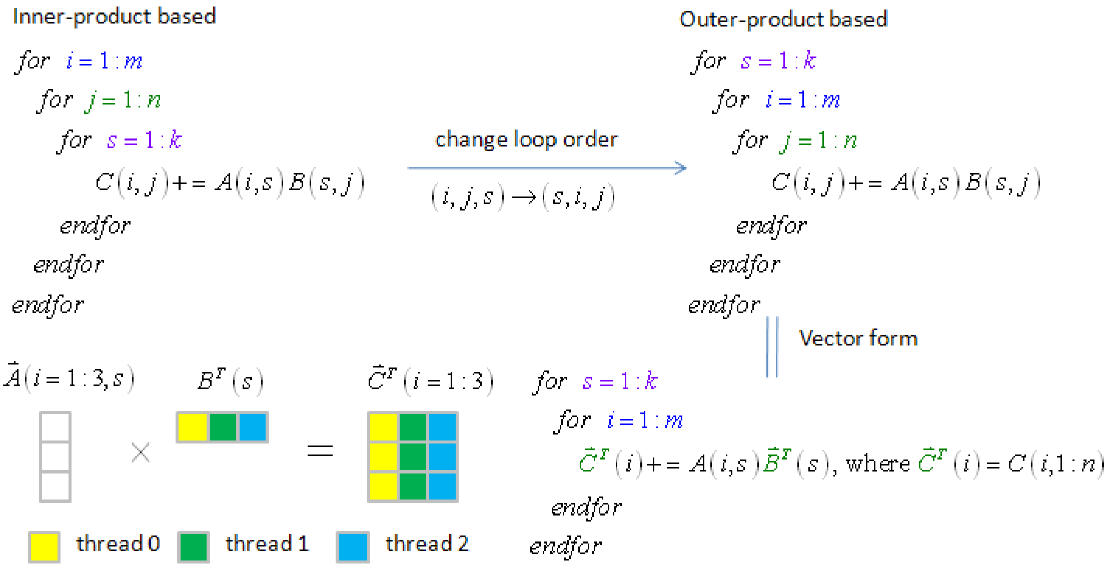

2
$s=2$
$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}$
$c_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}$
$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$
$c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}$
$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}$
$c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33}$
$c_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}$
3
$s=3$
$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}$
$c_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}$
$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$
$c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}$
$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}$
$c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33}$
$c_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}$

Figure 11: rank-1 update of remaining columns of A and rows of B.



Inner-product based

```
for $i = 1:m$
  for $j = 1:n$
    for $s = 1:k$
      $C(i,j) += A(i,s)B(s,j)$
    endfor
  endfor
endfor
```

change loop order

$(i,j,s) \rightarrow (s,i,j)$

Outer-product based

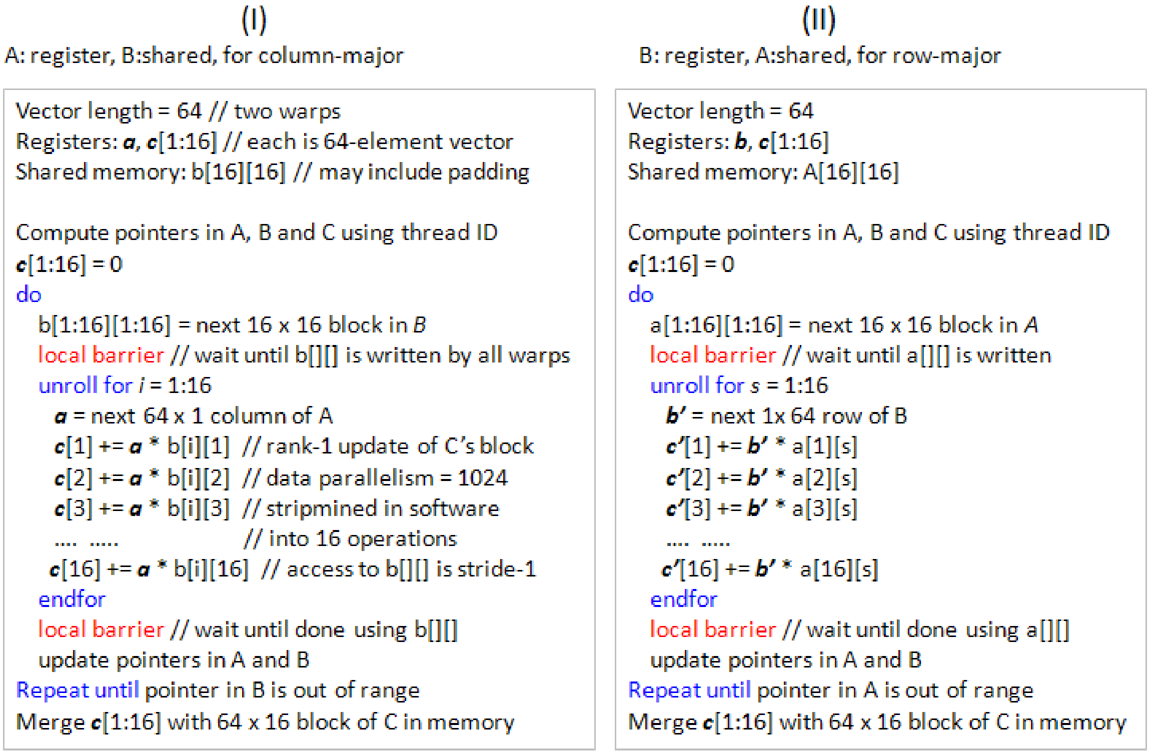
```
for $s = 1:k$
  for $i = 1:m$
    for $j = 1:n$
      $C(i,j) += A(i,s)B(s,j)$
    endfor
  endfor
endfor
```

Vector form

```
for $s = 1:k$
  for $i = 1:m$
    $\vec{C}^T(i) += A(i,s)\vec{B}^T(s)$, where $\vec{C}^T(i) = C(i,1:n)$
  endfor
endfor
```

Figure 12: relationship between inner-product based algorithm and outer-product based algorithm, which is good when matrix B is row-major.

To sum up, we can write down pseudo-code of these two outer-product formulations in Figure 13. In this work we adopt algorithm (I) and parameters of grid and block are depicted in Figure 14.



(I)

A: register, B:shared, for column-major

```
Vector length = 64 // two warps
Registers: a, c[1:16] // each is 64-element vector
Shared memory: b[16][16] // may include padding

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  b[1:16][1:16] = next 16 x 16 block in B
  local barrier // wait until b[][] is written by all warps
  unroll for i = 1:16
    a = next 64 x 1 column of A
    c[1] += a * b[i][1]  // rank-1 update of C's block
    c[2] += a * b[i][2]  // data parallelism = 1024
    c[3] += a * b[i][3]  // stripmined in software
    ....  .....          // into 16 operations
   c[16] += a * b[i][16]  // access to b[][] is stride-1
  endfor
  local barrier // wait until done using b[][]
  update pointers in A and B
Repeat until pointer in B is out of range
Merge c[1:16] with 64 x 16 block of C in memory
```

(II)

B: register, A:shared, for row-major

```
Vector length = 64
Registers: b, c[1:16]
Shared memory: A[16][16]

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  a[1:16][1:16] = next 16 x 16 block in A
  local barrier // wait until a[][] is written
  unroll for s = 1:16
    b' = next 1x 64 row of B
    c'[1] += b' * a[1][s]
    c'[2] += b' * a[2][s]
    c'[3] += b' * a[3][s]
    ....  .....
   c'[16] += b' * a[16][s]
  endfor
  local barrier // wait until done using a[][]
  update pointers in A and B
Repeat until pointer in A is out of range
Merge c[1:16] with 64 x 16 block of C in memory
```

Figure 13: pseudo-code of two outer-product formulations. left panel comes from figure 4 in [1]

Remark 3: algorithm (I) is copied from Volkov's paper, in our method, value of b_M, b_K, b_N may be different but whole structure would be the same.

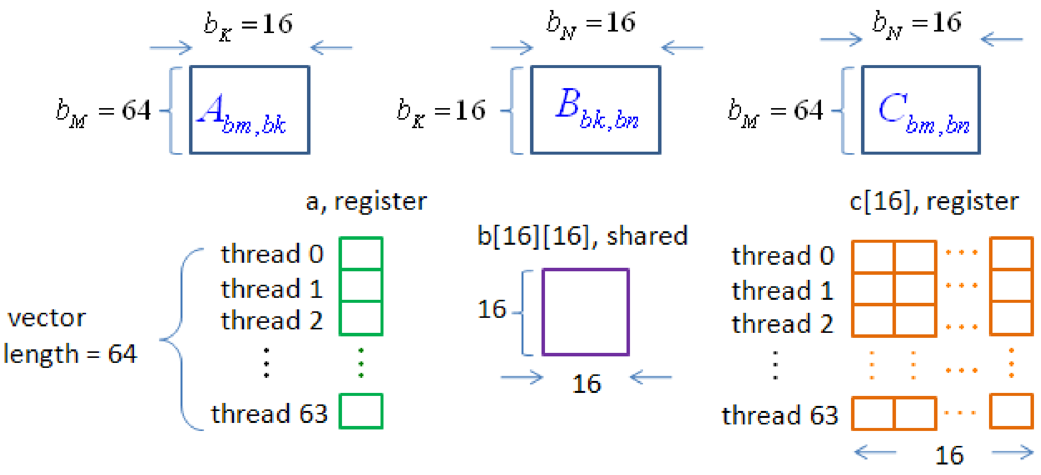


Figure 14: parameters of grid and block in (I) of Figure 13

2 CUBLAS versus Volkov's code

2.1 review Volkov's code

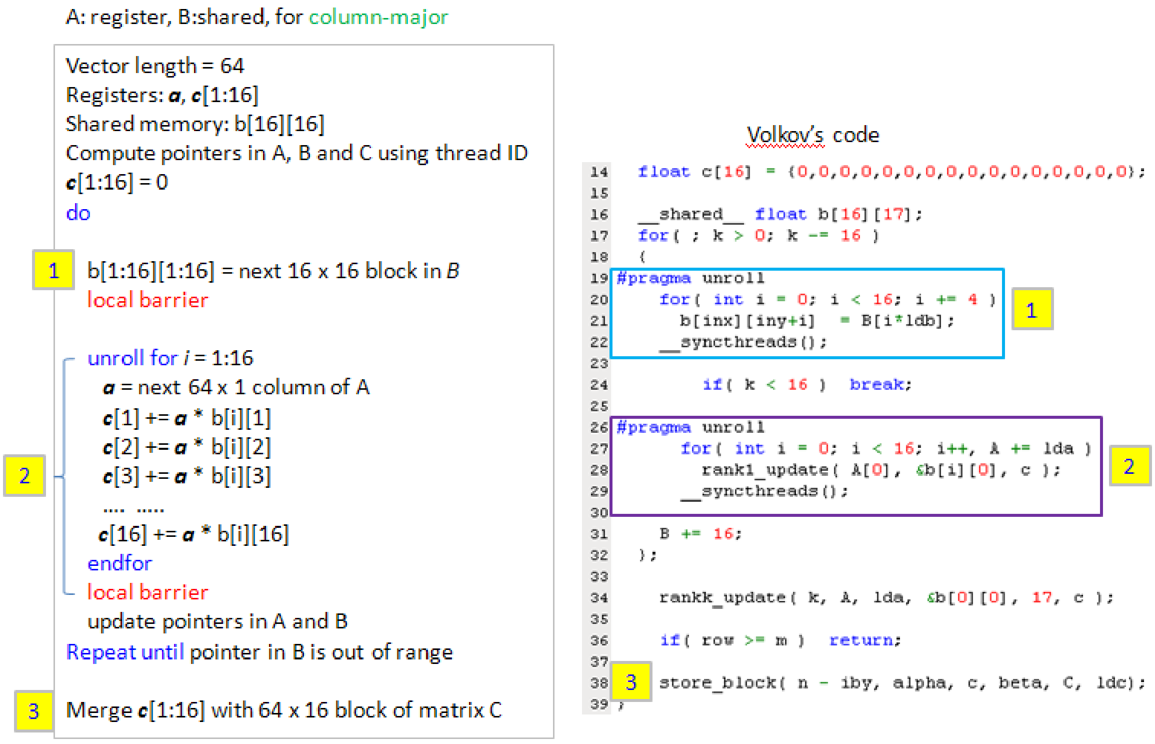
In this work, we use Volkov's code downloaded from http://forums.nvidia.com/index.php?showtopic=89084 as blueprint to build our method. It is necessary to review program structure of Volkovs' code. We have known that Volkov's code uses outer-product formulation, algorithm (I) in Figure 13, and term by term correspondence in Figure 15 can be described as following:

(1) load *16x16* block of matrix *B* into shared memory *b[16][16]*. One thread block has 64 threads (vector length = 64), each thread loads four elements (526/64 = 4).

(2) each thread loads one element of *A* (64 threads load one column of $A_{bm,bk}$) and then does rank-1 update, $c[j] = A[0] \cdot b[i][j] + c[j]$ for $j = 0:15$, where $c[0:15] = C_{bm,bn}(threadID, 0:15)$ is one row of $C_{bm,bn}$.

(3) store $C_{bm,bn}$ into matrix *C*.

Remark 4: in Figure 15 one can also notice device function "**rankk_update**" in Volkov's code, which does rank-1 update *k* times. Consider a simple example in Figure 16, matrix *A* has two sub-block A0 and A1, matrix *B* has two sub-block B0 and B1 and $C = A_0B_0 + A_1B_1$. First matrix-multiplication A_0B_0 requires rank-1 update 16 times (one column of A_0 multiplies one row of B_0), and these 16 rank-1 update code blocks are expanded into a large code block by compiler directive "#pragma unroll". However second matrix-multiplication A_1B_1 requires rank-1 update 7 times, we cannot unroll these 7 code blocks, that's why device function "**rankk_update**" exists.



A: register, B:shared, for column-major

```
Vector length = 64
Registers: a, c[1:16]
Shared memory: b[16][16]
Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  b[1:16][1:16] = next 16 x 16 block in B
  local barrier

  unroll for i = 1:16
    a = next 64 x 1 column of A
    c[1] += a * b[i][1]
    c[2] += a * b[i][2]
    c[3] += a * b[i][3]
    ....  .....
    c[16] += a * b[i][16]
  endfor
  local barrier
  update pointers in A and B
Repeat until pointer in B is out of range
Merge c[1:16] with 64 x 16 block of matrix C
```

Volkov's code

```
  float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

  __shared__ float b[16][17];
  for( ; k > 0; k -= 16 )
  {
#pragma unroll
    for( int i = 0; i < 16; i += 4 )
      b[inx][iny+i]  = B[i*ldb];
    __syncthreads();

        if( k < 16 )  break;

#pragma unroll
      for( int i = 0; i < 16; i++, A += lda )
        rank1_update( A[0], &b[i][0], c );
      __syncthreads();

    B += 16;
  };

    rankk_update( k, A, lda, &b[0][0], 17, c );

    if( row >= m )  return;

    store_block( n - iby, alpha, c, beta, C, ldc);
};
```

Figure 15: program structure of Volkov's code

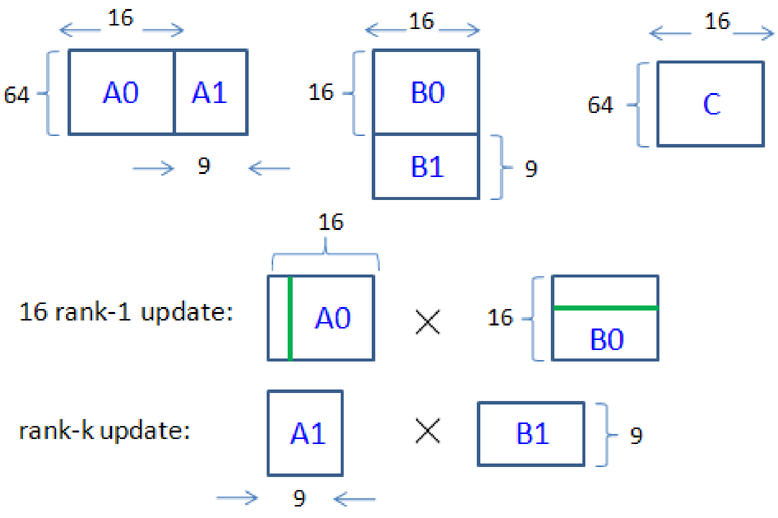


Figure 16: rank-1 update and rank-k update in Volkov's code

Furthermore in order to simplify discussion, we use carton picture to describe grid information of Volkov's code as you see in Figure 17 (we alias $A_{bm,bk}, B_{bk,bn}, C_{bm,bn}$ as A, B and C respectively, just simplify notation, nothing special). Under such picture, one can know $b_M = 64,\ b_K = 16,\ b_N = 16$ and then

(1) read A to register: $mnk\dfrac{1}{b_N} = \dfrac{mnk}{16}$,

(2) read B to shared memory: $mnk\frac{1}{b_M}=\frac{mnk}{64}$, and

(3) number of MAD ($c=a\cdot b+c$): mnk .

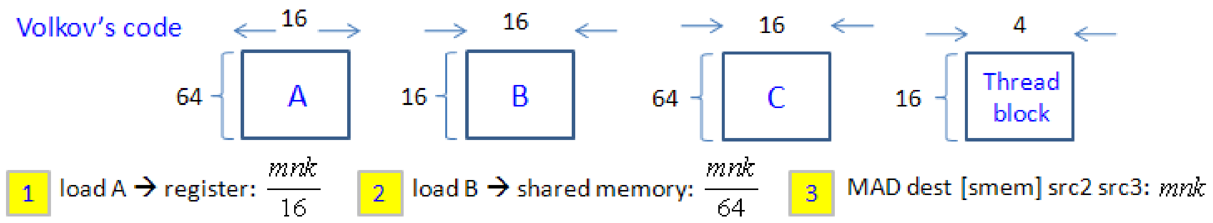

Volkov's code
16
64
A
16
16
B
16
64
C
4
16
Thread block
1
load A → register: $\frac{mnk}{16}$
2
load B → shared memory: $\frac{mnk}{64}$
3
MAD dest [smem] src2 src3: mnk

Figure 17: carton picture of grid information of Volkov's code.

One point should be kept in mind: there are two MAD operations in "single precision", one is "MAD dest, src1, src2, src3" and the other is "MAD dest, [smem], src2, src3" where dest, src1, src2 and src3 are registers but [smem] is shared memory. In Volkov's code, rank-1 update "rank1_update(A[0], &b[i][0], c);" can be expanded as

```
for( int j = 0 ; j < 16 ; j++){
        c[j] += A[0] * b[i][j] ;
}
```

and compiler ***nvcc*** translates " *c[j] += A[0] * b[i][j] ;*" into "MAD dest, [smem], src2, src3". This can be verified by ***decuda*** [2] and Volkov uses this MAD operation to analyze performance in [1].

2.2 comparison between CUBLAS and Volkov's code

In this work, we take Volkov's code as baseline and compare our method with it because our method is based on Volkov's code. First we compare performance of Volkov's code with CUBLAS in CUDA 2.3. To show comparison, we report two numbers, one is Gflop/s and the other is performance improvement.

Definition 1: in Volkov’s code, only MADs contribute to the flop count, so we define flop count as

$$Gflop/s=\frac{mnk\left(\#\ of\ \ MAD\right)\times 2\left(add+mul\right)}{time\left(s\right)}$$

Definition 2: let $\boldsymbol{R}$ be ratio of Gflop/s, defined by $R=\frac{\text{time of Volkov's code}}{\text{time of cublas}}=\frac{Gflop\ \text{ of cublas}}{Gflop\ \text{of Volkov's code}}$, then $1-R$ is performance improvement.

Figure 18 shows Gflop/s over $N=5:4096$ and Table 4 shows Gflop/s on specific dimension N . Generally speaking, Volkov's code has better performance than CUBLAS, Gflop/s of Volkov's code concentrates on $270\sim 350$ where Gflop/s of CUBLAS locates at $160\sim 350$. In fact, CUBLAS is faster a little bit than Volkov's code when $N=64k$.

N	CUBLAS (Gflop/s)	Volkov (Gflop/s)	***R***
256	198.92	207.67	0.9563
512	222.59	226.90	0.9810
1024	281.39	274.60	1.0248
2048	331.03	324.94	1.0187
4096	344.15	342.65	1.0044

Table 4: comparison between CUBLAS and Volkov's code for N = 256, 512, 1024, 2048, 4096 on TeslaC1060.

Hence it is reasonable to take Volkov's code as baseline since only 2% performance deviation between CUBLAS and Volkov's code even when *N* is multiple of 64. Moreover Gflop/s of Volkov's code is less than 350 Gflop/s whereas single precision performance without dual issue is 624 Gflop/s in TeslaC1060. This means that Volkov's code only reaches 56% of peak performance on TeslaC1060.

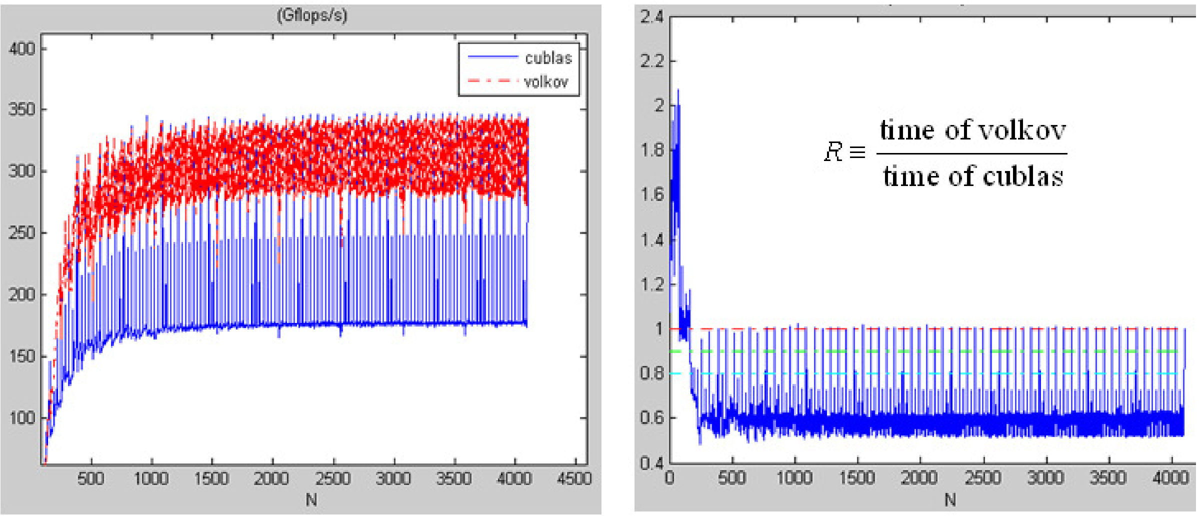


Figure 18: left panel is Gflop/s of CUBLAS and Volkov's code. Right panel is performance improvement. Platform is TeslaC1060.

3 basic idea

3.1 theoretical profile of Volkov's code

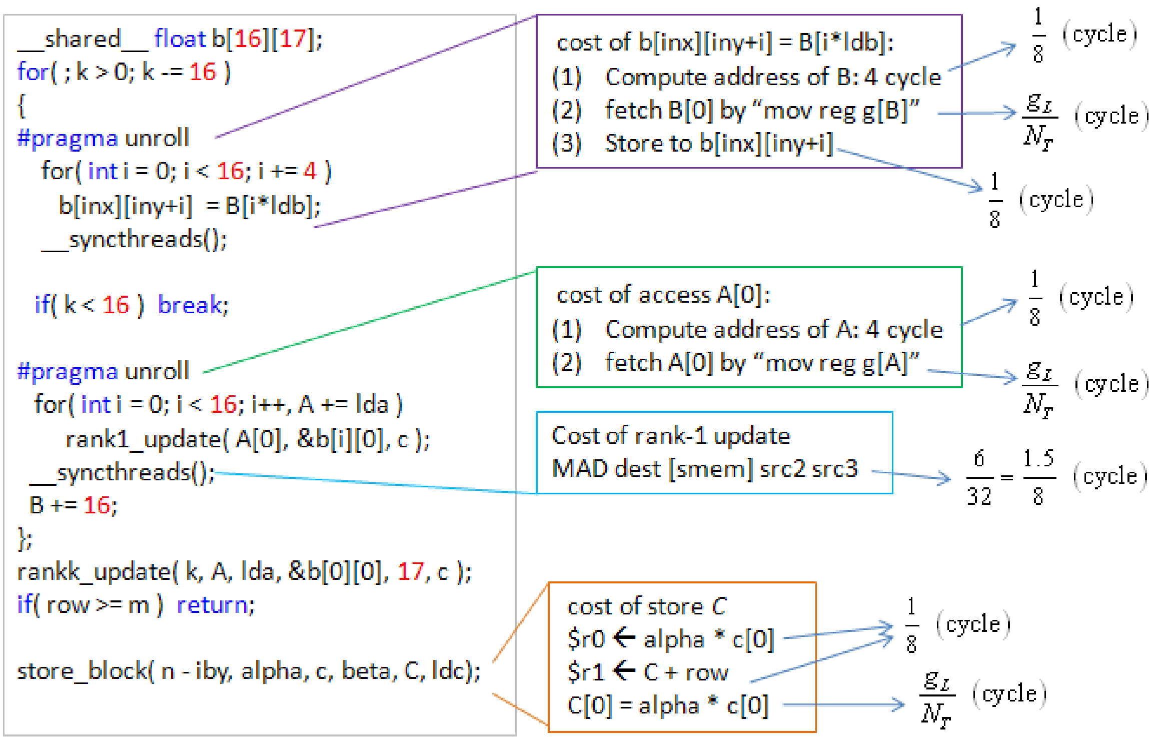
From discussion in section 1 and result of micro-benchmarking in [8], we can summarize latency and throughput of some instructions in Table 5.

Instruction	Type	Latency (cycles)	Throughput (cycles/warp)
MOV reg, [gmem]		530	4
MOV [smem], reg		36	4
ADD, SUB	Float	24	4
MAD dest, src1, src2, src3	Float	31.5	4
MAD dest, [smem], src2, src3	Float	34.6	6

Table 5: latency and throughput of arithmetic instructions on TeslaC1060.

Remark 5: authors in [8] use (ops/clock) as unit of throughput whereas we use (cycles/warp) as unit.

To estimate performance of Volkov's code without experiments, we need to check critical path in source code and decompose high level C code to basic instructions (one can use ***decuda*** to achieve this goal), then use throughput data in Table 5 to setup cost of instructions. Figure 19 samples critical path in Volkov's code and only leading order terms are considered. g_L is latency of global memory, in this work, we set $g_L = 530\ cycle$. N_T is number of active threads per SM, for example, $N_T = 512$ in Volkov's code. Furthermore we need several assumptions:



```
__shared__ float b[16][17];
for( ; k > 0; k -= 16 )
{
#pragma unroll
   for( int i = 0; i < 16; i += 4 )
      b[inx][iny+i]  = B[i*ldb];
   __syncthreads();

  if( k < 16 )  break;

#pragma unroll
  for( int i = 0; i < 16; i++, A += lda )
      rank1_update( A[0], &b[i][0], c );
  __syncthreads();
  B += 16;
};
rankk_update( k, A, lda, &b[0][0], 17, c );
if( row >= m )  return;

store_block( n - iby, alpha, c, beta, C, ldc);
```

cost of b[inx][iny+i] = B[i*ldb]:
(1) Compute address of B: 4 cycle
(2) fetch B[0] by "mov reg g[B]"
(3) Store to b[inx][iny+i]

cost of access A[0]:
(1) Compute address of A: 4 cycle
(2) fetch A[0] by "mov reg g[A]"

Cost of rank-1 update
MAD dest [smem] src2 src3

cost of store *C*
$r0 ← alpha * c[0]
$r1 ← C + row
C[0] = alpha * c[0]

$\frac{1}{8}$ (cycle)
$\frac{g_L}{N_T}$ (cycle)
$\frac{1}{8}$ (cycle)
$\frac{1}{8}$ (cycle)
$\frac{g_L}{N_T}$ (cycle)
$\frac{6}{32} = \frac{1.5}{8}$ (cycle)
$\frac{1}{8}$ (cycle)
$\frac{g_L}{N_T}$ (cycle)

Figure 19: timing estimate of Volkov's code

Assumption 1: number of active threads $\geqq 192$ such that pipeline latency of MOV, ADD, SUB, MUL can be hidden, then throughput of above instructions is 4 cycle/warp. Since source code is executed by threads, it is better to express throughput in terms of a thread, not a warp, so throughput of { MOV, ADD, SUB, MUL } is $\frac{4}{32} = \frac{1}{8}(cycle / thread)$.

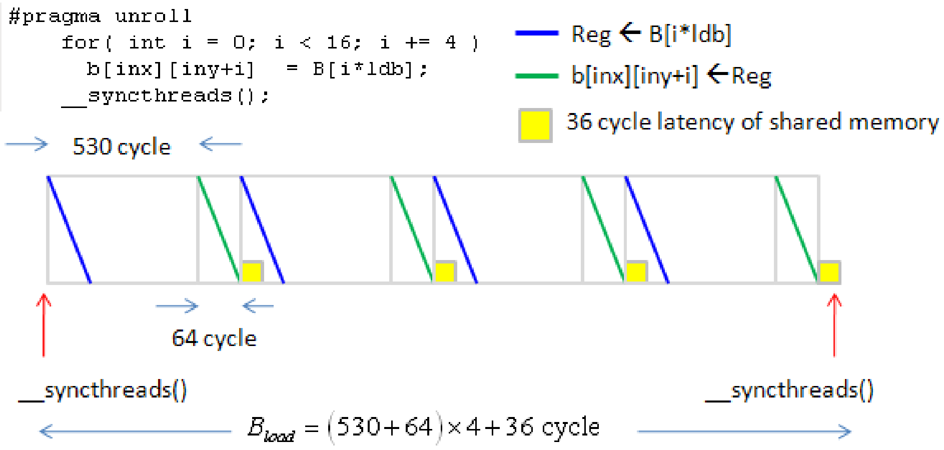
Assumption 2: number of threads > 256 such that latency of shared memory can be hidden. In Figure 6, number of active threads is set to be 512, then latency of shared memory is hidden by issuing time of consecutive command "*b[inx][iny+i] = Reg*". In such circumstance, we can ignore latency of shared memory and take throughput of "*b[inx][iny+i] = Reg*" into account. Under this assumption, total time of loading matrix *B* into shared memory *b[16][17]* is demonstrated in Figure 20.

Assumption 3: only leading order of critical path is considered.

Assumption 4: 30 SMs access independent 30 channels (in fact, only 8 physical channels exist, this assumption avoid contending for channels). This means we over-estimate bandwidth of global memory.

From profile in Figure 19, we can define four average quantities,

(1) $\overline{A}_{load} = \frac{g_L}{N_T} + \frac{1}{8}\left(cycle/thread\right)$ is cost of loading one element of matrix A into register, and total number of elements of A be accessed is $\frac{mnk}{b_N}$. Hence total time of loading A is $\frac{1}{N_{SM}}\frac{mnk}{b_N}\cdot\overline{A}_{load}$.



```
#pragma unroll
    for( int i = 0; i < 16; i += 4 )
      b[inx][iny+i]   = B[i*ldb];
    __syncthreads();
```

Figure 20:total time to load matrix B to shared memory $b[16][17]$

(2) $\overline{B}_{load} = \frac{g_L}{N_T} + \frac{2}{8}\left(cycle/thread\right)$ is cost of loading one element of matrix B into register, and total number of elements of B be accessed is $\frac{mnk}{b_M}$. Hence total time of loading B is $\frac{1}{N_{SM}}\frac{mnk}{b_M}\cdot\overline{B}_{load}$.

(3) $\overline{T}_{C=A\times B} = \frac{1.5}{8}\left(cycle/thread\right)$ is cost of "MAD dest, [smem], src2, src3" and total number of MADs is mnk, so total time of $C = A * B$ is $\frac{1}{N_{SM}}mnk\cdot\overline{T}_{C=A\times B}$

(4) $\overline{C}_{store} = \frac{g_L}{N_T} + \frac{2}{8}\left(cycle/thread\right)$ is cost of storing one element of matrix C, and total number of elements of C be accessed is mn. Hence total time of loading C is $\frac{1}{N_{SM}}mn\cdot\overline{C}_{store}$.

In this work, we focus on matrix-multiplication on square matrices, so $\frac{1}{N_{SM}}mn\cdot\overline{C}_{store}$ is not a leading order term. We can neglect it without affecting performance estimation. To sum up, performance of Volkov's code is calibrated by

$$\text{Total time of Volkov's code} = \frac{1}{N_{SM}}\left[\frac{mnk}{b_N}\cdot\overline{A}_{load} + \frac{mnk}{b_M}\cdot\overline{B}_{load} + mnk\cdot\overline{T}_{C=A\times B}\right]\cdot\frac{1}{core\ freq.}\ (\text{sec})$$

Example 1: consider Volkov's code on TeslaC1060. $N_{SM} = 30$, $N_T = 512$, $b_M = 64$, $b_N = 16$, then

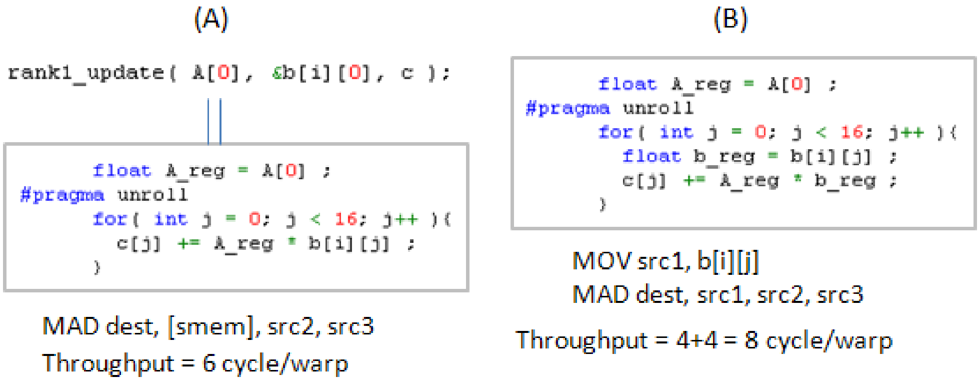
$\bar{A}_{load} = 1.16(cycle/thread)$, $\bar{B}_{load} = 0.02$, and $\bar{T}_{C=A\times B} = 0.1875(cycle/thread)$. Performance expressed in term of (Gflop/s) is $2\dfrac{mnk}{\text{total time of Volkov's code}} = \dfrac{2\cdot N_{SM}\cdot(1.3GHz)}{\dfrac{\bar{A}_{load}}{16}+\dfrac{\bar{B}_{load}}{64}+\bar{T}_{C=A\times B}} = 278.57\ \ (Gflop/s)$.

Remark 6: under profiling model, the more number of active threads per SM is, the smaller $\bar{A}_{load} = \dfrac{g_L}{N_T}+\dfrac{1}{8}$, $\bar{B}_{load} = \dfrac{g_L}{N_T}+\dfrac{2}{8}$, and $\bar{T}_{C=A\times B} = \dfrac{1.5}{8}$ are. However this property does not hold in Volkov's code. We change size of shared memory ***b*** in volkov's code to decrease number of active threads artificially and report Gflop/s in Table 6.

N	Volkov 512 threads	Volkov 384 threads	Volkov 320 threads	Volkov 256 threads	Volkov 192 threads
256	207.668	207.959	207.446	207.325	208.214
512	226.895	257.561	265.977	240.991	263.005
1024	274.597	295.442	304.576	307.883	301.340
2048	324.941	336.311	338.103	336.047	321.100
4096	342.651	343.713	343.595	341.188	325.226

Table 6: performance of Volkov's code under different active threads per SM on TeslaC1060. unit: Gflop/s.

Observation 1: it seems that 256 active threads are enough to hide memory latency and our profiling model cannot interpret this phenomenon. (data of 192 threads seems good for N = 256, 512, 1024, 2048, 4096, however it is 10% worse than data of 512 threads if one looks at graph of Gflop/s over N). This gives us a hint to speedup Volkov's code, we can focus $\bar{T}_{C=A\times B}$ first and skip effect of $\bar{A}_{load}$ and $\bar{B}_{load}$. In other words, if we have a good idea to accelerate $\bar{T}_{C=A\times B}$, then we have a chance to improve Volkov's code.



```
rank1_update( A[0], &b[i][0], c );
```

```
      float A_reg = A[0] ;
#pragma unroll
      for( int j = 0; j < 16; j++ ){
        c[j] += A_reg * b[i][j] ;
      }
```

MAD dest, [smem], src2, src3
Throughput = 6 cycle/warp

```
      float A_reg = A[0] ;
#pragma unroll
      for( int j = 0; j < 16; j++ ){
        float b_reg = b[i][j] ;
        c[j] += A_reg * b_reg ;
      }
```

MOV src1, b[i][j]
MAD dest, src1, src2, src3
Throughput = 4+4 = 8 cycle/warp

Figure 21: (A) rank-1 update in Volkov's code, use "MAD dest, [smem], src2, src3"
(B) change "MAD dest, [smem], src2, src3" to "MAD dest, src1, src2, src3"

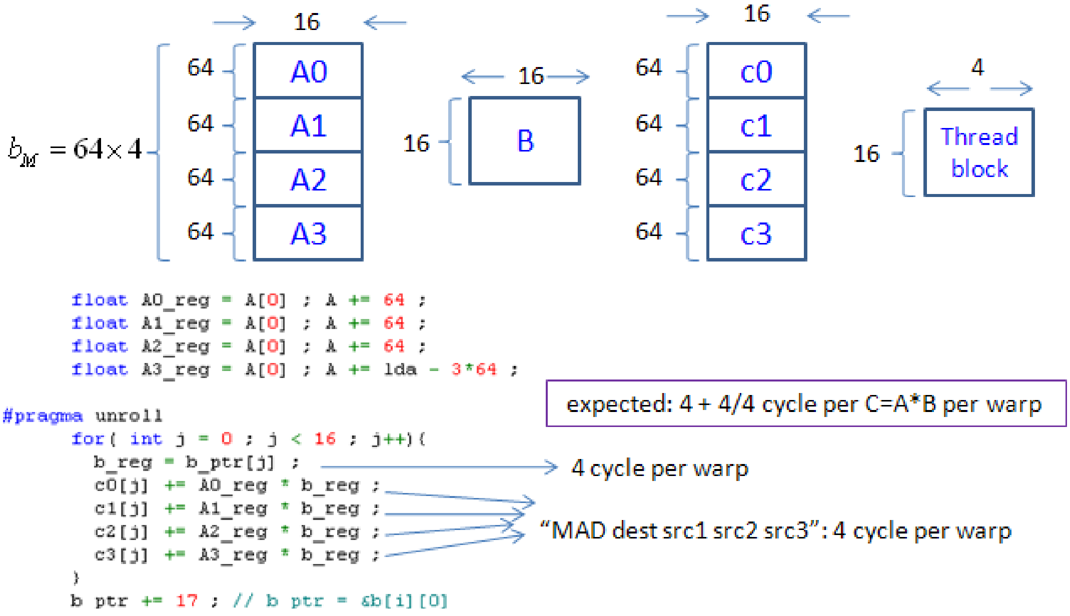
3.2 "MAD dest [smem] src2 src3" to "MAD dest src1 src2 src3"

In Figure 21 we re-write rank-1 update of volkov's code as explicit for-loop in (A), compiler ***nvcc*** translates "*c[j] += A_reg * b[i][j]*" to "MAD dest, [smem], src2, src3" whose throughput is 6 cycle per warp. However intuitive translation of "*c[j] += A_reg * b[i][j]*" is two-step form in (B):

moving *b[i][j]* to register *b_reg* followed by "MAD dest, src1, src2, src3"

This representation has no benefit since throughput of MOV is 4 cycle per warp, the same as throughput of "MAD dest, src1, src2, src3", and then total throughput is 4+4 = 8 cycle per warp. Therefore ***nvcc*** does a good job in Volkov's code since it saves 2 cycle per warp with respect to intuitive translation.

Unfortunately, "MAD dest, [smem], src2, src3" is not good because it runs at 66% of the peak on all GPUs. This means that the best performance we may expect is 66 % of peak rate. On TeslaC1060, peak rate without dual issue is 624 Gflop/s. Even Volkov's code reaches 342.651 Gflop/s in Table 6, it is still far from optimal value, 624 * 0.66 = 411.8 Gflop/s.



```
        float A0_reg = A[0] ; A += 64 ;
        float A1_reg = A[0] ; A += 64 ;
        float A2_reg = A[0] ; A += 64 ;
        float A3_reg = A[0] ; A += lda - 3*64 ;

#pragma unroll
        for( int j = 0 ; j < 16 ; j++){
          b_reg = b_ptr[j] ;
          c0[j] += A0_reg * b_reg ;
          c1[j] += A1_reg * b_reg ;
          c2[j] += A2_reg * b_reg ;
          c3[j] += A3_reg * b_reg ;
        }
        b ptr += 17 ; // b ptr = &b[i][0]
```

Figure 22: throughput decreases to 5 cycle per warp per C = A*B, 16% improvement.

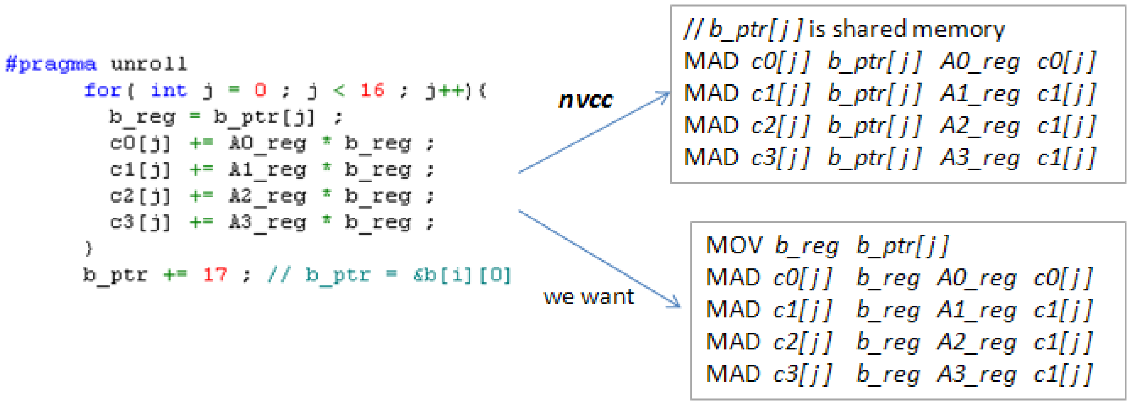
The key point is to decrease number of access of shared memory *b[i][j]*, in other words, keep *b[i][j* into register longer and longer. This could be achieved if we move *b[i][j]* into register and serve more vectors in matrix *A*. Figure 22 shows whole idea: vector length is still 64, but we load four consecutive vectors of matrix *A* into registers, then one *b[i][j* in register *b_reg* can serve four elements of *A* in registers, called *A0_reg*, *A1_reg*, *A2_reg* and *A3_reg*, then four "MAD dest, src1, src2, src3" operations are executed. Averagely speaking, one "MAD dest, src1, src2, src3" consumes $\frac{1}{4}$ "MOV reg, [smem]", so throughput of "*c[j] += A_reg * b[i][j]*" becomes

$4(MAD\ \text{dest,src1,src2,src3}) + \frac{1}{4} \times 4(MOV\ \text{reg, [smem]}) = 5\ (cycle/warp)$ which saves $\frac{1}{6}$ flop counts. If only

$\bar{T}_{C=A\times B}$ is considered, then we expect 16% improvement when using algorithm in Figure 22.

3.3 difficulty: how to achieve "MAD dest, src1, src2, src3"

The road to happiness is strewn with setbacks. In Figure 23 compiler ***nvcc*** always replaces *b_reg* by *b_ptr[j]* and uses "MAD dest, [smem], src2, src3" to do translation. We have tried setting variable *b_reg* as "volatile" but it does not work. Thanks to Wladimir J. van der Laan and Sylvain Collange, we can use ***decuda/cudasm*** to achieve this goal and we will discuss this workaround later.



```
#pragma unroll
      for( int j = 0 ; j < 16 ; j++){
        b_reg = b_ptr[j] ;
        c0[j] += A0_reg * b_reg ;
        c1[j] += A1_reg * b_reg ;
        c2[j] += A2_reg * b_reg ;
        c3[j] += A3_reg * b_reg ;
      }
      b_ptr += 17 ; // b_ptr = &b[i][0]
```

```
// b_ptr[ j ] is shared memory
MAD c0[j] b_ptr[j] A0_reg c0[j]
MAD c1[j] b_ptr[j] A1_reg c1[j]
MAD c2[j] b_ptr[j] A2_reg c1[j]
MAD c3[j] b_ptr[j] A3_reg c1[j]
```

```
MOV b_reg b_ptr[j]
MAD c0[j] b_reg A0_reg c0[j]
MAD c1[j] b_reg A1_reg c1[j]
MAD c2[j] b_reg A2_reg c1[j]
MAD c3[j] b_reg A3_reg c1[j]
```

Figure 23: ***nvcc*** replaces *b_reg* by *b_ptr[j]* and use "MAD dest, [smem], src2, src3". One can check this by ***decuda***.

3.4 algorithm volkov versus algorithm volkov_variant

To check consistency, we would like to change "MAD dest, [smem], src2, src3" in Volkov's code to "MAD dest, src1, src2, src3" and check the performance. Such variant of volkov's code is called algorithm **volkov_variant**, and Volkovs' code is called algorithm **volkov**. We keep silent on how to achieve this modification and focus on difference between algorithm **volkov** and algorithm **volkov_variant**.

Algorithm volkov_variant uses 32 registers per thread and has 512 active threads per SM, the same as algorithm volkov. We also modify size of shared memory to decrease number of active threads per SM and calibrate Gflop/s value in Table 7.

N	volkov 512 threads	volkov_variant 512 threads	volkov_variant 384 threads	volkov_variant 320 threads	volkov_variant 256 threads	volkov_variant 192 threads
256	207.668	157.290	157.469	157.246	158.149	157.359
512	226.895	206.597	235.508	250.517	206.906	214.208
1024	274.597	273.237	290.352	288.885	272.905	226.370
2048	324.941	326.253	333.650	325.749	302.861	238.335
4096	342.651	353.234	350.770	340.168	308.491	240.456

Table 7: performance of algorithm volkov_variant under different active threads per SM on TeslaC1060. unit: Gflop/s

Question 1: previous discussion reveals that algorithm volkov_variant is 33% slower than algorithm volkov because "*c[j] += A_reg * b[i][j]*" needs 6 cycle/warp in algorithm volkov whereas "*c[j] += A_reg * b[i][j]*" needs 4+4=8 cycle per warp in algorithm volkov_variant. However experimental result does not support this argument, in Figure 24, volkov_variant is 4% slower than volkov in worse case. How to interpret this phenomenon?

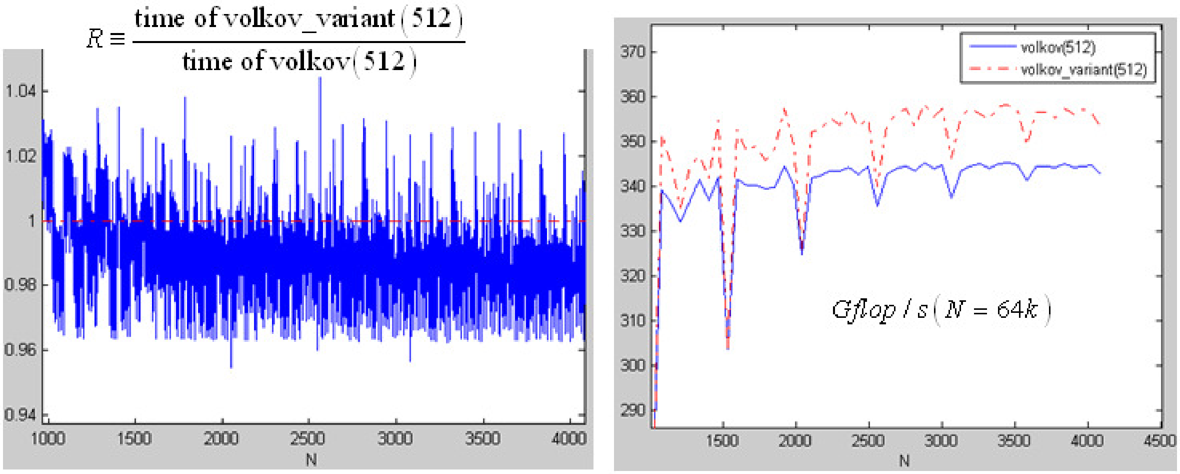


Figure 24: comparison between algorithm volkov and algorithm volkov_variant, Generally speaking, volkov_variant is not bad, this is different from expectation.

Observation 2: the fact, *volkov_variant* is not worse than *volkov*, gives us a hope that our idea may reach more performance improvement than expectation. Second if we want to keep algorithm *volkov_variant* comparable with algorithm *volkov*, then number of active threads should be not less than 320. This is very important since our method use more registers than volkov's code and then number of active threads would be less than 512.

4 method 1

Our first proposed algorithm, method 1, has same block information as that of algorithm *volkov* (Volkov's code) except two vectors (length = 64 per vector) share one movement of shared memory *b[i][j]*. Method 1 is an elementary version of algorithm in Figure 22 but uses fewer resources such that number of active threads can be kept higher (from Observation 2, we suggest number of active number per SM should be not less than 320).

In order to connect source code and this document, parameters of block information are introduced in Figure 25.

Resource usage in method 1 is

(1) two registers *A0_reg* and *A1_reg* hold elements of two vectors in $A_{bm,bk}$ respectively.

(2) shared memory *b[16][17]* is $B_{bk,bn}$

(3) two register's arrays, *c0[16]* and *c1[16]*, correspond to two consecutive sub-block $C_{bm,bn}$.

One more register's array *c1[16]* lead to high register count, 47, such that number of active threads is 320, less than 512 in algorithm *volkov*. However 320 active threads still satisfy threshold suggested by Observation 2.

4.1 performance of method 1

Source code of "method 1" is **method1.cu** and code segment of rank-1 update is shown in Figure 26. As we have demonstrated, compiler **nvcc** always uses "MAD dest, [smem], src2, src3" in such circumstance. As far as $\overline{T}_{C=A\times B}$ is concerned, method 1 has no advantage than algorithm *volkov*. However experimental result in Figure 27 shows good performance of method 1, the best performance improvement can be more than 10%.

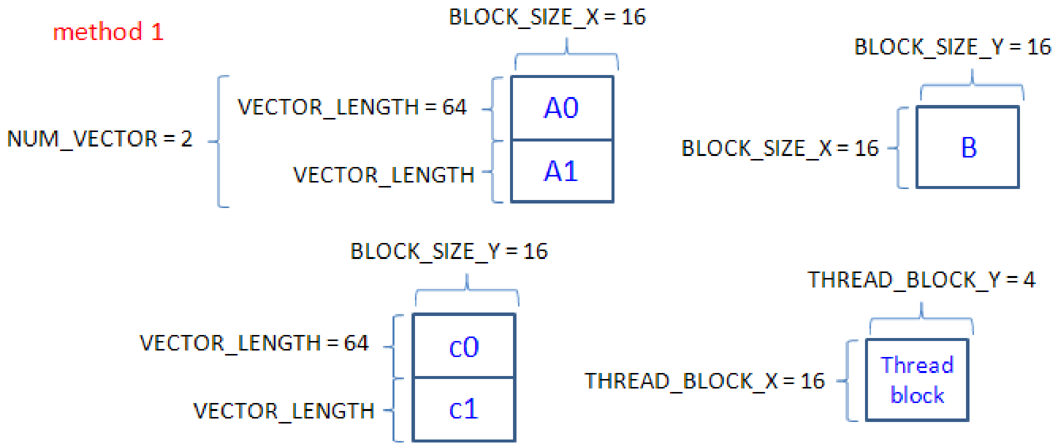
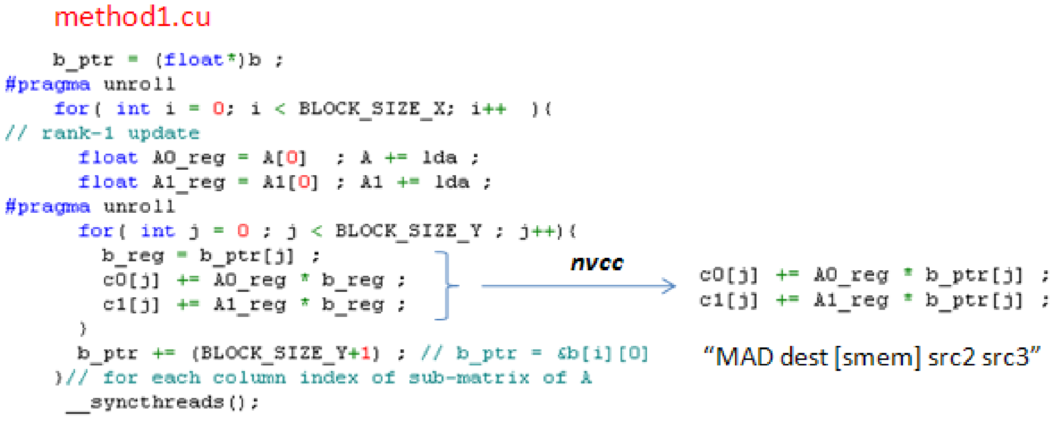


Figure 25: block information of method 1 and its variant.



method1.cu

```
    b_ptr = (float*)b ;
#pragma unroll
    for( int i = 0; i < BLOCK_SIZE_X; i++  ){
// rank-1 update
       float A0_reg = A[0]  ; A += lda ;
       float A1_reg = A1[0] ; A1 += lda ;
#pragma unroll
       for( int j = 0 ; j < BLOCK_SIZE_Y ; j++){
         b_reg = b_ptr[j] ;
         c0[j] += A0_reg * b_reg ;
         c1[j] += A1_reg * b_reg ;
       }
       b_ptr += (BLOCK_SIZE_Y+1) ; // b_ptr = &b[i][0]
    }// for each column index of sub-matrix of A
     __syncthreads();
```

```
c0[j] += A0_reg * b_ptr[j] ;
c1[j] += A1_reg * b_ptr[j] ;
```

Figure 26: ***nvcc*** uses two "MAD dest, [smem], src2, src3" in method 1. Basically method1 is the same as algorithm *volkov*.

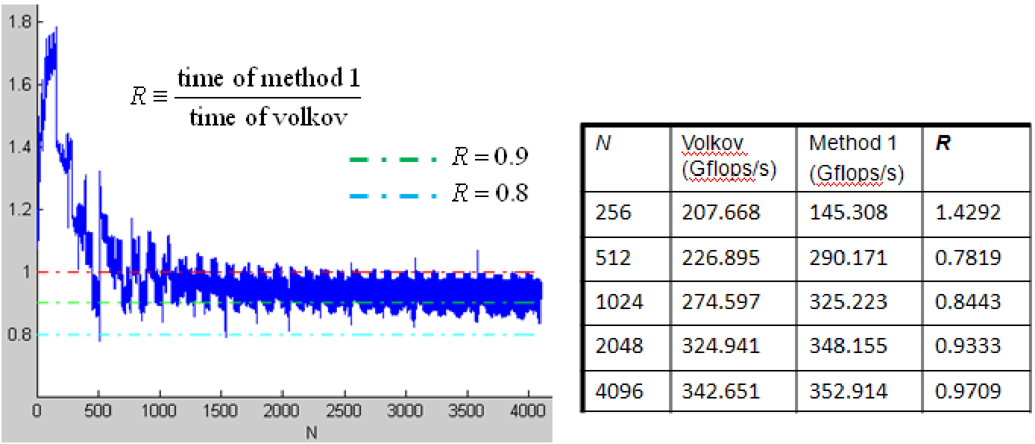
4.2 implementation of method1_variant

To achieve compilation "MAD dest, src1, src2, src3", we propose a workaround, which requires **decuda/cudasm**. In this section we will demonstrate this workaround in detail. We call this workaround as algorithm ***method1_variant***, and its source code locates in /method1/method1_variant.cu. Figure 28 shows rank-1 update of ***method1_variant***, roughly speaking, we have three steps

Step 1: modify "b_reg = b_ptr[j] ;" in algorithm ***volkov*** to "b_reg = b_ptr[j] * 4.0f ;", then ***nvcc*** has no choice but translates "b_reg = b_ptr[j] * 4.0f ;" to multiplication operation, "MUL b_reg, b_ptr[j], 4.0f ;", which can be verified by ***decuda***.

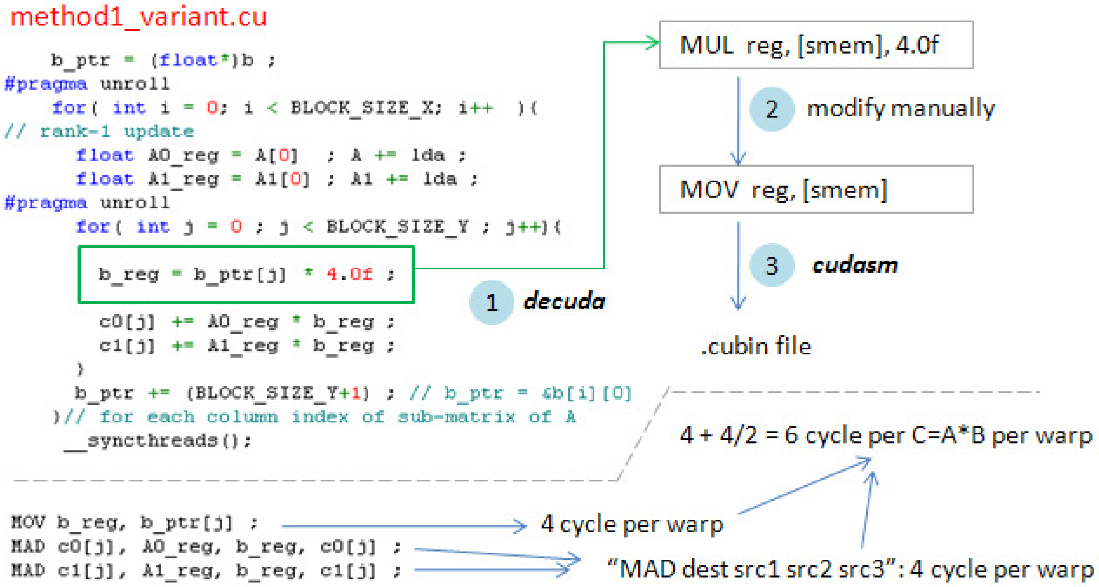
Step 2: in assembly file *method1_variant.asm* generated by **decuda**, we modify "MUL b_reg, b_ptr[j], 4.0f ;" to "MOV b_reg, b_ptr[j] ;" via any text editor.

Step 3: assembly modified *method1_variant.asm* again by ***cudasm***. Then the final binary .cubin file achieves our goal.



N	Volkov (Gflops/s)	Method 1 (Gflops/s)	***R***
256	207.668	145.308	1.4292
512	226.895	290.171	0.7819
1024	274.597	325.223	0.8443
2048	324.941	348.155	0.9333
4096	342.651	352.914	0.9709

Figure 27: performance of method 1. ***R*** ranges from 0.9 to 1.0, so method 1 is good.



```
    b_ptr = (float*)b ;
#pragma unroll
    for( int i = 0; i < BLOCK_SIZE_X; i++  ){
// rank-1 update
      float A0_reg = A[0]  ; A += lda ;
      float A1_reg = A1[0] ; A1 += lda ;
#pragma unroll
      for( int j = 0 ; j < BLOCK_SIZE_Y ; j++){
        b_reg = b_ptr[j] * 4.0f ;
        c0[j] += A0_reg * b_reg ;
        c1[j] += A1_reg * b_reg ;
      }
      b_ptr += (BLOCK_SIZE_Y+1) ; // b_ptr = &b[i][0]
    }// for each column index of sub-matrix of A
     __syncthreads();
```

```
MOV b_reg, b_ptr[j] ;
MAD c0[j], A0_reg, b_reg, c0[j] ;
MAD c1[j], A1_reg, b_reg, c1[j] ;
```

Figure 28: rank-1 update of algorithm mathod1_variant. ***nvcc*** translates "b_reg = b_ptr[j] * 4.0f " to "MUL reg, [smem], 4.0f ", then we can modify this MUL operation to "MOV reg, [smem]". Then two "MAD dest, src1, src2, src3" share one "MOV reg, [smem]".

Before describing three steps above precisely, we can sum up cost of data transfer and float-point cost among four algorithms, volkov, volkov_variant, method1, and method1_variant in Table 8. Please note that

(1) current GPU does not support direct data transfer between global memory and shared memory, hence "load *B* to shared memory *b*" must be decomposed as two steps, load *B* to register first and then store value of register to shared

memory.

(2) ***nvcc*** translates MAD operation to "MAD dest, [smem], src2, src3" in *volkov* and method 1, hence number of "MAD dest, [smem], src2, src3" is mnk. But

(3) we use such a trick "b_reg = b_ptr[j] * 4.0f " that "MAD dest, [smem], src2, src3" no long exists in *volkov_variant* and *method1_variant*. Instead we have "MAD dest, src1, src2, src3" and additional penalty, "MOV reg, [smem]". The main difference between *volkov_variant* and *method1_variant* is number of "MOV reg, [smem]". In *method1_variant*, two MAD operations share one load of shared memory, hence number of "MOV reg, [smem]" is $\frac{1}{2}mnk$.

operation	volkov	volkov_variant	method1	method1_variant
Load *A* to register "MOV reg, [gmem]"	$\frac{1}{16}mnk$	$\frac{1}{16}mnk$	$\frac{1}{16}mnk$	$\frac{1}{16}mnk$
Load *B* to shared memory *b* *1.*"MOV reg, [gmem]" *2.*"MOV [smem], reg"	$\frac{1}{64}mnk$	$\frac{1}{64}mnk$	$\frac{1}{64\times2}mnk$	$\frac{1}{64\times2}mnk$
Load shared memory *b* to register "MOV reg, [smem]"	0	mnk	0	$\frac{1}{2}mnk$
MAD dest, [smem], src2, src3	mnk	0	mnk	0
MAD dest, src1, src2, src3	0	mnk	0	mnk

Table 8: cost among four algorithms, volkov, volkov_variant, method1, method1_variant.

Observation 3: theoretically speaking, *method1_variant* has no benefit than *volkov* if only $\overline{T}_{C=A\times B}$ is considered because average cost of one "*c* += *a* * *b*" is 4+4/2 = 6 cycle per warp, which is the same as "MAD dest, [smem], src2, src3" in *volkov*. However we have showed performance of method 1 is better than *volkov* in Figure 27. We may expect good result of in Figure 27.

Next we elaborate how to accomplish the workaround step-by-step. The package ***decuda/cudasm*** is kept going and does not guarantee correctness for any combination of instructions, we must extract minimum region of binary code needed to be modified and keep remaining binary code unchanged. Generic procedure is depicted in Figure 29. binary file of method1_variant, method1_varinat.cubin, is divided into three parts, Y, X, Z where X contains code "*b_reg = b_ptr[j] * 4.0f* " which must be corrected as "MOV reg, [smem]" in assembly file. Then we edit part X in .asm file generated by ***decuda***, modify "MUL reg, [smem], 4.0f " to "MOV reg, [smem]", say assembly code segment X is modified as W. Then we use ***cudasm*** to assembly modified .asm file into binary file but replace binary code segments $\tilde{Y}, \tilde{Z}$ to original binary code segments Y, Z respectively.

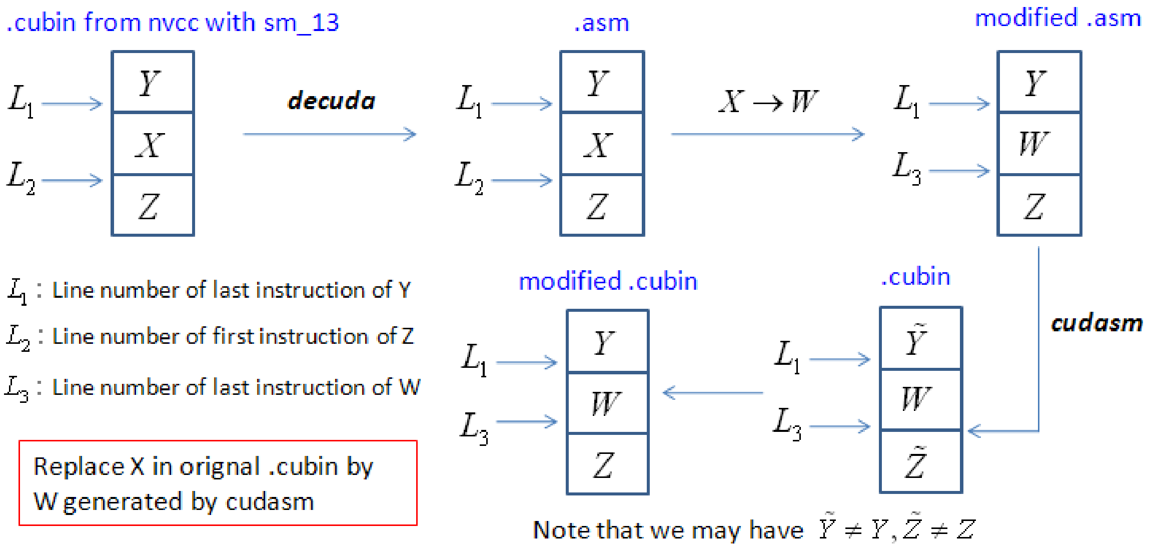

.cubin from nvcc with sm_13
L_1
L_2
Y
X
Z
decuda
.asm
L_1
L_2
Y
X
Z
$X \rightarrow W$
modified .asm
L_1
L_3
Y
W
Z
cudasm
L_1 : Line number of last instruction of Y
L_2 : Line number of first instruction of Z
L_3 : Line number of last instruction of W
Replace X in orignal .cubin by W generated by cudasm
modified .cubin
L_1
L_3
Y
W
Z
.cubin
L_1
L_3
$\tilde{Y}$
W
$\tilde{Z}$
Note that we may have $\tilde{Y} \neq Y, \tilde{Z} \neq Z$

Figure 29: generic procedure of the workaround in method1_variant.

Question 2: why do we use the trick "*b_reg = b_ptr[j] * 4.0f* "? Why not edit assembly directly?

Answer: I would like to write assembly directly in matrix-multiplication because it's structure is regular, programmers can control register usage themselves very well. However implementation of **cudasm** is not entirely complete, it is not a good idea to write whole assembly manually and rely on **cudasm**. Moreover all what we need is to compile *b_reg = b_ptr[j]* " to "MOV reg, [smem]", just only 256 lines in Volkov's code. So we can isolate minimum region containing these 256 lines and check validity of **cudasm** on this region. Another point must be mentioned, if we isolate a small code segment as our target and modify it such that code size is changed, then we have a serious problem on branch instructions after this code segment. We must update offset in these branch instructions. However if we use current trick, then we just need to modify "b_reg = b_ptr[j] * 4.0f " to "MOV reg, [smem]" in assembly code (256 lines) and code size does not change, that's why we use this trick.

To accomplish above generic procedures, we do the following steps:

Step 1: if ψ is binary code, then $cudasm\left(decuda\left(\psi\right)\right) \neq \psi$. We use ***cudasm*** to assembly .asm file generated by

decuda, then four errors occur, see Figure 30,

(1) Error on line 937: Invalid argument types

(2) Error on line 938: Invalid argument types

(3) Error on line 998: Type conflict -- expected half register

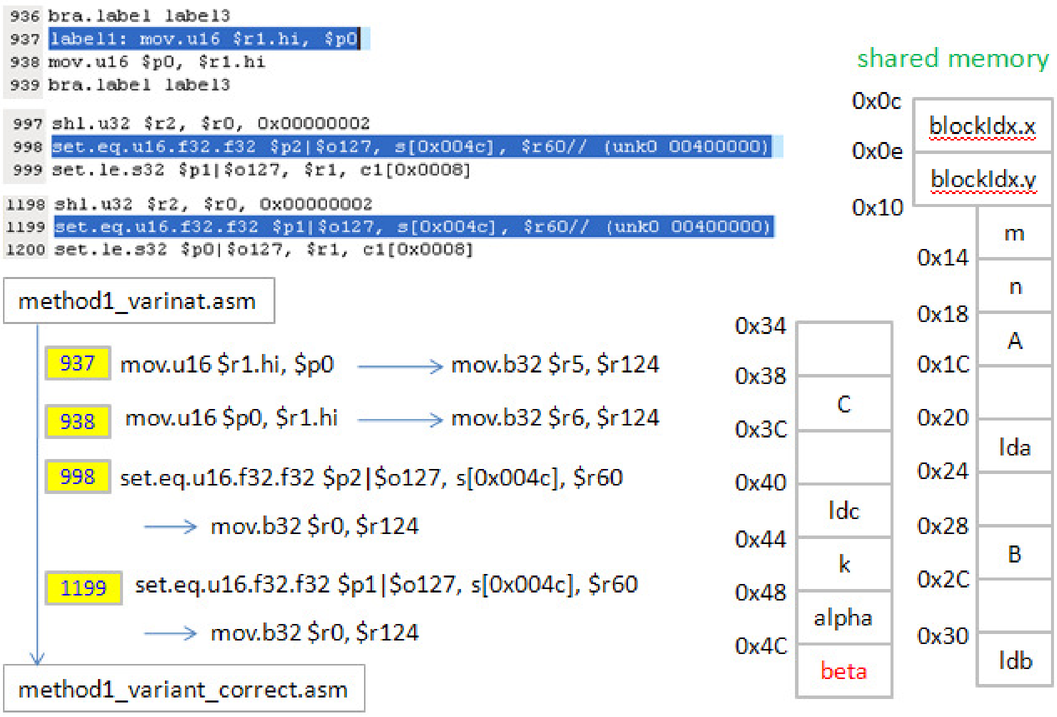
(4) Error on line 1199: Type conflict -- expected half register

Fortunately line 937, 938, 998, 1199 are not in code segment X in Figure 29, we can change these four instructions to any valid instructions (for example, line 937 "mov.u16 $r1.hi, $p0" is substituted by " mov.b32 $r5, $r124"). We save these changes into new assembly file, called method1_variant_correct.asm.

Remark 7: line 998 and line 1199 relate to instruction "if (beta == 0)" in *method1_variant.cu* because shared memory s[0x004c] is input parameter *beta*. However we have no idea about error message " Invalid argument types " at line 937 and line 938.

Step 2: in order to decompose binary files into three part Y, X, Z, we need to count line number L_1, L_2. For simplicity header of .cubin is removed and keep binary code only, and save binary code into file *from_nvcc.cubin*, as you see in Figure 31. Same procedure is done for .asm file, see Figure 32.

Remark 8: there are two instructions, one is 32-bit (operator has field "half" in assembly code, see Figure 33) and the other is 64-bit. However 32-bit instruction must appear in couple, in other words, length of binary code of one 32-bit instruction pair is the same as length of one 64-bit instruction. We can easily check number of instructions in assembly file and number of line is binary file. In assembly file, from_decuda.asm, there are four 32-bit instruction pairs (line 61 and 62, line 821 and 822, line 873 and 874, line 975 and 976), and number of assembly code is 1391. Hence number of 64-bit instruction is 1391 - 4 = 1387. Total line in *from_nvcc.bin* is $\left\lceil \frac{\text{\# of 64-bit instruction}}{2} \right\rceil = \left\lceil \frac{1387}{2} \right\rceil = 694$ because one line in *from_nvcc.cubin* contains two 64-bit instruction.



```
936 bra.label label3
937 label1: mov.u16 $r1.hi, $p0
938 mov.u16 $p0, $r1.hi
939 bra.label label3

997 shl.u32 $r2, $r0, 0x00000002
998 set.eq.u16.f32.f32 $p2|$o127, s[0x004c], $r60// (unk0 00400000)
999 set.le.s32 $p1|$o127, $r1, c1[0x0008]

1198 shl.u32 $r2, $r0, 0x00000002
1199 set.eq.u16.f32.f32 $p1|$o127, s[0x004c], $r60// (unk0 00400000)
1200 set.le.s32 $p0|$o127, $r1, c1[0x0008]
```

method1_varinat.asm

937 mov.u16 $r1.hi, $p0 → mov.b32 $r5, $r124

938 mov.u16 $p0, $r1.hi → mov.b32 $r6, $r124

998 set.eq.u16.f32.f32 $p2|$o127, s[0x004c], $r60 → mov.b32 $r0, $r124

1199 set.eq.u16.f32.f32 $p1|$o127, s[0x004c], $r60 → mov.b32 $r0, $r124

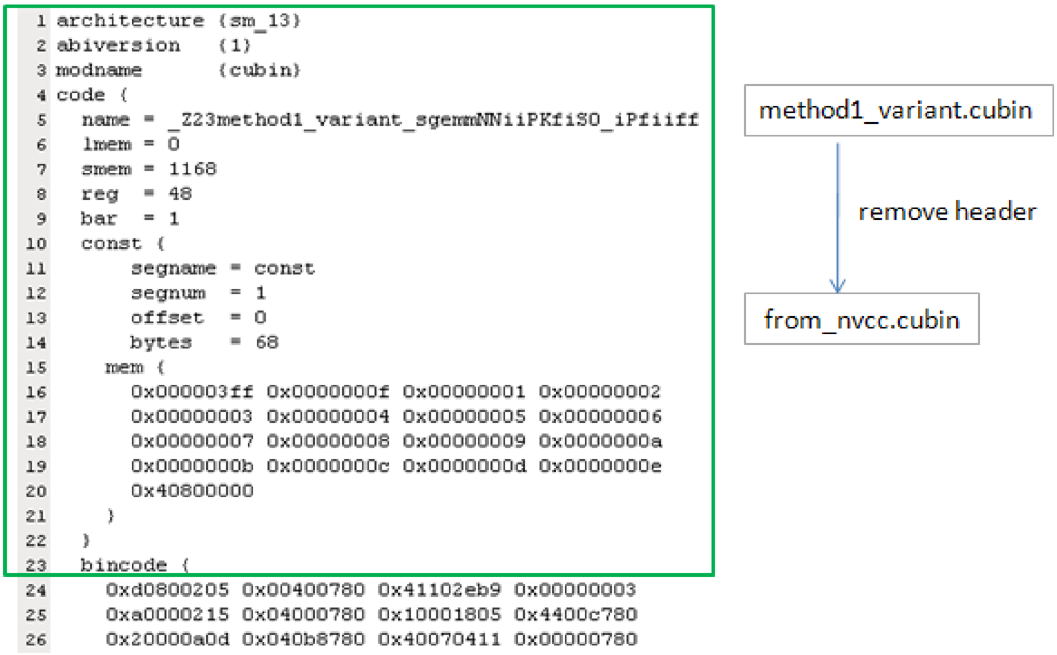
method1_variant_correct.asm

Figure 30: consistent check between ***decuda*** and ***cudasm***.

Step 3: In order to decompose *from_nvcc.cubin* and *from_decuda.asm* into three parts Y, X, Z, we use synchronization command as a landmark. In Figure 34 two synchronization commands, __synchthreads(), correspond to assembly code "bar.synch.u32 0x00000000" in line 78 and line 923 in file *from_decuda.asm* respectively (binary code of __syncthread() is "0x861ffe03 0x00000000"). First assembly code of "*breg = b_ptr[j] * 4.0f*" is "mul.rn.f32 $r47, s[0x050], c1[0x0040]" in line 83 of *from_decuda.asm* (one can check that shared memory *b[16][17]* starts at location 0x50 and c1[0x0040] = 0x40800000 which is 4.0f). Hence code segment X must contain line 84 of *from_decuda.asm*. Under consideration of 32-bit instruction pairs, line 83 of *from_decuda.asm* corresponds to 82-th 64-bit instruction, line 41 in *from_nvcc.cubin*, so we can choose binary code segment Y to be line 1 ~ 41 and store it into *from_nvcc_1_41.cubin*. Similarly the last assembly code of "*breg = b_ptr[j] * 4.0f*" is "mul.rn.f32 $r41, s[4ofs3+0x0008], 0x40800000" in line 916 of *from_decuda.asm*. So we can put assembly codes after line 916 into code segment Z. Here line 917 of *from_decuda.asm* corresponds to second 64-bit binary code in line 457 of *from_nvcc.cubin,* we choose binary code segment Z to be line 458~694 and store it into *from_nvcc_458_694.cubin*. To sum up, the result of decomposition is listed in Table 9.

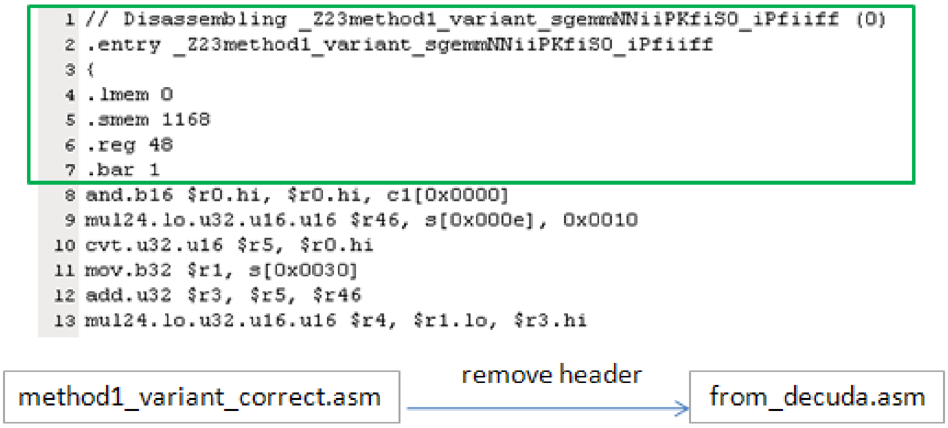
Code segment	from_nvcc.cubin	from_decuda.asm
Y	from_nvcc_1_41.cubin: line 1 ~ 41	from_decuda_1_83.asm: line 1 ~ 83
X	from_nvcc_42_457.cubin: line 42 ~ 457	from_decuda_84_917.asm: line 84 ~ 917
Z	from_nvcc_458_694.cubin: line 458~694	from_decuda_918_1391.asm: line 918~1391

Table 9: decompose binary file *from_nvcc.cubin* and assembly file *from_decuda.asm* into three parts, Y, X, Z according to landmarks in Figure 34 .



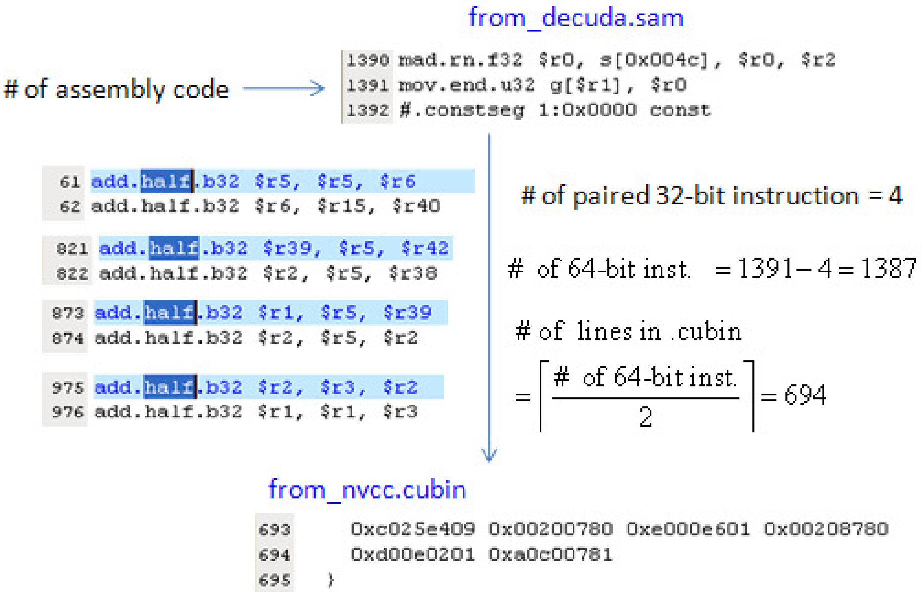
```
architecture (sm_13)
abiversion   (1)
modname      (cubin)
code {
  name = _Z23method1_variant_sgemmNNiiPKfiS0_iPfiiff
  lmem = 0
  smem = 1168
  reg  = 48
  bar  = 1
  const {
      segname = const
      segnum  = 1
      offset  = 0
      bytes   = 68
    mem {
      0x000003ff 0x0000000f 0x00000001 0x00000002
      0x00000003 0x00000004 0x00000005 0x00000006
      0x00000007 0x00000008 0x00000009 0x0000000a
      0x0000000b 0x0000000c 0x0000000d 0x0000000e
      0x40800000
    }
  }
  bincode {
    0xd0800205 0x00400780 0x41102eb9 0x00000003
    0xa0000215 0x04000780 0x10001805 0x4400c780
    0x20000a0d 0x040b8780 0x40070411 0x00000780
```

Figure 31: green box is header of .cubin file, remove it and store remaining part into from_nvcc.cubin



```
// Disassembling _Z23method1_variant_sgemmNNiiPKfiS0_iPfiiff (0)
.entry _Z23method1_variant_sgemmNNiiPKfiS0_iPfiiff
{
.lmem 0
.smem 1168
.reg 48
.bar 1
and.b16 $r0.hi, $r0.hi, c1[0x0000]
mul24.lo.u32.u16.u16 $r46, s[0x000e], 0x0010
cvt.u32.u16 $r5, $r0.hi
mov.b32 $r1, s[0x0030]
add.u32 $r3, $r5, $r46
mul24.lo.u32.u16.u16 $r4, $r1.lo, $r3.hi
```

Figure 32: green box is header of .asm file, remove it and store remaining part into from_decuda.asm



from_decuda.sam
# of assembly code

```
1390 mad.rn.f32 $r0, s[0x004c], $r0, $r2
1391 mov.end.u32 g[$r1], $r0
1392 #.constseg 1:0x0000 const
```

```
61 add.half.b32 $r5, $r5, $r6
62 add.half.b32 $r6, $r15, $r40

821 add.half.b32 $r39, $r5, $r42
822 add.half.b32 $r2, $r5, $r38

873 add.half.b32 $r1, $r5, $r39
874 add.half.b32 $r2, $r5, $r2

975 add.half.b32 $r2, $r3, $r2
976 add.half.b32 $r1, $r1, $r3
```

# of paired 32-bit instruction = 4
# of 64-bit inst. = 1391 − 4 = 1387
# of lines in .cubin
$= \left\lceil \frac{\#\text{ of 64-bit inst.}}{2} \right\rceil = 694$
from_nvcc.cubin

```
693      0xc025e409 0x00200780 0xe000e601 0x00208780
694      0xd00e0201 0xa0c00781
695    }
```

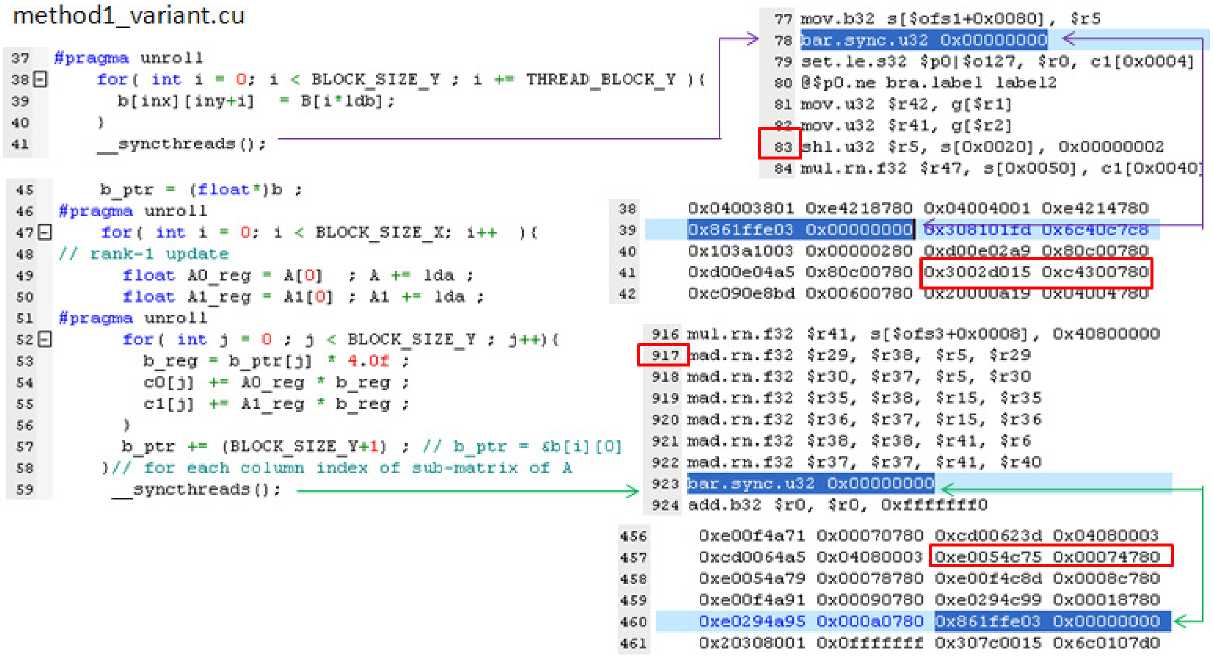
Figure 33: there are four 32-bit instruction pairs. Then number of assembly code is 1391 but number of lines in binary file is 694.

Step 4: replace "MUL reg, [smem], 4,0f" by "MOV reg, [smem]" in code segment X

In assembly file from_decuda_84_917.asm, we must modify code of "MUL reg, [smem], 4,0f" to "MOV reg, [smem]". There are four kinds of "MUL reg, [smem], 4,0f" listed in Table 10, we can do two steps to obtain "MOV reg, [smem]" and store corrected assembly code into new file from_decuda_84_917_ldsb32.asm.

(1) replace operator " mul.rn.f32" by "lds.b32" and

(2) remove four patterns ", c1[0x0040]", ", 0x40800000", ", c1[$ofs2+0x0040]" and ", c1[$ofs3+0x0040]"



method1_variant.cu

```
37  #pragma unroll
38      for( int i = 0; i < BLOCK_SIZE_Y ; i += THREAD_BLOCK_Y ){
39        b[inx][iny+i]  = B[i*ldb];
40      }
41      __syncthreads();

45      b_ptr = (float*)b ;
46  #pragma unroll
47      for( int i = 0; i < BLOCK_SIZE_X; i++  ){
48  // rank-1 update
49        float A0_reg = A[0]  ; A += lda ;
50        float A1_reg = A1[0] ; A1 += lda ;
51  #pragma unroll
52        for( int j = 0 ; j < BLOCK_SIZE_Y ; j++){
53          b_reg = b_ptr[j] * 4.0f ;
54          c0[j] += A0_reg * b_reg ;
55          c1[j] += A1_reg * b_reg ;
56        }
57        b_ptr += (BLOCK_SIZE_Y+1) ; // b_ptr = &b[i][0]
58      }// for each column index of sub-matrix of A
59       __syncthreads();
```

```
77 mov.b32 s[$ofs1+0x0080], $r5
78 bar.sync.u32 0x00000000
79 set.le.s32 $p0|$o127, $r0, c1[0x0004]
80 @$p0.ne bra.label label2
81 mov.u32 $r42, g[$r1]
82 mov.u32 $r41, g[$r2]
83 shl.u32 $r5, s[0x0020], 0x00000002
84 mul.rn.f32 $r47, s[0x0050], c1[0x0040]
```

```
38      0x04003801 0xe4218780 0x04004001 0xe4214780
39      0x861ffe03 0x00000000 0x308101fd 0x6c40c7c8
40      0x103a1003 0x00000280 0xd00e02a9 0x80c00780
41      0xd00e04a5 0x80c00780 0x3002d015 0xc4300780
42      0xc090e8bd 0x00600780 0x20000a19 0x04004780
```

```
916 mul.rn.f32 $r41, s[$ofs3+0x0008], 0x40800000
917 mad.rn.f32 $r29, $r38, $r5, $r29
918 mad.rn.f32 $r30, $r37, $r5, $r30
919 mad.rn.f32 $r35, $r38, $r15, $r35
920 mad.rn.f32 $r36, $r37, $r15, $r36
921 mad.rn.f32 $r38, $r38, $r41, $r6
922 mad.rn.f32 $r37, $r37, $r41, $r40
923 bar.sync.u32 0x00000000
924 add.b32 $r0, $r0, 0xfffffff0
```

```
456      0xe00f4a71 0x00070780 0xcd00623d 0x04080003
457      0xcd0064a5 0x04080003 0xe0054c75 0x00074780
458      0xe0054a79 0x00078780 0xe00f4c8d 0x0008c780
459      0xe00f4a91 0x00090780 0xe0294c99 0x00018780
460      0xe0294a95 0x000a0780 0x861ffe03 0x00000000
461      0x20308001 0x0fffffff 0x307c0015 0x6c0107d0
```

Figure 34: decompose from_nvcc.cubin and from_decuda.asm into three parts.

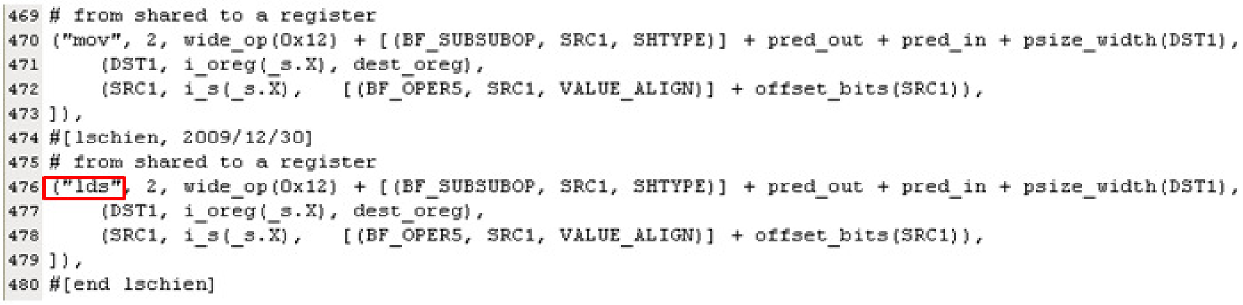
from_decuda_84_917.asm	from_decuda_84_917_ldsb32.asm
mul.rn.f32 $r47, s[0x0050], c1[0x0040]	lds.b32 $r47, s[0x0050]
mul.rn.f32 $r7, s[$ofs2+0x0000], 0x40800000	lds.b32 $r7, s[$ofs2+0x0000]
mul.rn.f32 $r40, s[$ofs2+0x0040], c1[$ofs2+0x0040]	lds.b32 $r40, s[$ofs2+0x0040]
mul.rn.f32 $r37, s[$ofs3+0x0044], c1[$ofs3+0x0040]	lds.b32 $r37, s[$ofs3+0x0044]

Table 10: modify form of "MUL reg, [smem], 4.0f" in first column into "MOV reg, [smem]" in second column.

Remark 9: There are two binary codes to do "MOV reg, [smem]", one corresponds to "mov.b32 reg, [smem]", the other corresponds to "lds.b32 reg, [smem]". Performance of both binary codes are the same in our experiment. In this work, we use "lds.b32 reg, [smem]" because cudasm translates it to same binary code as we obtain from ***nvcc***. However in order to translate "lds.b32 reg, [smem]", thanks to Sylvain Collange, you only need to add one rule

```
# from shared to a register
("lds", 2, wide_op(0x12) + [(BF_SUBSUBOP, SRC1, SHTYPE)] + pred_out +pred_in + psize_width(DST1), [
(DST1, i_oreg(_s.X), dest_oreg),
(SRC1, i_s(_s.X), [(BF_OPER5, SRC1, VALUE_ALIGN)] + offset_bits(SRC1)),
]),
```

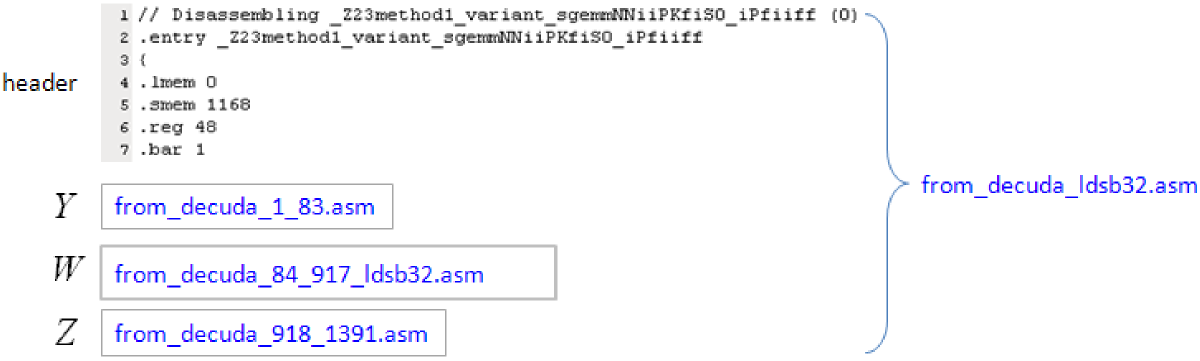
in AsmRules.py of decuda package. The modification is shown in Figure 35.



```
# from shared to a register
("mov", 2, wide_op(0x12) + [(BF_SUBSUBOP, SRC1, SHTYPE)] + pred_out + pred_in + psize_width(DST1),
    (DST1, i_oreg(_s.X), dest_oreg),
    (SRC1, i_s(_s.X),   [(BF_OPER5, SRC1, VALUE_ALIGN)] + offset_bits(SRC1)),
]),
#[lschien, 2009/12/30]
# from shared to a register
("lds", 2, wide_op(0x12) + [(BF_SUBSUBOP, SRC1, SHTYPE)] + pred_out + pred_in + psize_width(DST1),
    (DST1, i_oreg(_s.X), dest_oreg),
    (SRC1, i_s(_s.X),   [(BF_OPER5, SRC1, VALUE_ALIGN)] + offset_bits(SRC1)),
]),
#[end lschien]
```

Figure 35: add one rule in AsmRules.py, this rule can translate "lds.b32 reg, [smem]".

Step 5: Combine header of method1_variant.asm, fomr_decuda_1_83.asm, from_decuda_84_917_ldsb32.asm and from_decuda_918_1391.asm into one new file, called from_decuda_ldsb32.asm. Note that one must keep these files in order, see Figure 36. from_decuda_ldsb32.asm is a correct assembly file that can do matrix-multiplication because it replaces "MUL reg, [smem], 4,0f" by "MOV reg, [smem]".



```
// Disassembling _Z23method1_variant_sgemmNNiiPKfiS0_iPfiiff (0)
.entry _Z23method1_variant_sgemmNNiiPKfiS0_iPfiiff
{
.lmem 0
.smem 1168
.reg 48
.bar 1
```

Figure 36: combine header and code segments *Y, W, Z* into a new assembly file, which replaces "MUL reg, [smem], 4,0f" by "MOV reg, [smem]".

Step 6: translate from_decuda_ldsb32.asm into machine code, from_decuda_ldsb32_cudasm.cubin, via

```
cudasm -o   from_decuda_ldsb32_cudasm.cubin   from_decuda_ldsb32.asm
```

However machine code may not work because ***cudasm*** cannot guarantee consistency. Fortunately translation of code segment X (from_decuda_84_917_ldsb32.asm) is correct. so we can extract line 42 ~ 457 of from_decuda_ldsb32_cudasm.cubin to file from_decuda_ldsb32_cudasm_42_457.cubin. Finally we combine header of method1_variant.cubin with three code segments, from_nvcc_1_41.cubin, from_decuda_ldsb32_cudasm_42_457, and from_nvcc_458_694.cubin into final binary file, decuda_ldsb32_cudasm.cubin, which can be loaded into application by driver API. Order of combination of these files are shown in Figure 37.

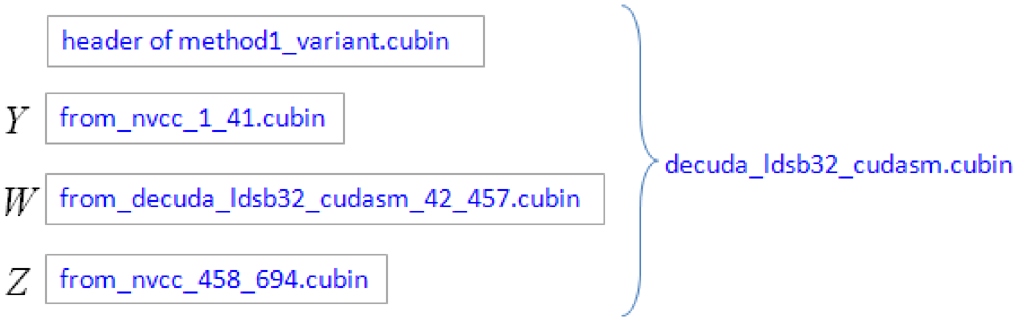
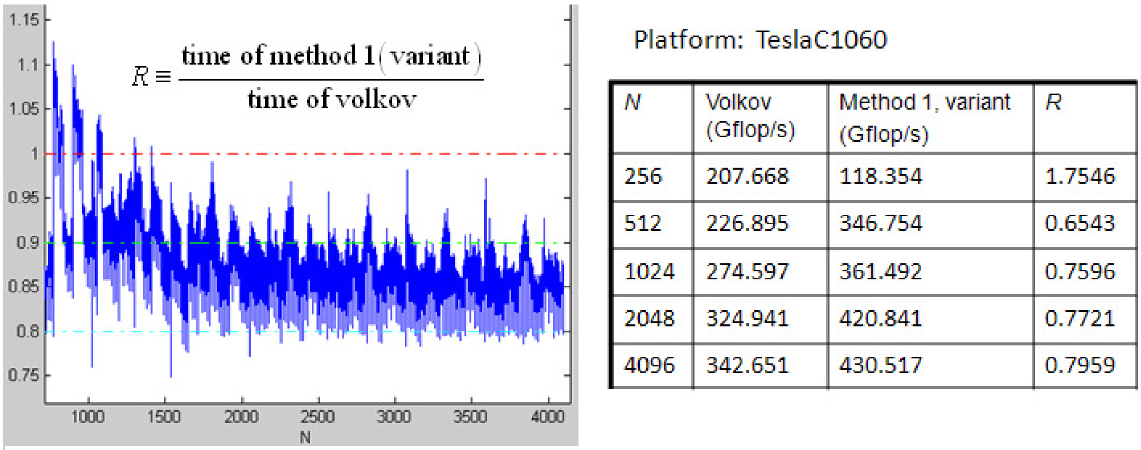

header of method1_variant.cubin
Y
from_nvcc_1_41.cubin
W
from_decuda_ldsb32_cudasm_42_457.cubin
Z
from_nvcc_458_694.cubin
decuda_ldsb32_cudasm.cubin

Figure 37: combine corrected binary code W with header, Y and Z to final binary file decuda_lds32_cudasm.cubin which can be loaded via driver API.

4.3 performance of method1_variant

Compared with performance of method 1 in Figure 27, the ratio R of method1_variant in Figure 38 is just R of method1 but shifts 0.1 downward. The table shows more than 20% performance improvement when dimension N = 512, 1024, 2048 and 4096 on TeslaC1060. However some peaks of R are not good though they occupy small region.



Platform: TeslaC1060

N	Volkov (Gflop/s)	Method 1, variant (Gflop/s)	R
256	207.668	118.354	1.7546
512	226.895	346.754	0.6543
1024	274.597	361.492	0.7596
2048	324.941	420.841	0.7721
4096	342.651	430.517	0.7959

Figure 38: performance of *method1_variant* on TeslaC1060. Averagely speaking, we have 10% improvement. One should compare this result with method1 in Figure 27.

The same speedup is proved on GTX285 but is more good, almost 10 % improvement at least when N > 1500. One may ask why GTX285 has better performance. I think that the reason is bandwidth. In Table 1, bandwidth of GTX285 is 159GB/s, 50% more than that of TeslaC1060 due to overclocking of DRAM. If we focus on graph of Gflop/s on GTX825 (right panel of Figure 39), TeslaC1060 (left panel of Figure 40) and GTX295 (right panel of Figure 40), then it is easy to find that graph of Gflop/s has large variation over N on TeslaC1060.

Question 3: why does graph of Gflop/s have large variation over N on TeslaC1060?

Answer: from programming guide 5.1.2.1, we know that segment size is 128 bytes (32 float) for 32-, 64-, and 128-bit data for compute capability 1.3 and if a half-warp addresses words in **n** different segments, **n** memory transactions are issued, one for each segment (for example, right two transactions are issued in panel in Figure 41). In our experiment, size of A is $m \times k$, size of B is $k \times n$, and size of C is $m \times n$, all matrices are column-major and leading dimension is set by $lda = m,\ ldb = k,\ ldc = m$. When dimension is not multiple of 32, then we cannot merge requests of half-warp into one transaction only. In other words, when dimension *N* is not multiple of 32, then effective bandwidth decreases. the consequence is performance degradation and that's why large variation of Gflop/s on TeslaC1060. However such phenomenon is not found on GTX285 and GTX295 because high bandwidth of them compensates extra transactions.

In section 1.1.3.2 we model throughput of global memory under 512 active threads per SM as 1.16 cycle/thread. Actually this estimation is wrong because the cycle here is core frequency, 1.3GHz, but memory frequency is 800MHz which leads to 100GB/s bandwidth. Suppose throughput is 1.16 cycle/thread and one thread transfers one float element (4 byte), then bandwidth per thread is $\dfrac{4(byte)}{(1/1.3GHz)} = 5.2\ GB/s$. TeslaC1060 has 240 cores, each core runs one thread, then total bandwidth is $240 \times (5.2\ GB/s) = 1248 GB/s$, this is impossible. So what we are agree with is: under same algorithm, throughput of global memory from high to low is $\mathrm{GTX285} > \mathrm{GTX925} < \mathrm{TeslaC1060}$. High throughput in GTX285 can tolerate extra transactions when lda is not multiple of 32.

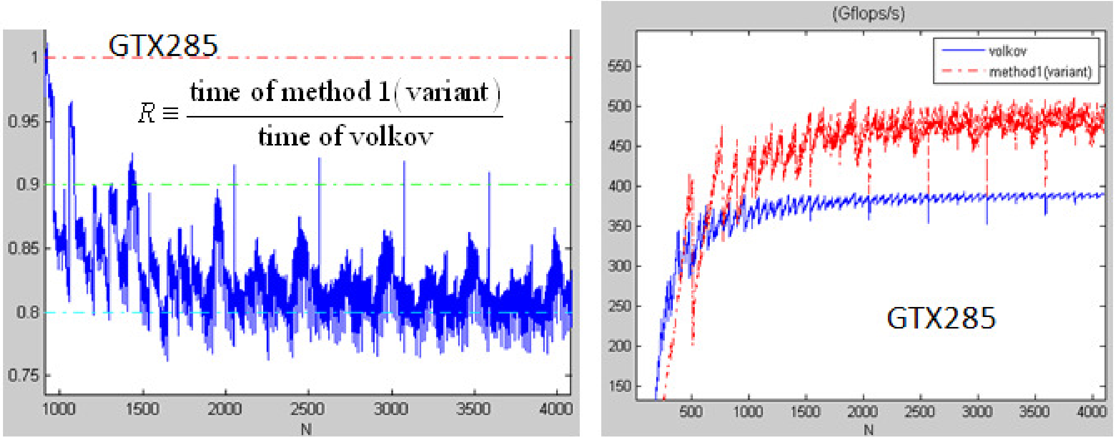


Figure 39: performance of *method1_variant* on GTX285, 10% improvement at least.

Remark 10: number of data transfer in matrix *A* in all algorithms in this work is main overhead, such extra transactions mainly come from contribution of matrix A. If number of data transfer of A can be reduced, then we expect that large variation in Gflop/s is relaxed. For example, we configure different active threads per SM from 320 to 192 artificially in method1_variant and list results in Table 11. Performance of 256 threads in method1_variant is worse than performance of 320 threads but it is still better than algorithm *volkov*. Hence we can use less active threads but more resource (registers per thread) to decrease number of data transfer of matrix A, that is why we introduce method 2 in section 5.

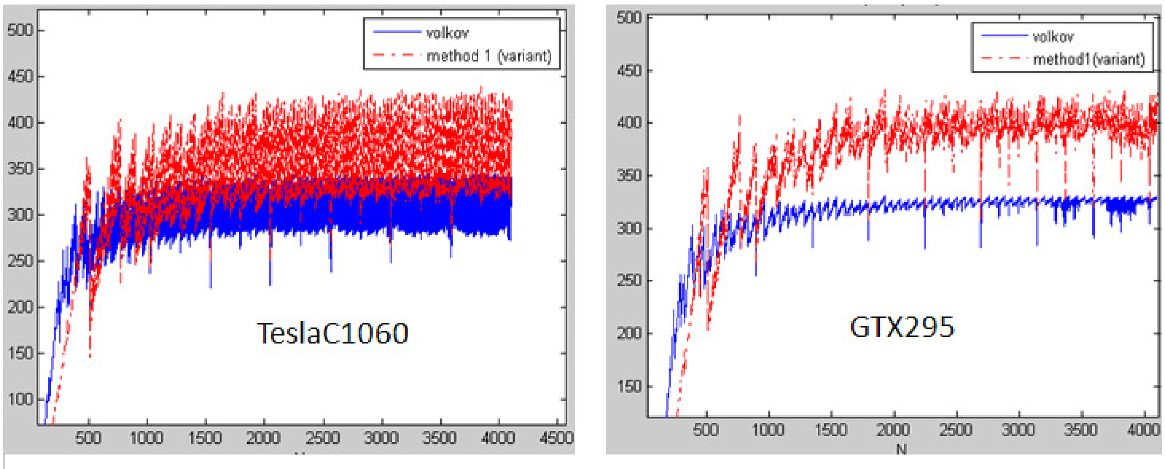


Figure 40: Gflop/s of method1 on TeslaC1060 and GTX295.

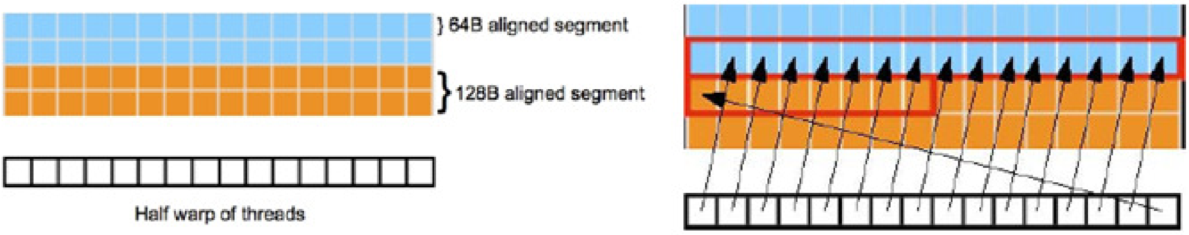


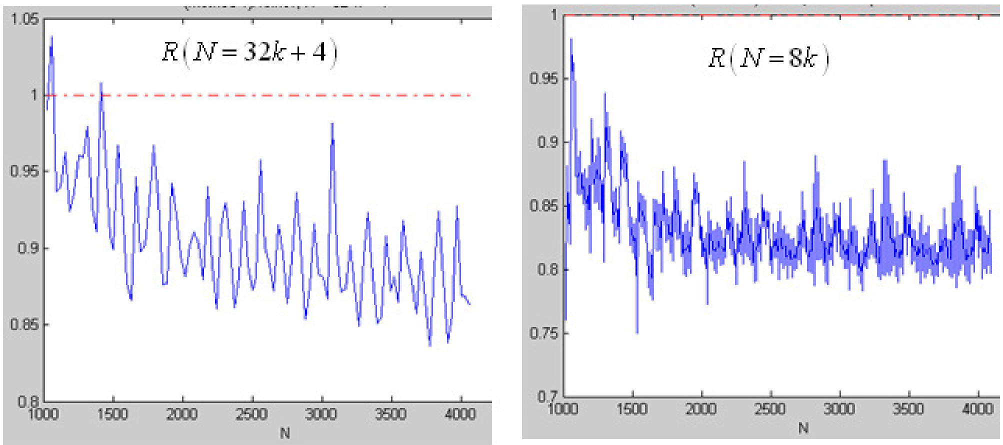
Figure 41: left panel: linear memory segments in figure 3.3 of best programming guide [10]. Right panel: misaligned sequential addresses result in two transaction in figure 3.6 of best programming guide.

N	volkov 512 threads	method1 320 threads	method1_variant 320 threads	method1_variant 256 threads	method1_variant 192 threads
256	207.668	145.308	118.354	118.392	118.477
512	226.895	290.171	346.754	225.800	224.832
1024	274.597	325.223	361.492	333.976	287.793
2048	324.941	348.155	420.841	379.480	305.243
4096	342.651	352.914	430.517	382.770	307.271

Table 11: performance of algorithm method1_variant under different active threads per SM on TeslaC1060. unit: Gflop/s.

4.4 padding to improve performance of method1_variant

Although performance of method1_variant on TeslaC1060 has lager variation than performance on GTX285 and GTX295, it can still have 10% improvement for specific dimension. In Figure 42 we show two specific choices of *N* on ratio *R*, one is $N = 32k + 4$ (left panel), the other is $N = 8k$ (right panel). This means that if we choose leading dimension lda as multiple of 8, then we should have 10% improvement for $N > 1500$. However if lda as multiple of 4, then performance is not good. Next we want to explain why $lda = 8k$ is better than $lda = 4k$.



$$R \equiv \frac{\text{time of method 1(variant)}}{\text{time of volkov}} \quad \text{on TeslaC1060}$$

Figure 42: sample special dimension *N* for performance of method1_variant.

The trick, $lda \neq N$, is called **padding**. If $lda = 8k$, then we say padding unit is 8. If $lda = 4k$, then we say padding unit is 4.

4.4.1 penalty of padding unit = 8

Suppose that matrix *A* is of size $m \times k$, and $m = 32 \times 2 + 8$. Compute capability of TeslaC1060 is 1.3 and then one segment is 128 bytes for 64-bit data access (i.e. 16 float). The distribution of segments is shown in Figure 43, here it suffices to show 8 segments and value of dimension k is immaterial. Next we calculate effective bandwidth of each column access.

In Figure 44, we consider coalesced pattern on column-0 of matrix *A*, data of 16 threads (half-warp) fall into the same segment, one 64-byte transaction is issued and then effective bandwidth is 100 %.

In Figure 45, coalesced pattern on column-1 of matrix *A* is considered.

(1) Threads of half-warp 0 access data in the same segment, one 128-byte transaction is issued and then effective bandwidth is 50 %.

(2) Threads of half-warp 1 access data in two segments, two 32-byte transactions are issued, and effective bandwidth is 100 %.

Total effective bandwidth of column-1 = $\dfrac{\text{data transfer in kernel}}{\text{transfer through bus}} = \dfrac{64+64}{128+32\times 2} = \dfrac{2}{3}$

In Figure 46, coalesced pattern on column-2 of matrix *A* is considered.

(1) Threads of half-warp 0 access data in the same segment, one 64-byte transaction is issued and then effective bandwidth is 100 %.

(2) So does half-warp 1

Total effective bandwidth of column-2= $\dfrac{\text{data transfer in kernel}}{\text{transfer through bus}} = \dfrac{64+64}{64+64} = 1$

In Figure 47, coalesced pattern on column-3 of matrix *A* is considered.

(1) Threads of half-warp 0 access data in two segments, two 32-byte transactions are issued and then effective bandwidth is 100 %.

(2) Threads of half-warp 1 access data in one segment, one 128-byte transaction is issued, effective bandwidth = 50 %.

Total effective bandwidth of column-3= $\dfrac{\text{data transfer in kernel}}{\text{transfer through bus}} = \dfrac{64+64}{32\times 2+128} = \dfrac{2}{3}$

To sum up, total effective bandwidth on loading of matrix *A* is

average of all column-access = $\dfrac{1+2/3+1+2/3}{1+1+1+1} = \dfrac{5}{6} = 0.833$.

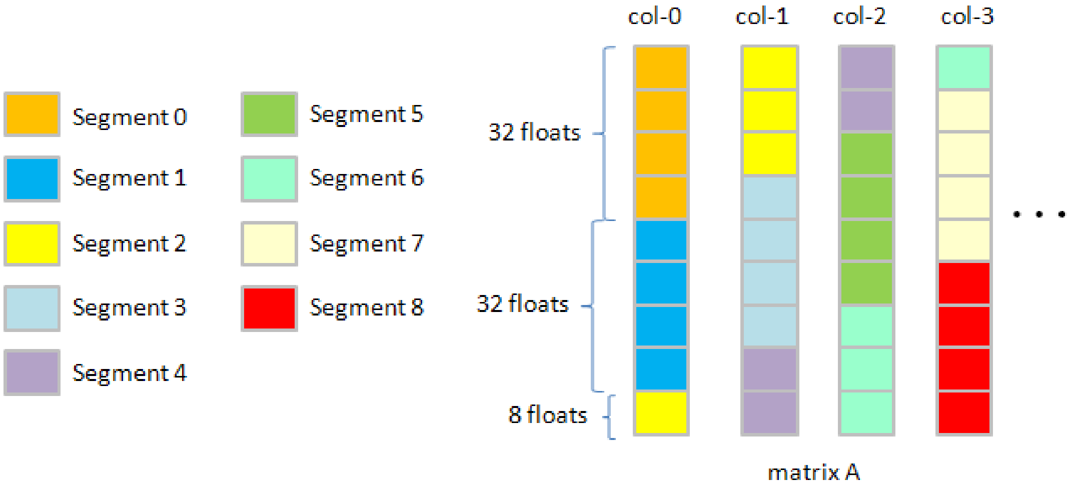


Figure 43: distribution of segments (128 byte per segment) in matrix *A* with *mxk*, and *m = 32 x 2 + 8*.

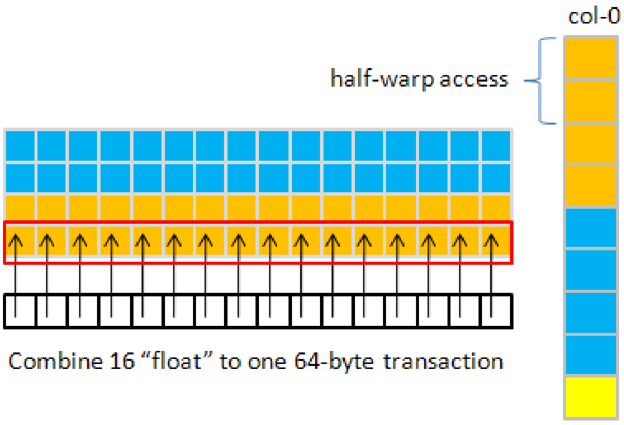


Figure 44: effective bandwidth of column-0 access is 100%.

Observation 4: $N = 32k + 8$ is worse than $N = 16k$ since effective bandwidth is 5/6. However 5/6 is not so bad, hence we can pad matrix such that *lda* is multiple of 8.

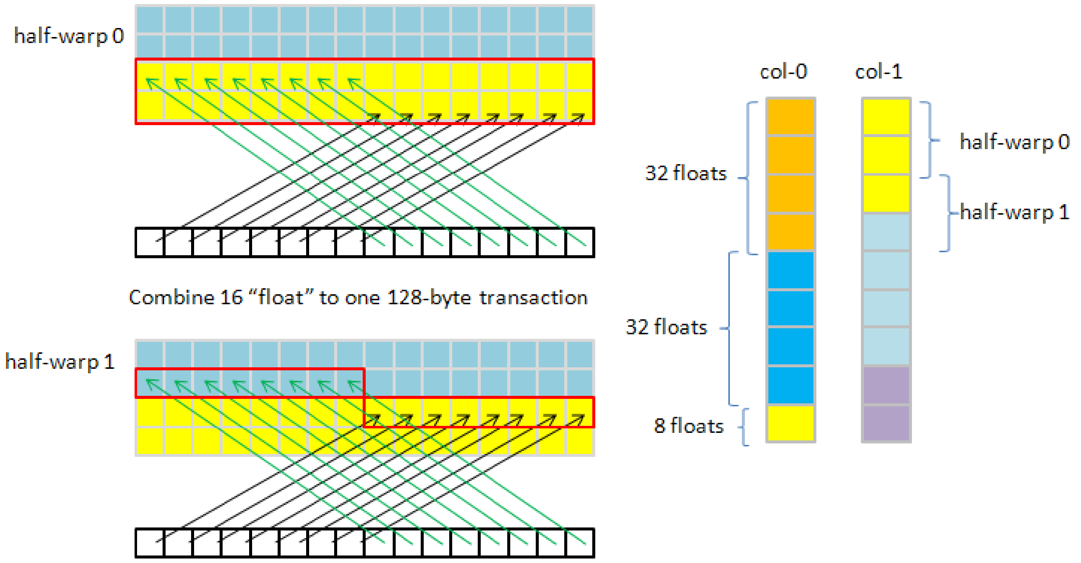


Figure 45: effective bandwidth of column-1 is 66%

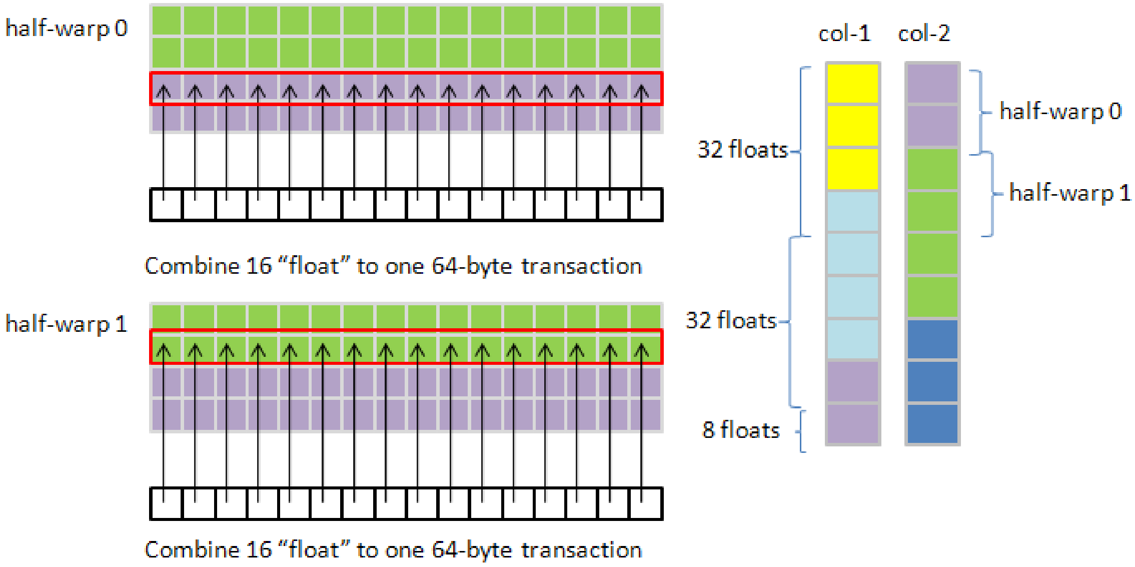

half-warp 0
Combine 16 "float" to one 64-byte transaction
half-warp 1
Combine 16 "float" to one 64-byte transaction
col-1
col-2
32 floats
32 floats
8 floats
half-warp 0
half-warp 1

Figure 46: effective bandwidth of column-2 is 100%

4.4.2 penalty of padding unit = 4

Suppose that matrix A is of size $m \times k$, and $m = 32 \times 2 + 4$. We summarize effective bandwidth of each column in Figure 48, and then total effective bandwidth on loading of matrix A is average of all column-access $=\frac{1}{8}\left(1+\frac{4}{7}+\frac{2}{3}+\frac{4}{7}+1+\frac{4}{7}+\frac{2}{3}+\frac{4}{7}\right)=0.7024$, which is worse than padding unit = 8.

Finally we show performance improvement of padding unit = 8, 16 in Figure 49. For $N > 150$, Padding unit = 8 has 10% ~ 25% improvement whereas padding unit = 16 has 14% ~ 25% improvement.

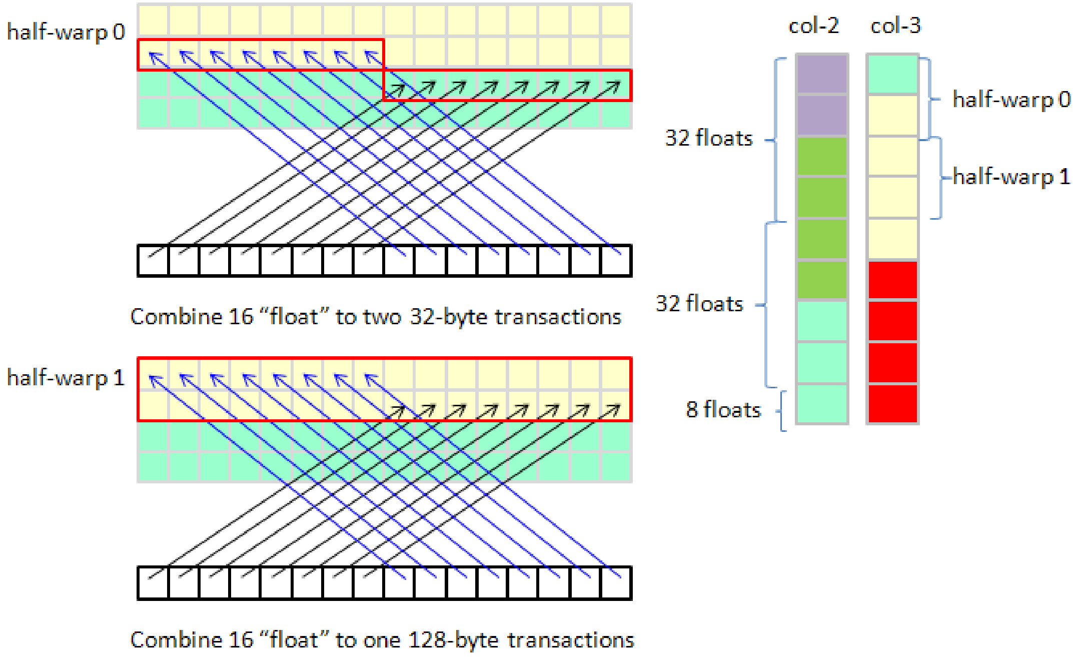

half-warp 0
col-2
col-3
half-warp 0
32 floats
half-warp 1
32 floats
Combine 16 “float” to two 32-byte transactions
half-warp 1
8 floats
Combine 16 “float” to one 128-byte transactions

Figure 47: effective bandwidth of column-3 is 66%

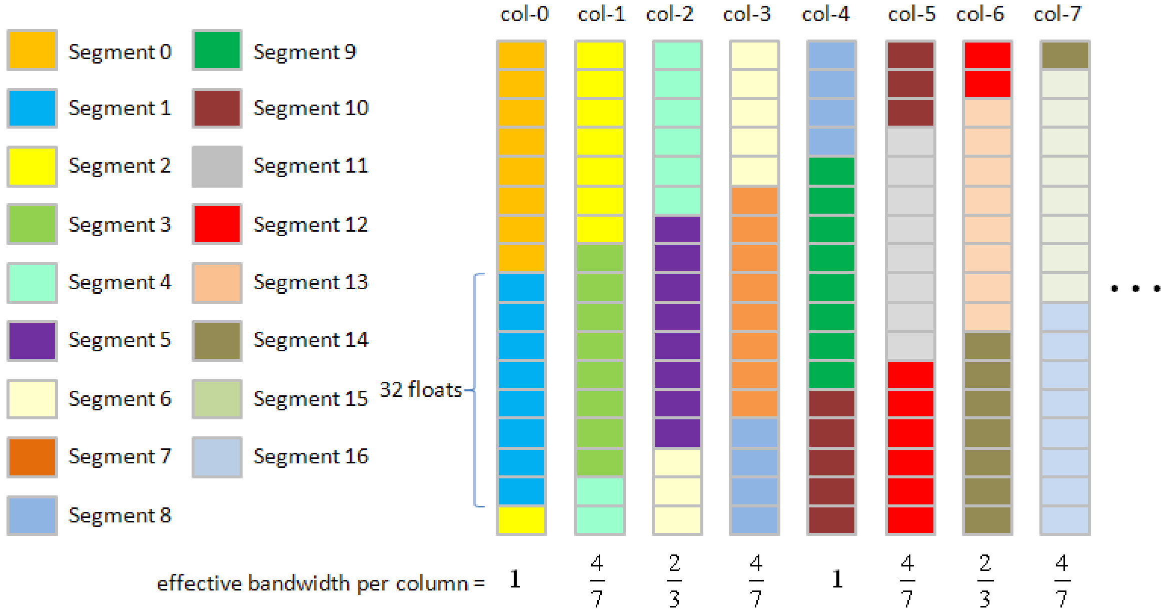


Figure 48: distribution of segments (128 byte per segment) in matrix A with mxk, and $m = 32 \times 2 + 4$. effective bandwidth is 0.7024

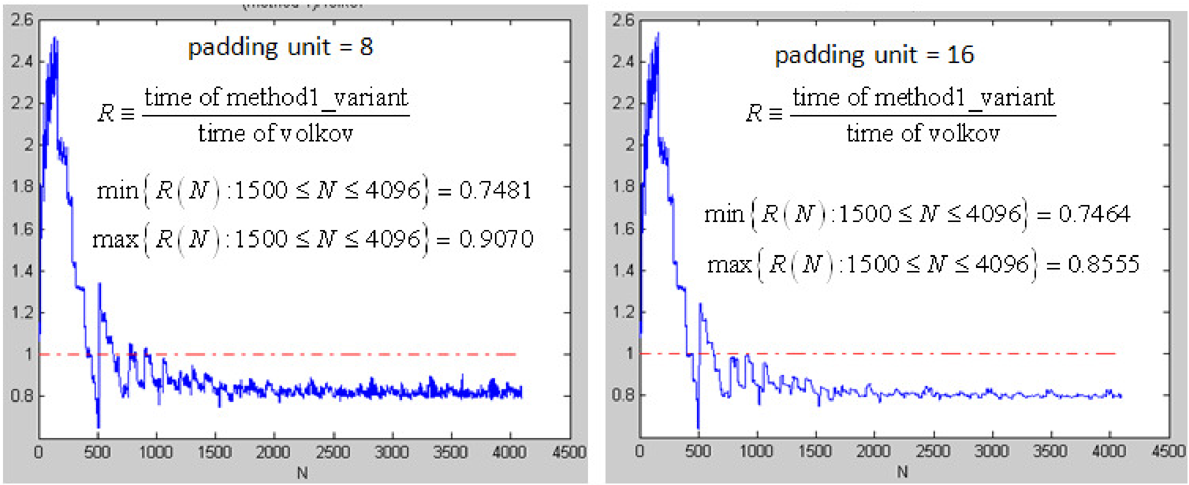


Figure 49: performance of padding unit = 8 is 10% ~25% improvement whereas performance of padding unit = 16 is 14%~25% improvement.

5 method 2 and method 3

method 2 is the same as method 1 except parameter BLOCK_SIZE_Y is 24, not 16. The larger BLOCK_SIZE_Y is, the smaller number of data transfer of matrix A is. Number of data transfer of matrix A in method 2 is $\frac{mnk}{24}$ which is 66% of that in method 1. Such reduction of memory load can compensate insufficiency of bandwidth of TeslaC1060 as we discussed in Question 3. However we must pay extra registers (number of reregister per thread = 64) in order to achieve memory reduction. The number of active threads per SM in method 2 becomes 256, smaller than 320 in method 1.

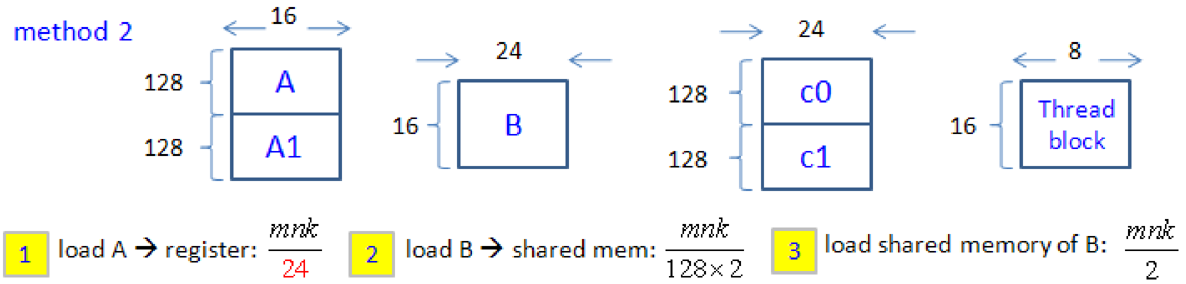

method 2
16
128
128
A
A1
24
16
B
24
128
128
c0
c1
8
16
Thread block
1
load A → register: $\frac{mnk}{24}$
2
load B → shared mem: $\frac{mnk}{128 \times 2}$
3
load shared memory of B: $\frac{mnk}{2}$

Figure 50: block information of method 2 and its variant. The difference between method 1 and method 2 is value of parameter BLOCK_SIZE_Y. In Figure 25, BLOCK_SIZE_Y is 16 in method 1, however we set BLOCK_SIZE_Y = 24 in method 2.

We show experimental result of method2_variant on TeslaC1060 in Figure 51. Red curve in right panel of Figure 51 is Gflop/s of method2_variant, more uniform than Gflop/s of method1_variant in left panel of Figure 40. This uniformity of Gflop/s reveals additional improvement. In left panel of Figure 51, R ranges from 0.7 to 0.9 and has no peaks for large N.

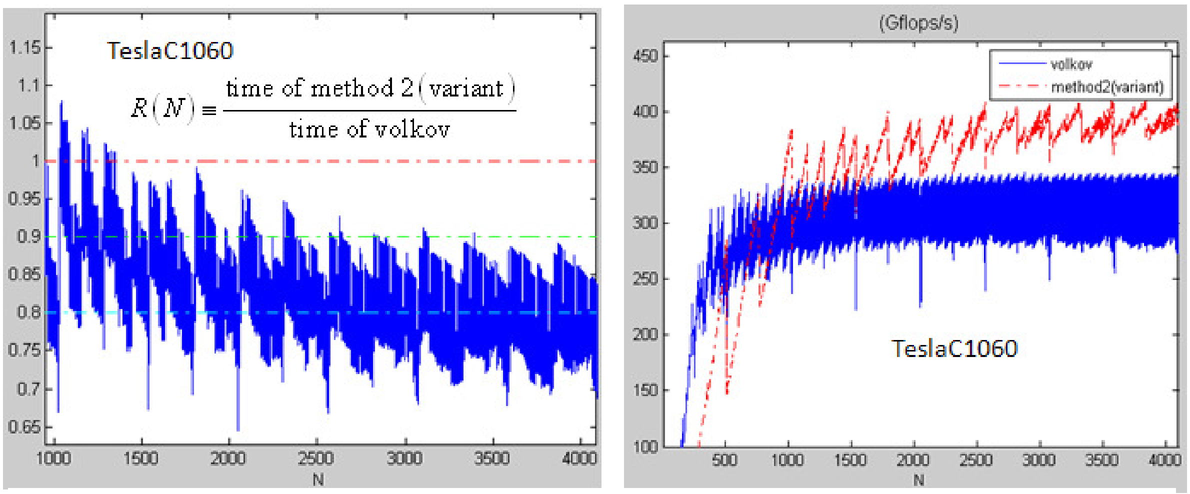


Figure 51: left panel: performance of methd2_variant on TeslaC1060. We have uniform 10% improvement more uniform than that of method1_variant in Figure 38. Such uniformity comes from uniformity of graph of Gflop/s (right panel).

Observation 5: Although method2_variant is better than method1_variant on TeslaC1060, same performance improvement does not hold on GTX285 and GTX295. In Figure 52, graph of Gflop/s of volkov is more uniform than that of method2_variant and then ratio R of method2_variant (left panel of Figure 52) is worse than R of method1_variant (left panel of Figure 39). Hence if we insist that performance of SGEMM is consistent in all GPUs of GT200, say high bandwidth leads to high improvement, then we should use method1_variant.

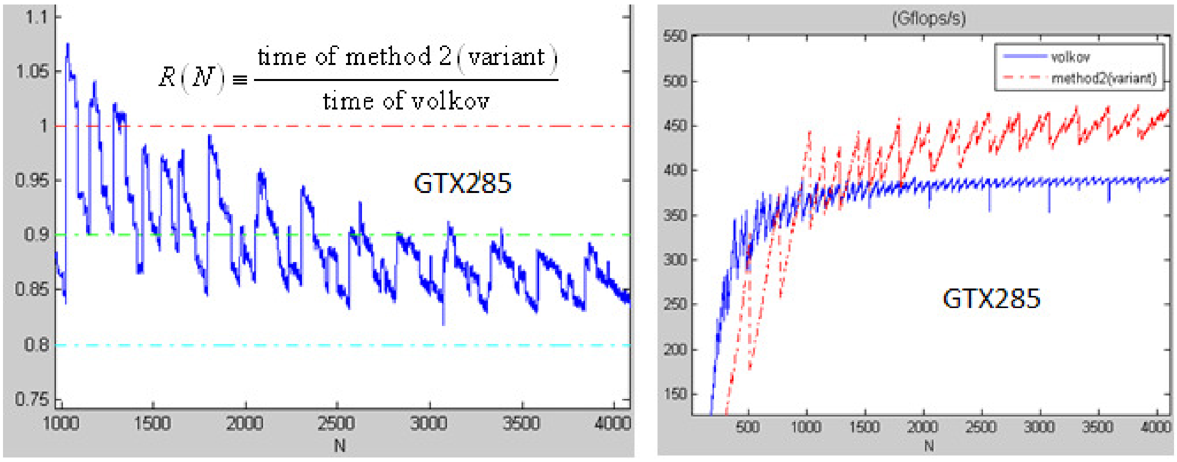
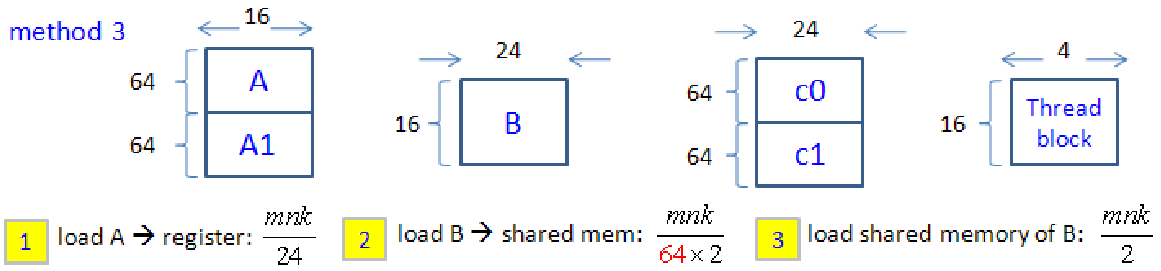


Figure 52: left panel: performance of methd2_variant on GTX285. Not good as method1_variant in Figure 39 since Gflop/s of volkov is more uniform than that of method2_variant in right panel.

Question 4: even method2_variant has good performance on TeslaC1060 but there is something so strange that we cannot find any explanation. If we change vector length of method 2 from 128 to 64 but keep all other parameters (number of active threads are 256 in both methods), then method 3 arises. The difference between method 2 (Figure 50) and method 3 (Figure 53) is number of data transfer of matrix *B*. However data transfer of matrix *B* is relative smaller than data transfer of matrix *A*, it should not be a bottleneck and there should be no difference of performance between method 2 and method 3. Unfortunately method 3 is pretty bad.



method 3
16
64
64
A
A1
24
16
B
24
64
64
c0
c1
4
16
Thread block
1 load A → register: $\frac{mnk}{24}$
2 load B → shared mem: $\frac{mnk}{64\times 2}$
3 load shared memory of B: $\frac{mnk}{2}$

Figure 53: block information of method 3 and its variant. Only difference from method 2 is vector length, vector length of method 3 is 64 whereas vector length of method 2 is 128.

In Figure 54 performance of method3_variant is bad then method2_variant both on TeslaC1060 and GTX285, so far we cannot find any reason for this phenomenon.

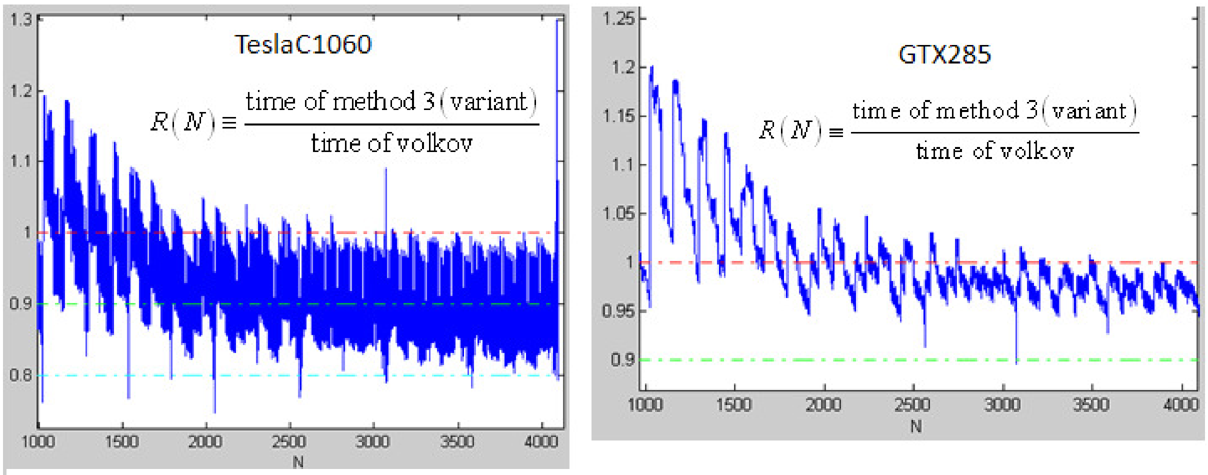


Figure 54: left panel: performance of methd3_variant on TeslaC1060.

right panel: performance of methd3_variant on GTX285

6 method 4

From section 3, Figure 22, we introduce basic idea to improve 16% if only is $\overline{T}_{C=A\times B}$ considered. However we don't obtain any speedup experimentally. Figure 55 shows block information of method 4 and its variant. There are four vectors with a gap between consecutive vectors in method 4 but no gap in Figure 22. Such gap can prevent partition camping on load of matrix *A*.

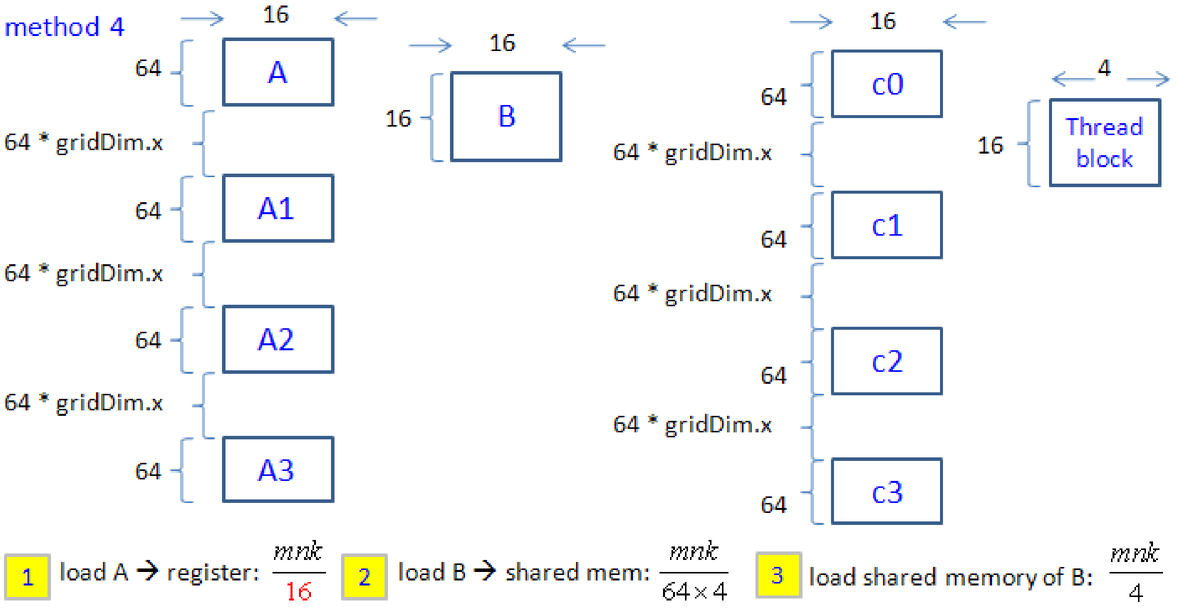


Figure 55: block information of method 4 and its variant

Experiment in Figure 56 reveals dramatically bad results on both TeslaC1060 and GTX285. I think the key is number of active threads per SM. Algorithm *volkov* has 512 active threads per SM because it uses only 30 registers per thread ,however method 4 has only 192 active threads per SM due to 80 registers per thread. 192 threads (6 warps) cannot hide latency of shared memory and cannot hide pipeline latency of "MAD dest, src1, src2, src3". In Table 3 pipeline latency of "MAD dest, src1, src2, src3" is 31.5 cycle and throughput is 4 cycle/warp. This means that we need 8 warps at least to hide pipeline latency of "MAD dest, src1, src2, src3".

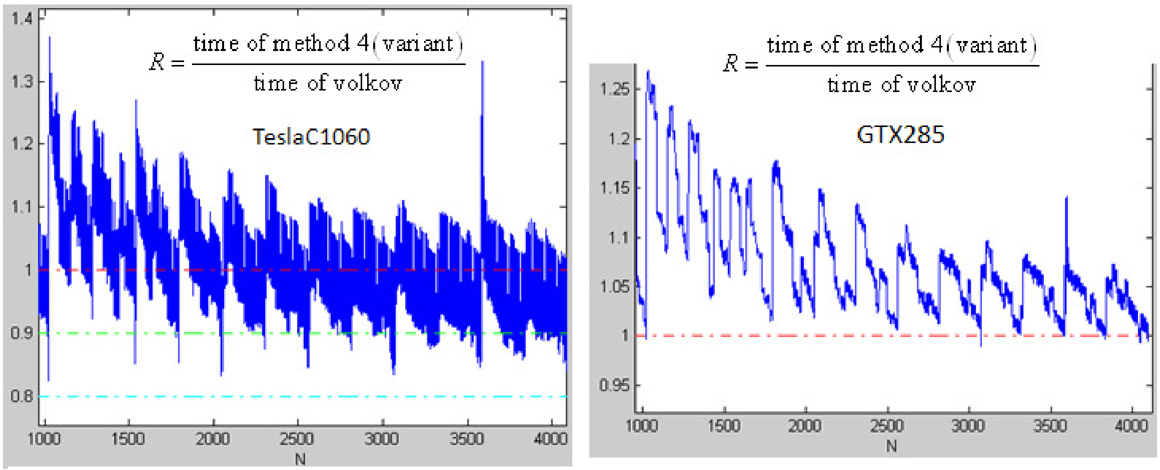


Figure 56: left panel: performance of methd4_variant on TeslaC1060.

right panel: performance of methd4_variant on GTX285

7 Volkov's code does not work for arbitrary dimension

Volkov’s code fetches sub-matrix of *B* of size 16x16 into shared memory and sub-matrix of *A* of size 64x1 into registers. If dimension of the matrix is not multiple of 16 or 64, then we have out of array bound problem.

Example 2: matrix *A* is $m \times k$, matrix *B* is $k \times n$ and matrix *C* is $m \times n$, assume

(1) Column-major and leading dimension $lda = m,\ ldb = k,\ ldc = n$ and

(2) ***m*** is not multiple of 64, ***k*** is not multiple of 16 and ***n*** is not multiple of 16. see Figure 57

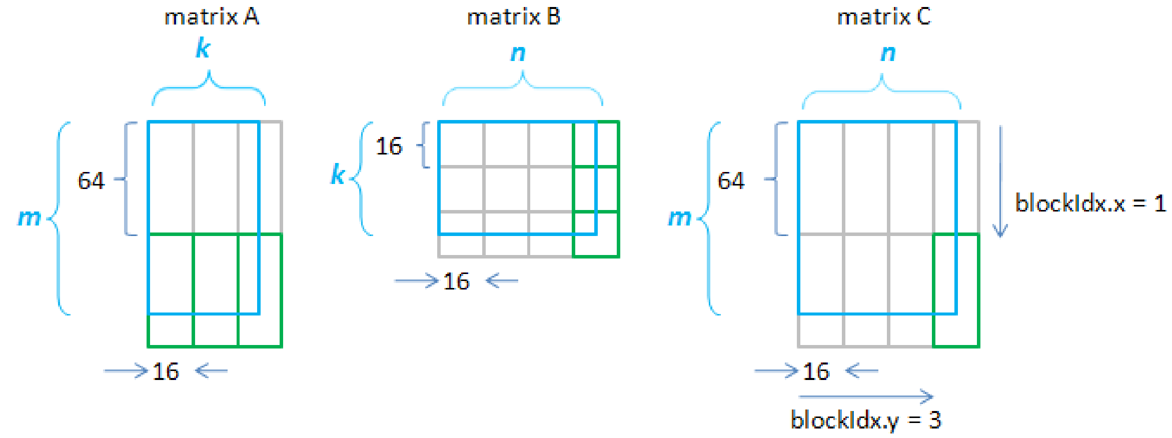


Figure 57: grid information of algorithm *volkov*, *m, n* are not multiple of 64 and *k* is not multiple of 16.

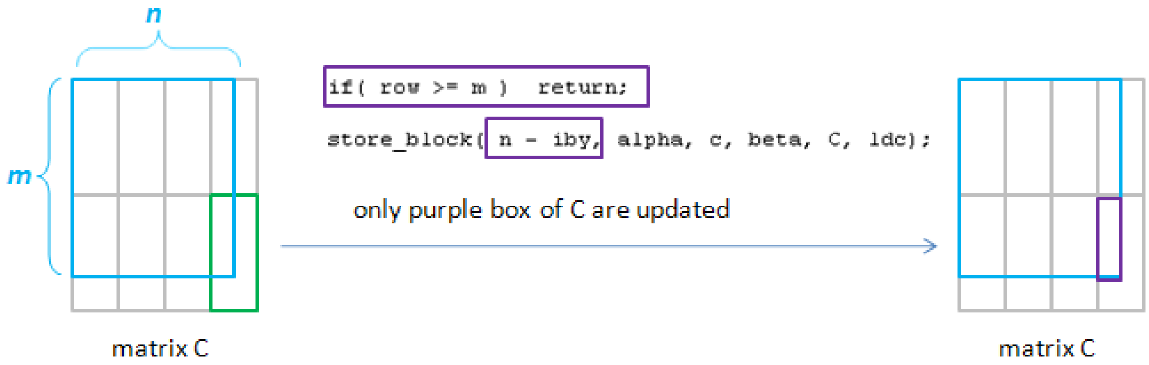
We take blockIdx = (1, 3) of matrix *C* as an example. blockIdx = (1, 3) is labeled as green box and corresponding rows of *A* and columns of *B* are also labeled as green boxes.

First, source code of algorithm volkov only allows valid entry of matrix *C* by two mechanisms.

(1) each thread updates one row of C, so if the row is not valid, then statement "if(row >= m) return;" can prevent writing.

(2) when a thread is allowed to update a row, then we need to check if all columns of the row are valid or not. This could be done for parameter "n - iby" passing into device function "store_block".

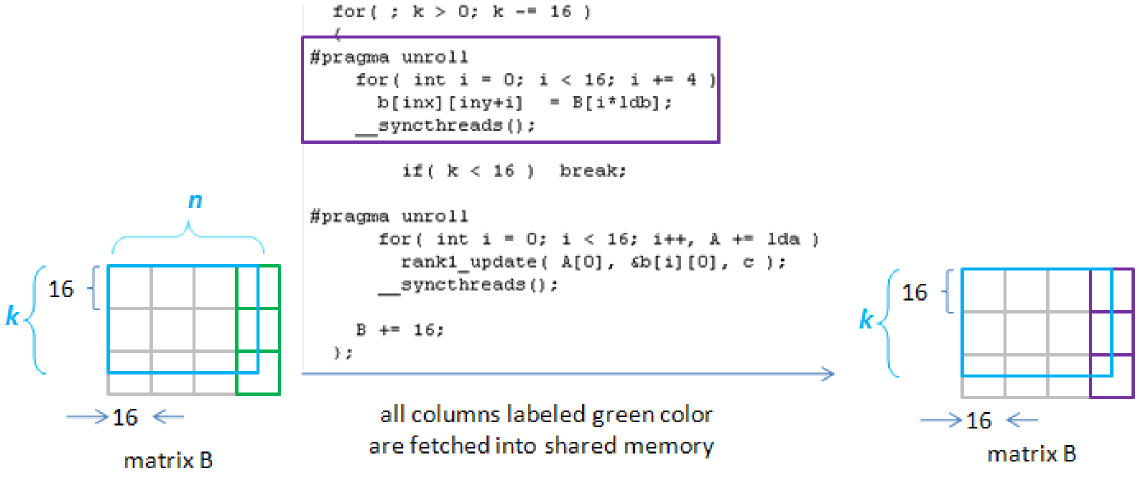
To sum up, there is no out-of-bound problem when writing to matrix C.



```
if( row >= m )  return;
store_block( n - iby, alpha, c, beta, C, ldc);
```

Figure 58: only valid entry of matrix C can be written into global memory, there is no out-of-array bound when writing data to matrix C.

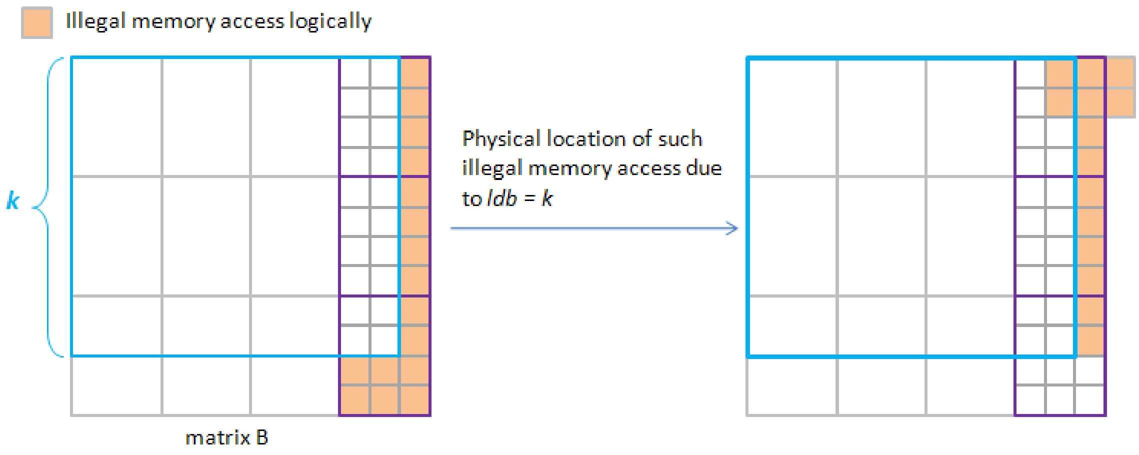
Second algorithm volkov load *16x16* sub-matrix of *B* before doing C = A * B, when ***k*** is not multiple of 16, then out-of-array bound occurs when compute rightmost grid of *C*, purple grids in Figure 59. Left panel of Figure 60 shows illegal memory-access logically by orange boxes. But their physical locations are different from logical locations, right panel of Figure 60 shows physical locations of such illegal memory-access.



```
  for( ; k > 0; k -= 16 )
  {
#pragma unroll
    for( int i = 0; i < 16; i += 4 )
      b[inx][iny+i]  = B[i*ldb];
    __syncthreads();

          if( k < 16 )  break;

#pragma unroll
      for( int i = 0; i < 16; i++, A += lda )
        rank1_update( A[0], &b[i][0], c );
      __syncthreads();

    B += 16;
  };
```

n
k
16
16
matrix B
all columns labeled green color
are fetched into shared memory
k
16
16
matrix B

Figure 59: load *16x16* sub-matrix of B when *k* is not multiple of 16, then out-of-array bound occurs.



Illegal memory access logically
k
matrix B
Physical location of such
illegal memory access due
to *ldb* = *k*

Figure 60: orange boxes in left panel are illegal logically, but their physical locations locate in right panel.

If we define three quantities $\bar{m} = 64\left\lfloor \frac{m+63}{64} \right\rfloor$ (smallest integer greater than *m* and be multiple of 64), $\bar{k} = 16\left\lfloor \frac{k+15}{16} \right\rfloor$ and $\bar{n} = 16\left\lfloor \frac{n+15}{16} \right\rfloor$. Then red boxes in Figure 61 represent logical elements of matrix *B* which result in out-of-array bound (segmentation fault). If the last element $\left(\bar{k}, \bar{n}\right)$ is still valid, then we can avoid segmentation fault. In other words, the condition to avoid segmentation fault is

$$\left(\bar{k} + ldb \times \bar{n}\right) \times sizeof\left(float\right) < alloc\left(B\right)$$

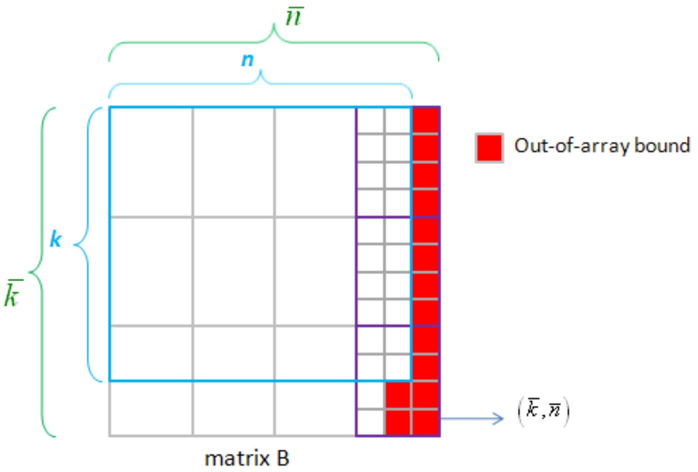
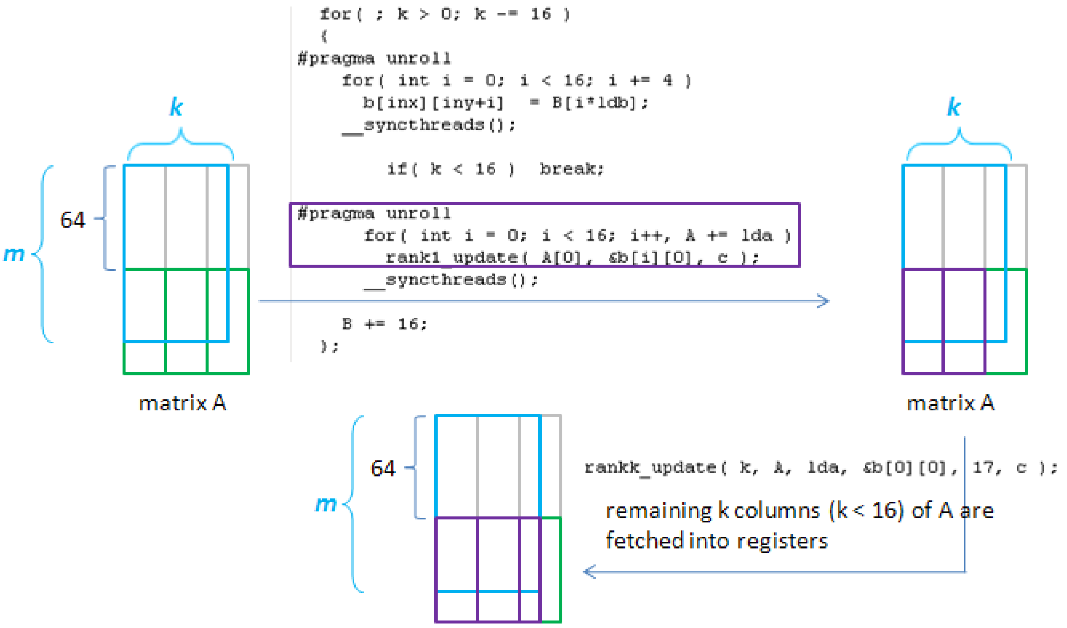


Figure 61: red boxes denote logical locations which result in out-of-array bound.

Third, algorithm *volkov* load *64x1* sub-matrix of *A* when doing rank-1 update. In Figure 62, the device function "rankk_update" avoids loading invalid columns of *A*, so we only focus on invalid rows of *A*. Although there are two rows of *A* are out-of-array bound logically, only two elements among these two rows are out-of-array bound physically. These two elements are labeled as red boxes in Figure 63. The condition to avoid segmentation fault is

$$\left(\bar{m}+lda\times\bar{k}\right)\times sizeof\left(float\right)<alloc\left(A\right).$$



```
  for( ; k > 0; k -= 16 )
  {
#pragma unroll
    for( int i = 0; i < 16; i += 4 )
      b[inx][iny+i]  = B[i*ldb];
    __syncthreads();

        if( k < 16 )  break;

#pragma unroll
      for( int i = 0; i < 16; i++, A += lda )
        rank1_update( A[0], &b[i][0], c );
      __syncthreads();

    B += 16;
  };
```

```
rankk_update( k, A, lda, &b[0][0], 17, c );
```

Figure 62: load *64x1* sub-matrix of A into register. It suffices to take care of invalid rows.

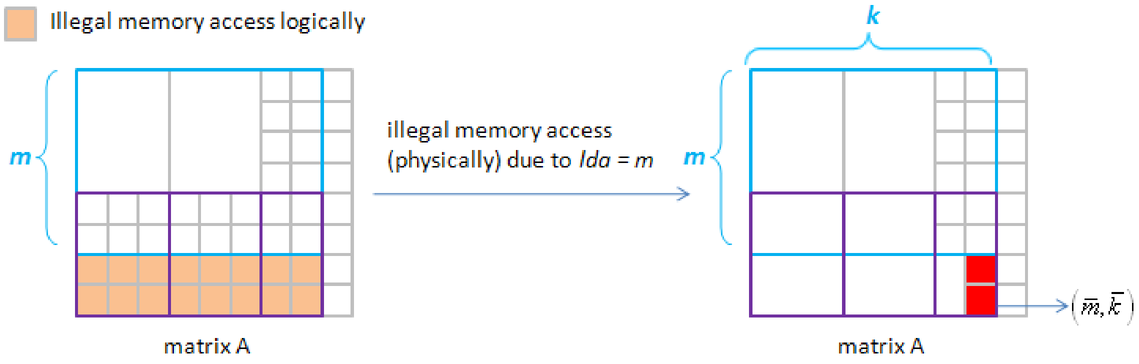

Illegal memory access logically
m
matrix A
illegal memory access
(physically) due to *lda* = *m*
k
m
($\bar{m}, \bar{k}$)
matrix A

Figure 63: left panel shows illegal logical locations of A and right panel shows illegal logical elements whose physical location are out-of-array bound indeed.

8 How to extend our methods to arbitrary dimension

We summarize Figure 61 and Figure 63 into Figure 64. Clearly out-of-array bound occurs in last column of sub-blocks of *B* and last row of sub-blocks of *A*. Besides if dimension *k* is not multiple of 16, then we have rank-1 update and additional rank-k update when computing $C_{bm,bn} = A_{bm,bk} B_{bk,bn}$. Out-of-array bound occurs in rank-k update when loading matrix *B* into $B_{bk,bn}$.

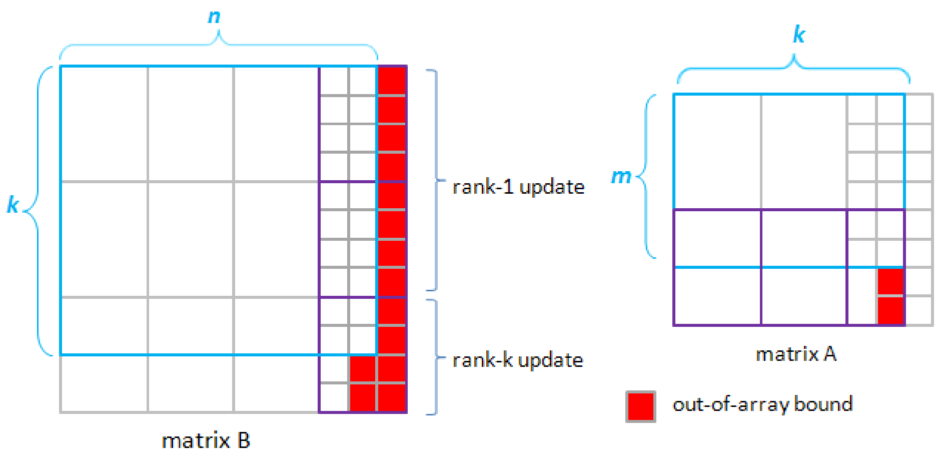


Figure 64: out-of-array bound occurs in last column of sub-blocks of *B* and last row of sub-blocks of *A*.

8.1 extend Volkov's code to arbitrary dimension

Formally speaking, we can classify grids of matrix *C* into four categories, and use six cases among these four categories to check out-of-array bound of matrix *A* or *B*. In Figure 65, we take Volkov" code as an example and use CPU code to determine which category current tripe (*m, n, k*) is. Inside the kernel, intrinsic variables, gridDim and blockDim, are used to determine which case is proper for this thread block. The six cases include

I: original Volkov's code, no out-of-array bound occurs.

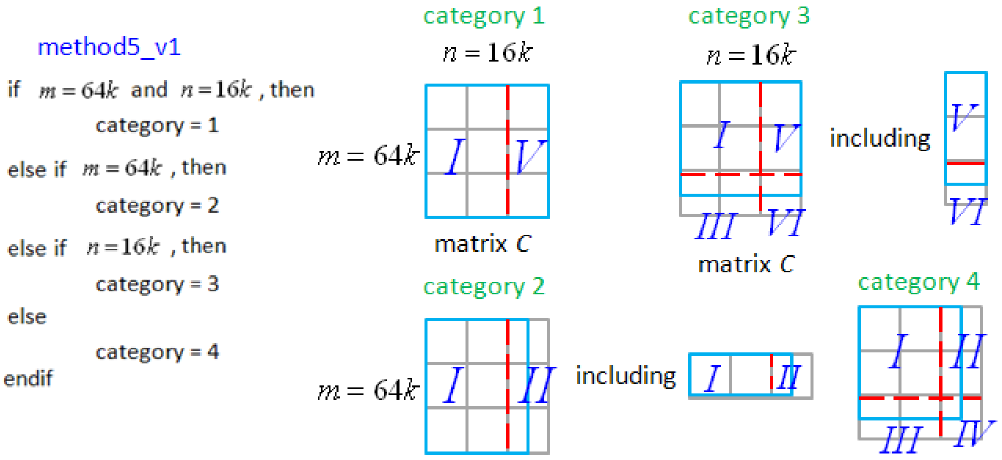
II: check *B* (ignore illegal memory access of matrix *B*)

III: check *A* (ignore illegal memory access of matrix *A*)

IV: check *A* and check *B*

V: check *B* only for rank-k update, keep load of B in rank-1 update unchanged. (in case II, "check B" means check load of *B* both in rank-1 update and rank-k update)

VI: check *A* and check *B* for rank-k update.



```
method5_v1
if  $m = 64k$  and  $n = 16k$ , then
        category = 1
else if  $m = 64k$ , then
        category = 2
else if  $n = 16k$ , then
        category = 3
else
        category = 4
endif
```

Figure 65: classify grids of matrix *C* into four categories, we use six cases (*I, II, III, IV, V, VI*) to check out-of-array bound of *A* or *B*.

Unfortunately, such intuitive modification is not good. We compare method5_v1 (version 1 of method 5) with Volkov's code in Figure 66. Method5_v1 only has better result when *n* is multiple of 64 since case *I* is applied. Otherwise method5_v1 is dramatically bad.

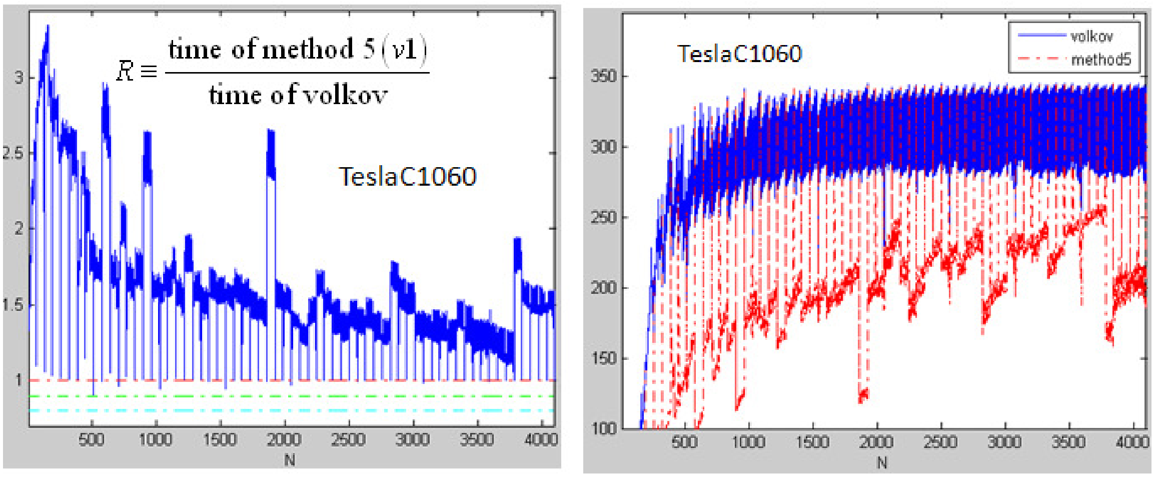


Figure 66: Left panel: performance improvement of method5_v1. Right panel: Gflop/s of method5_v1. Clearly method5_v1 is pretty bad.

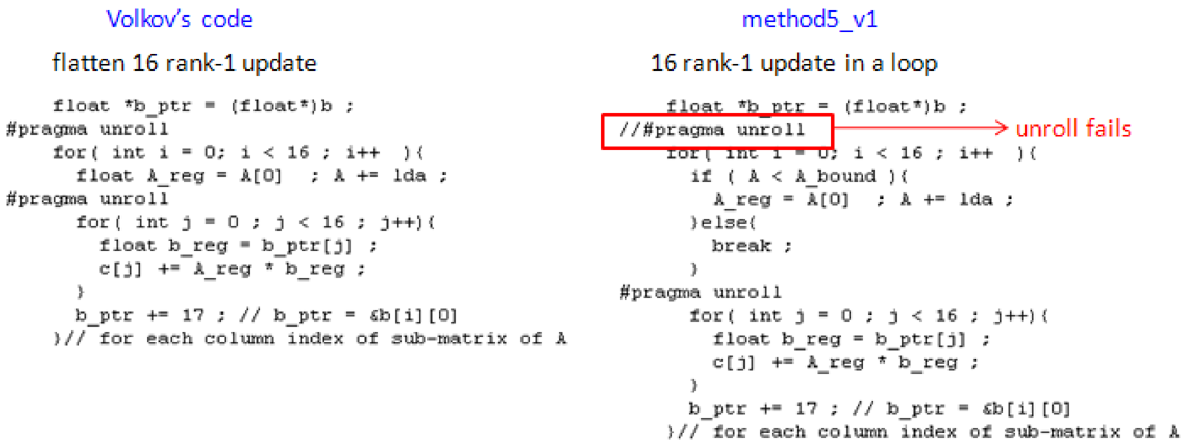
Question 5: why so bad is method5_v1?

Answer: though method5_v1 uses 39 registers per thread, which is so many that number of active threads is 384, not 512, mortal wound is "check A". There are two reasons, one is failure of loop-unrolling in Figure 67, and the other is

extreme high cost of "check A". Intuitive thinking of cost of "check A" in right panel of Figure 67 includes two factors (1) comparison between pointer A (pointer A in Volkov's code is movable, it points to A_{ik} of $C_{ij} = \sum_k A_{ik} B_{kj}$. Don't confuse it with address of matrix *A*) and pointer A_bound (A_bound is summation address of matrix *A* and $lda \times k$, so any address behind A_bound is illegal). cost is 4 cycle per warp
(2) jump if "A < A_bound" is not true. cost is 4 cycle per warp.

So only 8 cycle per warp is paid for each access of matrix A. Number of data transfer of *A* in Volkov's code is $\frac{mnk}{16}$ but "check A" only occurs in last row of sub-block of *A* (only M thread blocks). Hence cost of "check A" = $\frac{mnk}{16}(threads) \times \frac{1}{M} \times (8\ cycle / wrap)$.

On the other hand, cost of MAD is $mnk \times (4\ cycle / wrap)$. This means $\frac{\text{"check A"}}{MAD} = \frac{1}{8M}$, "check A" should not degrade performance so much.



```
    float *b_ptr = (float*)b ;
#pragma unroll
    for( int i = 0; i < 16 ; i++  ){
      float A_reg = A[0]   ; A += lda ;
#pragma unroll
      for( int j = 0 ; j < 16 ; j++){
        float b_reg = b_ptr[j] ;
        c[j] += A_reg * b_reg ;
      }
      b_ptr += 17 ; // b_ptr = &b[i][0]
    }// for each column index of sub-matrix of A
```

```
    float *b_ptr = (float*)b ;
//#pragma unroll
    for( int i = 0; i < 16 ; i++  ){
      if ( A < A_bound ){
        A_reg = A[0]   ; A += lda ;
      }else{
        break ;
      }
#pragma unroll
      for( int j = 0 ; j < 16 ; j++){
        float b_reg = b_ptr[j] ;
        c[j] += A_reg * b_reg ;
      }
      b_ptr += 17 ; // b_ptr = &b[i][0]
    }// for each column index of sub-matrix of A
```

Figure 67: nvcc cannot do loop-unrolling for each column of sub-block of A because we check validity of address A inside the loop.

However from result of ***decuda***, extra overhead of "check A" is 67 lines assembly code, i.e. $67 \times (4\ cycle / wrap)$ at least, not $8\ cycle / wrap$. Therefore $\frac{\text{"check A"}}{MAD} = \frac{4.1875}{M}$, that is the reason of poor performance.

Observation 6: if number of rows of *A* is greater than 64, then we can ignore case *III* since memory access in case *III* is out-of-bound logically but is still valid physically. We develop method5_v2 (version 2 of method 5), which is method5_v1 without case *III*. In Figure 68, method5_v2 is indeed better than method5_v1. However we can do more aggressively.

Observation 7: In Volkov's code, $C_{bm,bn} = A_{bm,bk} B_{bk,bn}$ is computed but only legal address of $C_{bm,bn}$ can be written into matrix C. In other words, if some row of $C_{bm,bn}$ is illegal, then corresponding value of *A* is immaterial. Hence in

Figure 69 we can re-direct illegal pointer A to last row of matrix *A*. This trick avoids operation "check A", then case *III*, *IV* and *VI* are useless, which can be replaced by case *II*. So we only use case *I, II* and *V* to implement method5_v3 (version 3 of method 5) in Figure 70. As we expect, Figure 71 shows that performance of method5_v3 is almost the same as Volkov's code without considering out-of-array bound.

Remark 11: although performance of method5_v3 is almost the same as Volkov's code, we still take Volkov's code as our reference model. So we compare our methods with Volkov's code in remaining sections.

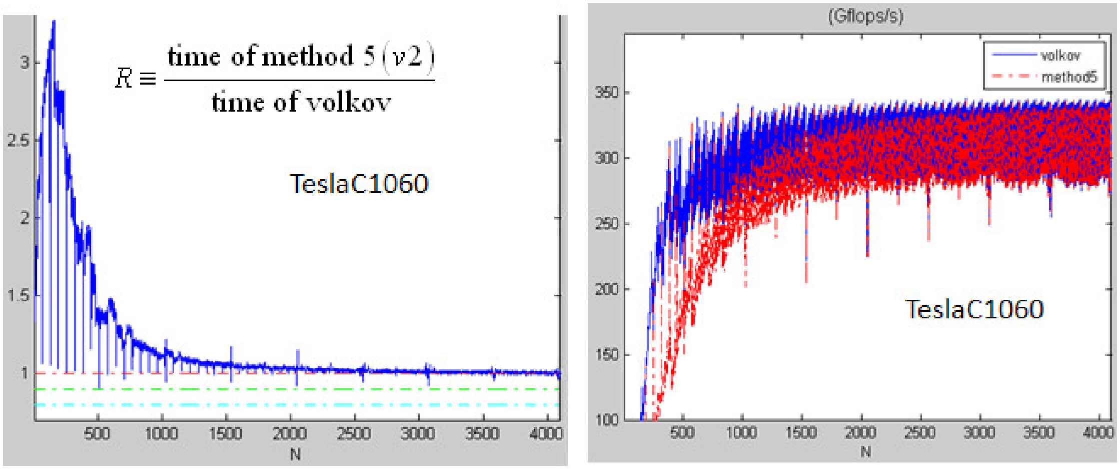
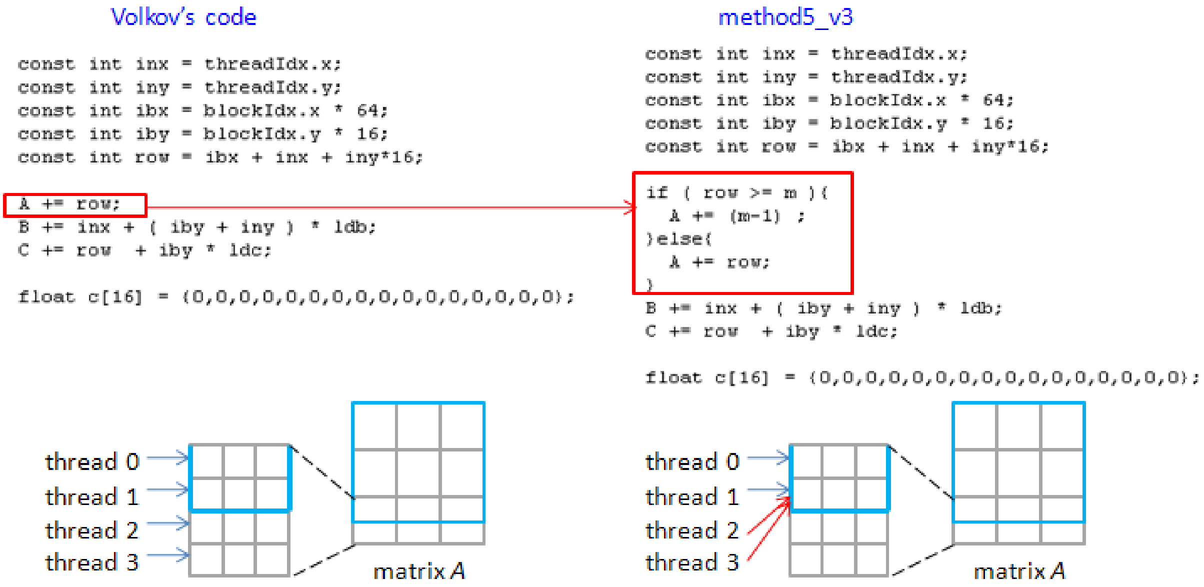


Figure 68: Left panel: performance improvement of method5_v2. Right panel: Gflop/s of method5_v2. method5_v2 is much better than method5_v1.



```
const int inx = threadIdx.x;
const int iny = threadIdx.y;
const int ibx = blockIdx.x * 64;
const int iby = blockIdx.y * 16;
const int row = ibx + inx + iny*16;

A += row;
B += inx + ( iby + iny ) * ldb;
C += row  + iby * ldc;

float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

```
const int inx = threadIdx.x;
const int iny = threadIdx.y;
const int ibx = blockIdx.x * 64;
const int iby = blockIdx.y * 16;
const int row = ibx + inx + iny*16;

if ( row >= m ){
  A += (m-1) ;
}else{
  A += row;
}
B += inx + ( iby + iny ) * ldb;
C += row  + iby * ldc;

float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

Figure 69: when pointer A is illegal logically, then we re-direct it to last row of matrix A. Though value of pointer A is wrong, it does not affect C = A*B.

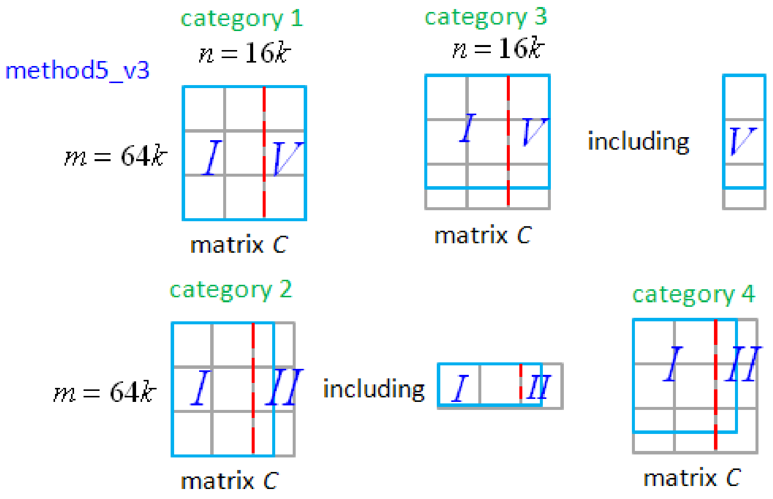


Figure 70: we don't need to "check A" in method_v3 and then case II, IV, VI can be replaced by case II.

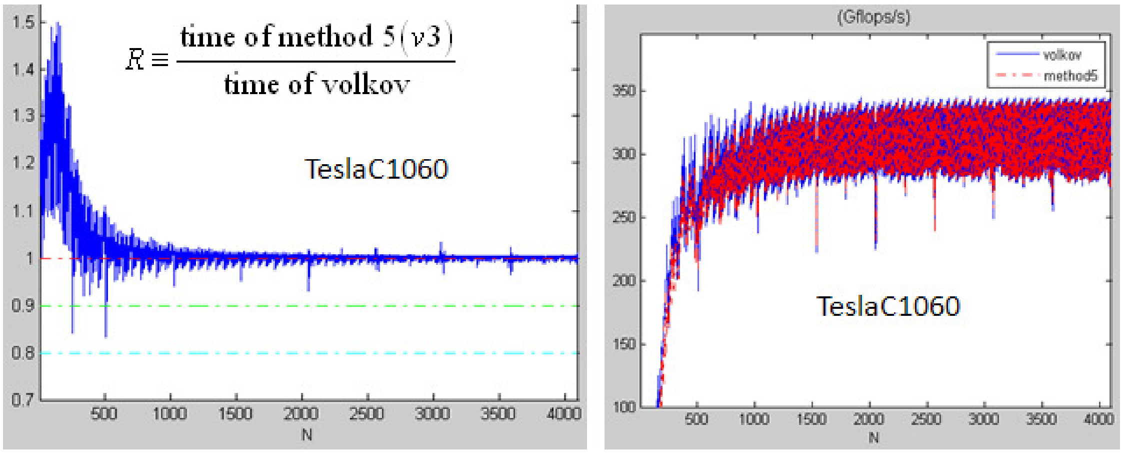


Figure 71: Left panel: performance improvement of method5_v3. Right panel: Gflop/s of method5_v3. method5_v3 is almost the same as Volkov's code for $n > 1000$.

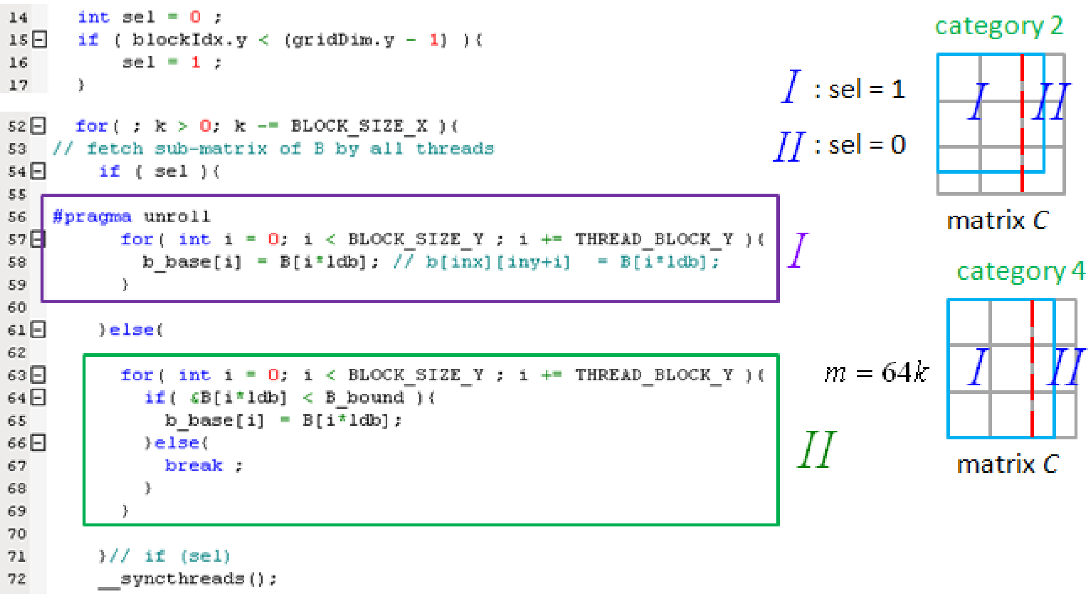
8.2 method 6 (extension of method 1)

method 6 is method 1 with out-of-array bound checker. However we cannot use four categories of method5_v3 because of register count. There are 320 active threads per SM in method 1, then threshold of register count is 48 (it register count > 48, then we have only 256 active threads). Since method 6 is an extension of method 1, we expect that there are also 320 active threads per SM in method 6. To achieve this goal, we must abandon case V in method5_v3 and merge common part of case I and case II. The code of case II is the same as code of case I except that case II needs to check validity of address B, hence in Figure 72 we use one flag "sel" to distinguish case I and case II. Then register count of method 6 is 54 but register count of method6_variant is 47 if we use compiler option, "-maxrregcount

49". Hence method6_variant has 320 active threads per SM, the same as active threads of method1_variant.

We compare performance between method1_variant and method6_variant in Figure 73. Generally speaking, method6_variant has the same performance profile as method1_variant. Of course if one looks at result of GTX285, then method1_variant has more test cases beneath $R = 0.8$, however R of method6_variant also ranges from 0.8 to 0.9 for $n > 1500$ on GTX285.

To sum up, we have equipped method1_variant with out-of-array bound checker and do not sacrifice performance on TeslaC1060. As far as we are concerned about **library**, method6_variant is no doubt a better choice than method1_variant because users don't need to care about size of allocation. However if one uses method1_variant, then he can obtain better performance, up to 5% in base cases. However he must remind himself to allocate large memory block for matrix A and B, the library cannot check size of array for him.



```
    int sel = 0 ;
    if ( blockIdx.y < (gridDim.y - 1) ){
        sel = 1 ;
    }

   for( ; k > 0; k -= BLOCK_SIZE_X ){
 // fetch sub-matrix of B by all threads
      if ( sel ){

 #pragma unroll
        for( int i = 0; i < BLOCK_SIZE_Y ; i += THREAD_BLOCK_Y ){
          b_base[i] = B[i*ldb]; // b[inx][iny+i]   = B[i*ldb];
        }

      }else{

        for( int i = 0; i < BLOCK_SIZE_Y ; i += THREAD_BLOCK_Y ){
          if( &B[i*ldb] < B_bound ){
            b_base[i] = B[i*ldb];
          }else{
            break ;
          }
        }

      }// if (sel)
      __syncthreads();
```

Figure 72: merge common part of case I and II and use flag "sel" to distinguish different part.

9 could we achieve uniform speedup on GT200?

Although we have completed method1_variant in section 8 and propose method6_variant, there are several peaks above $R = 0.9$ on TeslaC1060. We have discussed this phenomenon and we conclude that this is because of lower bandwidth of TeslaC1060. In other words, if one can overclock memory frequency of TeslaC1060 from 800 MHz to 1GHz, method6_variant should have uniformly speedup for large n. The truth is "we cannot overclock TeslaC1060" and then two methods can be used to compensate low bandwidth, one is to avoid partition camping, the other is to decrease number of data transfer of matrix A.

9.1 avoid partition camping

Global memory is divided into either 7 partitions on GTX295 or 8 partitions (on 200- and 10-series GPUs) of 256-byte width (64 floats) [6]. Order of thread blocks of Volkov's code is column-major. For example, first 8 thread blocks compute $C_{bm,bn} = A_{bm,bk} B_{bk,bn}$ on first column sub-blocks of *C*, this means that these 8 thread blocks load first column sub-blocks of *B* and each thread block loads different row sub-block of *A*. Since data transfer of matrix *B* is relative small, we can neglect effect of competence on data transfer of matrix *B*. Let us focus on data transfer of matrix *A* and assume *m* = multiple of 64. "vector length = 64" means that each sub-block of matrix *A* occupies only one partition. First 8 thread blocks use 8 partitions mutual exclusively.

However this property does not hold in method 6. Each thread block loads $A_{bm,bk}$ of size 128 x16, which occupies two partitions. We develop method 7 to keep the property, one thread block occupies one partition of matrix A. Grid information of method 7 is shown in Figure 74, and two vectors of matrix A are not adjacent. Unfortunately performance of method 7 on TeslaC1060 is worse than method 1, and has more peaks above *R* = 0.9. On the other hand, method 7 and method 1 have the same performance on GTX285 due to high bandwidth of GTX285, this is what we expected.

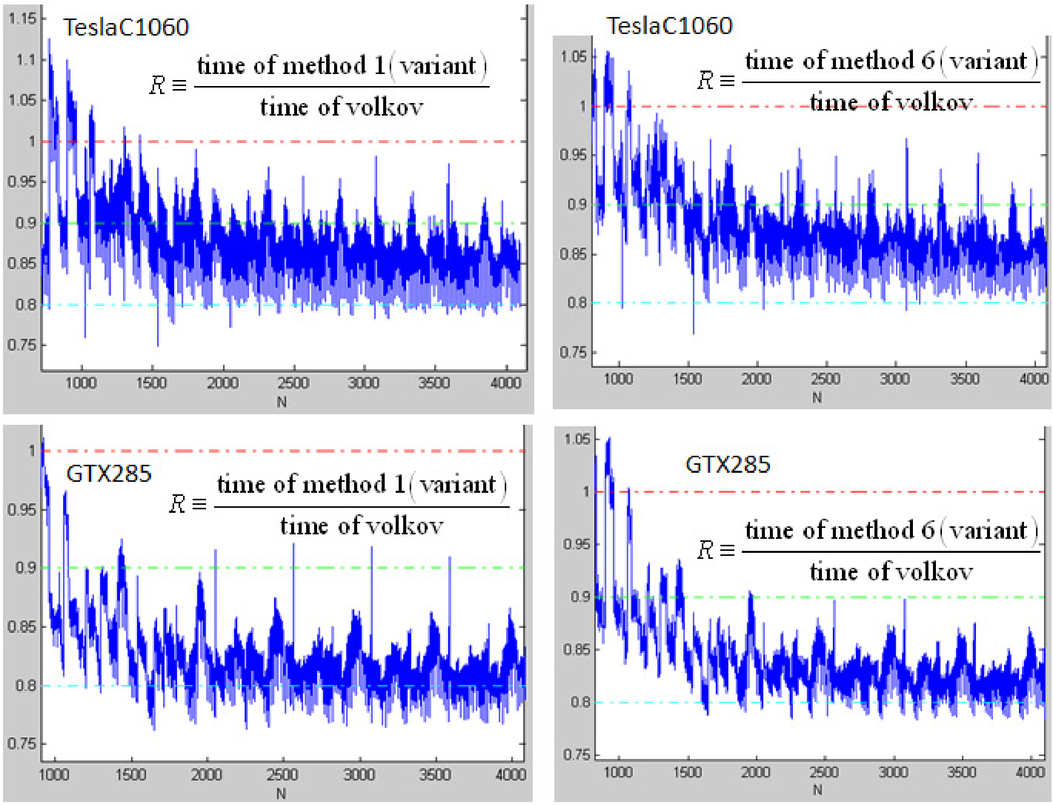


Figure 73: performance comparison between method1_variant and method6_variant. upper panel is TeslaC1060 and bottom panel is GTX285. Generally speaking, ethod6_variant is as good as method1_variant.

9.2 decrease number of data transfer of matrix A

We have use this idea in developing method 2 and method 3, but method 3 does not have expected speedup. In this section we modify parameter BLOCK_SIZE_Y in method 3, decrease it from 24 to 20, but keep other parameters unchanged. Also we add out-of-array bound check and then call this method as method 8. Figure 75 shows uniform performance of method8_variant on TeslaC1060, but sacrifice 5% performance averagely on GTX285. In other words, method 8 has larger speedup on TeslaC1060 than speedup on GTX285, this extra speedup on TeslaC1060 comes from large variation of Gflop curve in Volkov's code.

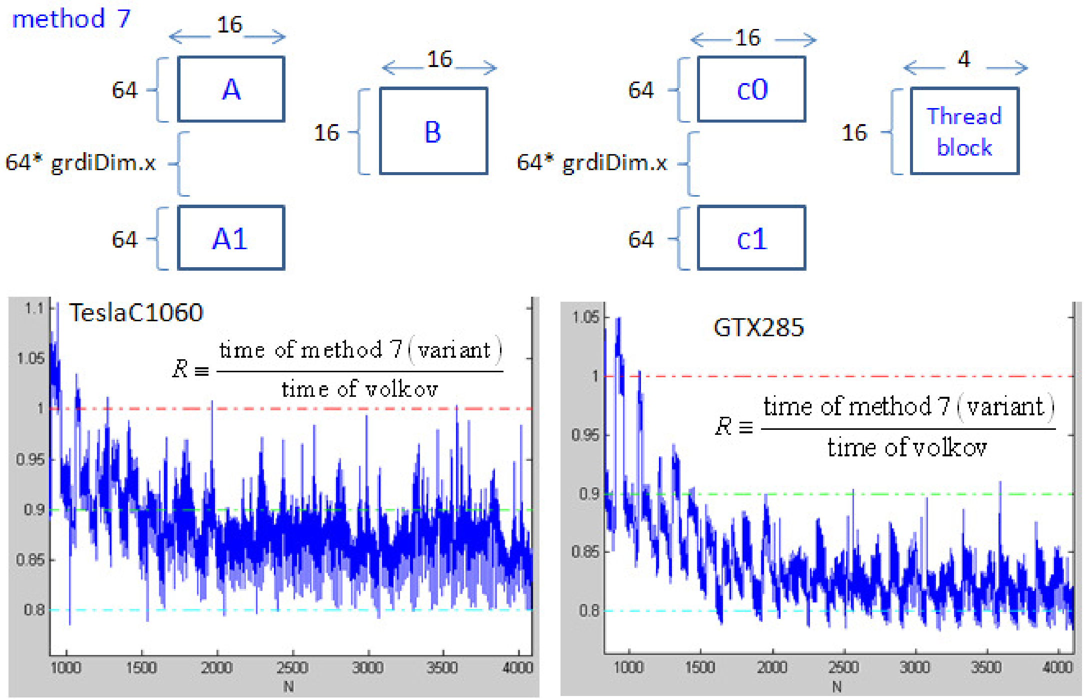


Figure 74: grid information of method 7 and performance on TeslaC1060 and GTX285. In fact method7_variaint has more peaks than method6_variant on TeslaC1060.

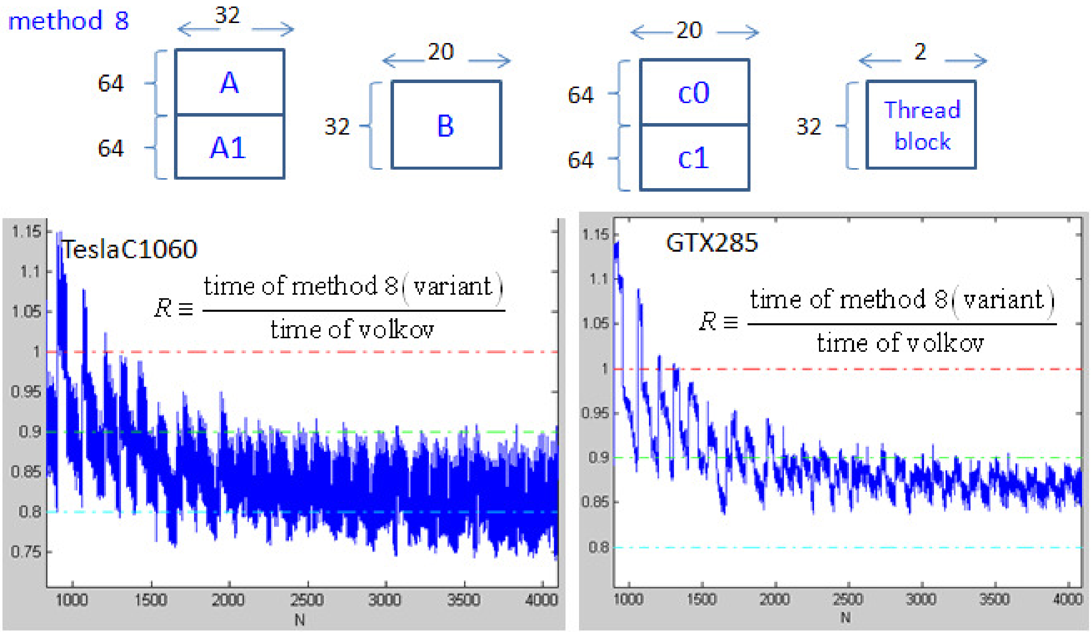


Figure 75: grid information of method 8 and performance on TeslaC1060 and GTX285. method8_variant has uniform performance for $n > 2000$.

10 conclusions

In this work, we implement eight methods (method 5,6,7 and 8 consider out-of-array bound) and compare them with Volkov's code. The package **decuda/cudasm** is used to modify binary code of variant version of each method. What we care about is performance of variant version, in order to simplify notation, we ignore "variant" in this section. For example, method 1 means method1_variant.

If programmers want a black-box SGEMM solver and don't want to care about size of allocation, then method 8 is the only choice though it sacrifices about 5% performance on game card. On the contrary if programmers can control allocation size of matrix A, B and C themselves in their application, then method 1 is a good choice, especially on game card. However size of allocation must be dealt carefully or out-of-array bound may occur.

Resource usage and computational cost of these five methods are listed in Table 12. According to resource usage, including threads per block, registers per thread and shared memory per block, one can determine number of active threads per SM via CUDA occupancy calculator. Then method 4 has only 192 active threads, smallest among five methods, and such six warps cannot hide pipeline latency of MAD operation and latency of shared memory. Hence it is not surprised that method 4 is even worse than Volkov's code. Next we can use computational cost to determine which one is better, method 1 or method 2 ? Note that we cannot explain why method 3 is not good as method 2. Computation cost has three factors, one is load of matrix *A* and *B* (dominated by data transfer of *A*), one is number of data transfer from shared memory to register, another is number of MADs. Method 1 and method 2 only differ on first factor. Moreover method 1 has more active threads than method 2, this means that memory throughput of method 1 is higher a little bit than that of method 2. If we take both number of data transfer and memory throughput into account, then ratio of cost of data transfer between method 1 and method 2 is

$$\text{method 1} : \text{method 2} = \frac{1}{320}\left(\frac{1}{16}+\frac{1}{64\times 2}\right)mnk : \frac{1}{256}\left(\frac{1}{24}+\frac{1}{128\times 2}\right)mnk = 1 : 0.81$$

Hence method 2 should be slightly faster than method 1.

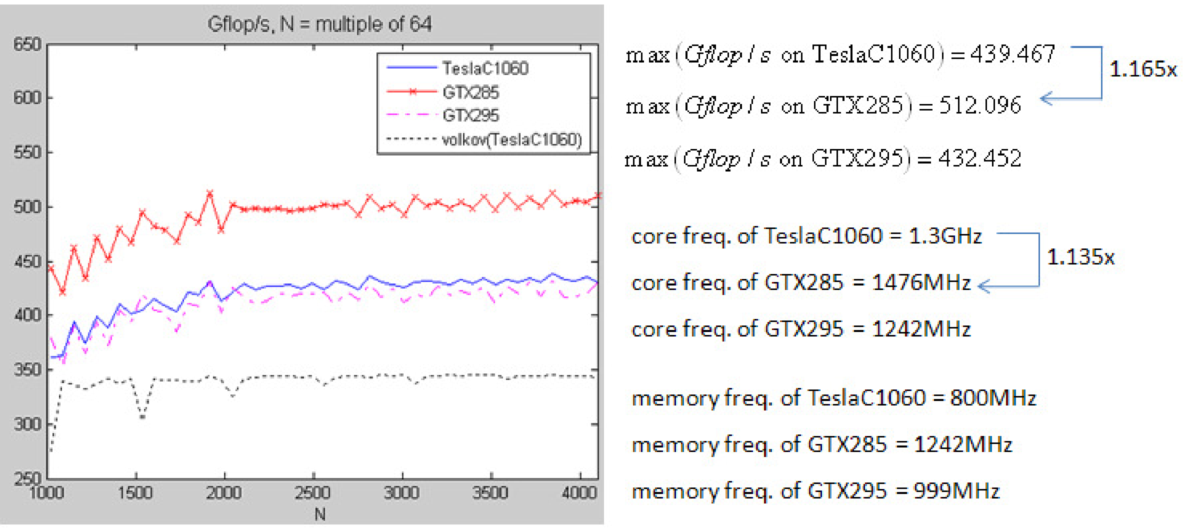
However situation is more complex, method 2 is not better than method 1 on overclocking cards, GTX285 and GTX295. Since we want to build SGEMM on GT200, including TeslaC160, GTX285 and GTX295, method 1 is better when three GPUs are considered together. So ranking of method 1 in Table 12 is one, the best choice.

algorithm	volkov	method 1 (variant)	method 2 (variant)	method 3 (variant)	method 4 (variant)
Resource usage					
Threads per block	64	64	128	64	64
Registers per thread	30	48	63	64	80
Shared memory per block	1168	1168	1680	1680	1176
Active threads per SM	512	320	256	256	192
Computational cost					
Load *A* to register "MOV reg, [gmem]"	$\frac{1}{16}mnk$	$\frac{1}{16}mnk$	$\frac{1}{24}mnk$	$\frac{1}{24}mnk$	$\frac{1}{16}mnk$

Load B to shared memory b	$\frac{1}{64}mnk$	$\frac{1}{64\times2}mnk$	$\frac{1}{128\times2}mnk$	$\frac{1}{64\times2}mnk$	$\frac{1}{64\times4}mnk$
Load shared b to register "MOV reg, [smem]"	0	$\frac{1}{2}mnk$	$\frac{1}{2}mnk$	$\frac{1}{2}mnk$	$\frac{1}{4}mnk$
MAD dest, [smem], src2, src3	mnk	0	0	0	0
MAD dest, src1, src2, src3	0	mnk	mnk	mnk	mnk
ranking	3	1	2	4	5

Table 12: resource usage and computational cost among five algorithms, volkov, method1_variant, method2_variant, method3_variant and method4_variant.

Figure 76 shows performance (Gflop/s) of method1on TeslaC1060, GTX285 and GTX295. The baseline is Volkov's code on TeslaC1060 (black dash line). Core frequency of GTX285 is 1.135x than that of TeslaC1060, and it is reasonable that performance of GTX285 is 1.165x than that of TeslaC1060. From Table 1 we know peak performance of single precision without dual issue on TeslaC1060 is 624 Gflop/s, and maximum performance of method 1 on TeslaC1060 in Figure 76 is 439.467Gflop/s. Hence method 1 achieves 70% of peak performance without dual issue.



$\max(Gflop/s \text{ on TeslaC1060}) = 439.467$

$\max(Gflop/s \text{ on GTX285}) = 512.096$

$\max(Gflop/s \text{ on GTX295}) = 432.452$

core freq. of TeslaC1060 = 1.3GHz

core freq. of GTX285 = 1476MHz

core freq. of GTX295 = 1242MHz

memory freq. of TeslaC1060 = 800MHz

memory freq. of GTX285 = 1242MHz

memory freq. of GTX295 = 999MHz

Figure 76: performance of method 1 over N = multiple of 64 on TeslaC1060, GTX285 and GTX295. The baseline is performance of Volkov's code on TeslaC1060.

As a result, method 1 has 10% ~ 20% improvement for large N indeed. However we cannot have a theoretical model to combine all factors, including latency of global memory and its throughput under different number of active threads per SM. It may be a coincidence that method 1 has better performance than Volkov's code .

In this work, we spend much effort on how to use **decuda/cudasm** to modify binary code as we explain in section 4. This is tedious and the modified binary code cannot be compatible with incoming Fermi architecture, so we must re-do the work again for Fermi architecture, TeslaC2050 and TeslaC2070. Hope that NVIDIA is going to release official **decuda/cudasm** at that time.

Acknowledgement

We want to thank Wladimir J. van der Laan and Sylvain Collange for opinion on **decuda/cudasm**, because this work could not be done without help of **decuda/cudasm**. Also we thank to Mark Harris for discussion of our idea and some hardware issues, Massimiliano Fatica for production of this report.

References

[1] Vasily Volkov, James W. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra. In SC '08: Preceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA, 2008, IEEE Press. source code can be downloaded from NVIDIA forum, http://forums.nvidia.com/index.php?showtopic=89084

[2] Wladimir J. van der Laan. Decuda and cudasm, the cubin utilities package, 2009.
http://wiki.github.com/laanwj/decuda/

[3] http://ati.amd.com/products/radeonx1k/whitepapers/X1800_Memory_Controller_Whitepaper.pdf

[4] NVIDIA Programming Guide, version 2.3

[5] David Kanter, NVIDIA's GT200: Inside a Parallel Processor
http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=9

[6] Greg Ruetsch, Paulius Micikevicius, Optimizing Matrix Transpose in CUDA,
source: \transposeNew\doc\MatrixTranspose.pdf in SDK

[7] NVIDIA on-line forum, thread about "question about latency of global memory" ,
http://forums.nvidia.com/index.php?showtopic=109558&pid=603432&mode=threaded&start=#entry603432

[8] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Henry Wong, Micro-benchmarking the GT200 GPU,
http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/microbenchmark_report.pdf

[9] NVIDIA on-line forum, thread about " latency of shared memory of Tesla C1060",
http://forums.nvidia.com/index.php?showtopic=152674

[10] Optimization, NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3