Hand-Tuned CGEMM on GT200 GPU

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

March 2010, version 1

Abstract: in [3], we improve SGEMM (matrix multiplication on single precision) of Volkov's code [1]. We test $C = AB$ on square matrices A, B and C. Our method reaches 440Gflop/s on TeslaC1060 whereas Volkov's code reaches 346 Gflop/s, which is 55.45% of peak performance (Volkov criticizes my experiment because he gets 60% of peak performance on GTX280, I am sorry that I cannot explain this difference so far).

In this paper, we extend the idea in SGEMM to CGEMM (matrix multiplication on complex type) and reach 445.7Gflop/s whereas CUBALS reaches 277.7Gflop/s, we have 37.69% improvement. Basic idea is to utilize amazing pattern, "MOV reg, [smem]" followed by two "MAD dest, src1, src2, src3". Volkov conjectures that such pattern would activate dual issue, he says "*One of them is streaming data from shared memory to registers behind register-only multiply-and-adds. Moves from shared memory to registers may go via SFUs on GT200, which would utilize "dual-issue" capability. This would avoid using the slow multiply-and-adds with shared memory operands, so might be faster.* " in the thread http://forums.nvidia.com/index.php?showtopic=47689&pid=1002936&mode=threaded&start=#entry1002936 on NVIDIA's online forum. In this work, we confirm this conjecture.

Although we reach 445.7Gflop/s on CGEMM, this number is near 439.467 Gflop/s we have in SGEMM [3], we still believe that CGEMM may reach 500 Gflop/s in some way.

The technique in this paper comes from workaround in [3],
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".
but we re-write rank-1 update in assembly code level (not inline assembly in PTX file but assembly code defined by Wladimir J. van der Laan, author of decuda/cudasm). This is a big breakthrough that we can design desired pattern ourselves, try many combinations of MOV, MUL, MAD, ADD in rank-1 update. This may give us a clue to add some patterns into compiler optimization.

Same as SGEMM in [3], we deliver CUDA binary code on 64-bit platform (it does not work on 32-bit platform). The CGEMM we deliver can work for any dimension. I must offer my heartfelt thanks to Wladimir J. van der Laan and Sylvain Collange, their package, **decuda/cudasm**, is crucial to this work.

Introduction

Matrix-multiplication $C = AB$ is a basic operation in BLAS library. Vasily Volkov and James W. Demmel provide a faster algorithm in 2008 [1]. We improve Volkov's code on SGEMM in [3] and in this paper, we further improve CGEMM based on Volkov's program.

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.

Basic operation in CGEMM is $c = a \times b + c$ where *a, b, c* are complex number. Volkov's method keeps *a* in register and fetches *b* from shared memory but uses only one MAD operation to execute $c = a \times b + c$ in SGEMM. However in CGEMM, a MAD operation on "complex" should be decomposed to several basic operations on "float". This give us a lot of degree of freedom to design a "good" pattern to execute "complex *c* = *a* * *b* + *c*". That is why we can achieve 445.7Gflop/s whereas CUBALS only reaches 277.7Gflop/s.

In this work, we try to invoke dual issue to hide a MUL operation into a MAD operation. Unfortunately, no benefit is seen because of RAW (Read-After-Write) hazard. On the contrary huge speedup appears if we use amazing pattern, "MOV reg, [smem]" followed by two "MAD dest, src1, src2, src3", found in [3].

The remaining sections are organized as follows: some preliminaries are introduced in section 1, including SPEC of GT200 (TeslaC1060, GTX285 and GTX295), pipeline latency and throughput of MAD operation, latency and throughput of global DRAM, block version of matrix-multiplication and outer-product based algorithm. Then we compare Volkov's code with CSGEMM in CUBLAS (CUDA 2.3) in section 2. Also we estimate peak performance of Volkov's code and propose how to invoke dual issue. Our man two ideas are introduced in section 3, one is using amazing pattern, the other is to avoid RAW hazard. In section 4, we mention method 1 which embed amazing pattern, and its variant, method1_variant, which is used to verify effectiveness of amazing pattern. Moreover we explain how to design a pattern in assembly level under help of package **decuda/cudasm.** Structure and performance of method 2 variant is shown in section 5, it is designed to avoid RAW hazard. In section 6, we try to combine advantages of idea 1 and idea 2. Finally we have some conclusions in section 7.

Table of Contents

1 preliminary

1.1 SPEC of GT200

In this work we use three GPUs listed in Table 1 to measure performance of CGEMM. 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 = $240(core) \times 1.3(core\ freq.) \times 3(flop\ count)$

Main cost in SGEMM is MAD and dual issue [8] can be neglected since it is unlikely to merge MAD and MUL in flight due to few MUL operations in SGEMM. However we may activate dual issue in CGEMM since one "*c*+=*a***b*" has 4 MUL and 4 ADD, it is intuitive that CGEMM would reach higher Gflop/s. Unfortunately, CUBLAS (CUDA 2.3) has better Gflop/s on SGEMM (SGEMM reaches 344Gflop/s whereas CGEMM reaches 277.7Gflop/s), that is why we focus on CGEMM again.

Even CGEMM can utilize power of dual issue, it is impossible to exceed single precision performance without dual issue (shown later in section 2.3). Hence we also report single precision performance without dual issue which is upper bound of CGEMM that we can pursue.

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

	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
DRAM (MByte)	896	1024	4096

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

1.2 pipeline latency and throughput

From result of micro-benchmarking in [9], we can summarize latency and throughput of some instructions in Table 2.

Instruction	Type	Latency (cycles)	Throughput (cycles/warp)
MOV reg, [gmem]		530	4
MOV [smem], reg		36	4
ADD, SUB	Float	24	4

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

MAD dest, src1, src2, src3	Float	31.5	4
MAD dest, [smem], src2, src3	Float	34.6	6

Table 2: latency and throughput of arithmetic instructions on TeslaC1060, authors in [9] use (ops/clock) as unit of throughput whereas we use (cycles/warp) as unit.

Remark 1: we have confirmed these numbers in [3].

1.3 notation of matrices and partition of grid, block

We adopt notations in Figure 1 which is the same as we use in [3]. 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 CGEMM, including $C=\alpha AB+\beta C$, $C=\alpha A^TB+\beta C$, $C=\alpha AB^T+\beta C$, $C=\alpha A^TB^T+\beta C$ 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*: $MN\times b_Mb_N=mn$

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

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

(4) number of flop ($c=a\cdot b+c$): $MNK\times b_Kb_Mb_N\times 8(4\ MUL+4\ ADD)=mnk$ independent of grid dimension.

$c.x\leftarrow a.x\times b.x-a.y\times b.y+c.x$ 2 MUL and 2ADD

$c.y\leftarrow a.x\times b.y+a.y\times b.x+c.y$ 2 MUL and 2ADD

Moreover in this work, we focus discussion on $C=AB$ but deliver source code to deal with $C=\alpha AB+\beta C$.

Remark 2: CGEMM is more compute-intensive than SGEMM because flops of CGEMM is 8 times flops of SGEMM but CGEMM is only two times size of SGEMM when dimension of matrices are the same.

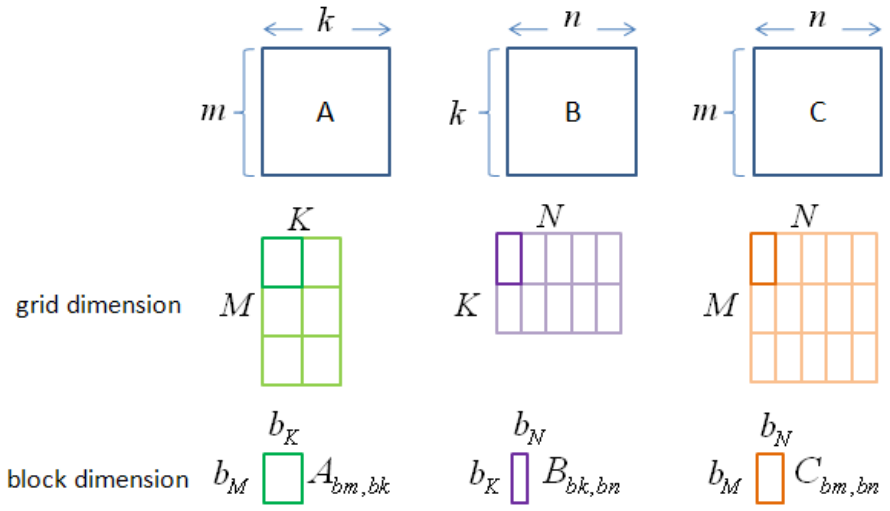


Figure 1: dimension of matrices A, B, C and execution configuration.

1.4 outer-product based algorithm

In this work, we inherit framework of Volkov's source code but modify it in order to avoid local memory usage.

Volkov's code uses outer-product[2] formulation

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

which fetches one column of matrix *A* into registers and then save one movement from shared memory to register. 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.

To sum up, we can write down pseudo-code of two outer-product formulations in Figure 2. In this work we adopt algorithm (I) and parameters of grid and block are depicted in Figure 3.

Remark 3: in Figure 3, we use registers *c[32]* to represent 16 complex numbers, real part locates at even index and imaginary part locates at odd index, $C_{bm,bn}(i,j) = c[2j] + \sqrt{-1} \cdot c[2j+1]$. Moreover in order to save number of transactions of matrix *B,* we use shared memory *b1* to represent real part and *b2* to represent imaginary part, i.e.

$B_{bk,bn}(i,j) = b1[i][j] + \sqrt{-1} \cdot b2[i][j]$

[2] difference between inner-product based algorithm and outer-product based algorithm is described in [3]. We don't implement inner-product based algorithm, it may has better Gflop/s than that of SGEMM because CGEMM is more compute-intensive than SGEMM.

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

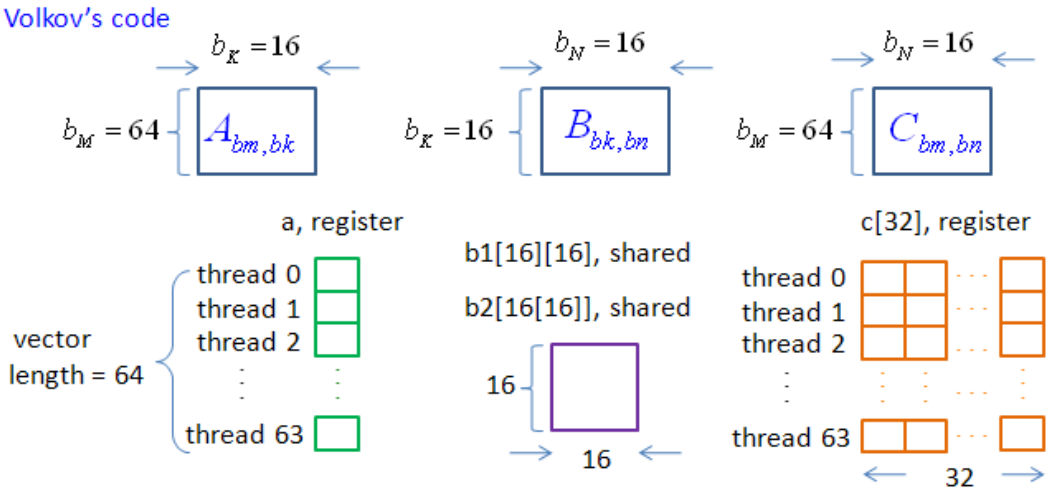


Figure 3: parameters of grid and block in (I) of Figure 2

2 CUBLAS versus Volkov's code

2.1 modification of Volkov's code

In this work, we use Volkov's code with boundary condition checker in [3] but minor modification to match framework in Figure 3. We have known that Volkov's code uses outer-product formulation, algorithm (I) in Figure 2,

and term by term correspondence in Figure 4 can be described as following:

(1) load *16x16* block of matrix *B* into shared memory *float b1[16][16]* and *float b2[16][16]* such that

$B_{bk,bn}(i,j) = b1[i][j] + \sqrt{-1} \cdot b2[i][j]$. One thread block has 64 threads (vector length = 64), each thread loads four elements (526/64 = 4) in *b1* or *b2*. If we use *complex b[16][16]* and use "*b[inx][iny+i] = B[i*idb]*" , then the number of global memory transaction on matrix *B* is doubled due to interleaving property of complex type. Here variable *sel* is used to check boundary condition.

(2) each thread loads one element (8 bytes) 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 $C_{bm,bn}$.

" *Complex A_reg = A[0]*" uses 128-byte transaction (each thread loads 8 bytes), this can be confirmed by result of decuda (" *Complex A_reg = A[0]*" is translated to "*mov.b64 reg, g[reg]*").

(3) store $C_{bm,bn}$ into matrix *C*. In order to utilize 128-byte transaction, we use two-step coding

Step 1: compute $complex\ cc \leftarrow \alpha c_{reg}$ by

$$cc.x \leftarrow \alpha.x \cdot c[2j] - \alpha.y \cdot c[2j+1]$$

$$cc.y \leftarrow \alpha.y \cdot c[2j] + \alpha.x \cdot c[2j+1]$$

Step 2: store cc to global matrix C by

$C[ldc] += cc$ // use 128-byte transaction

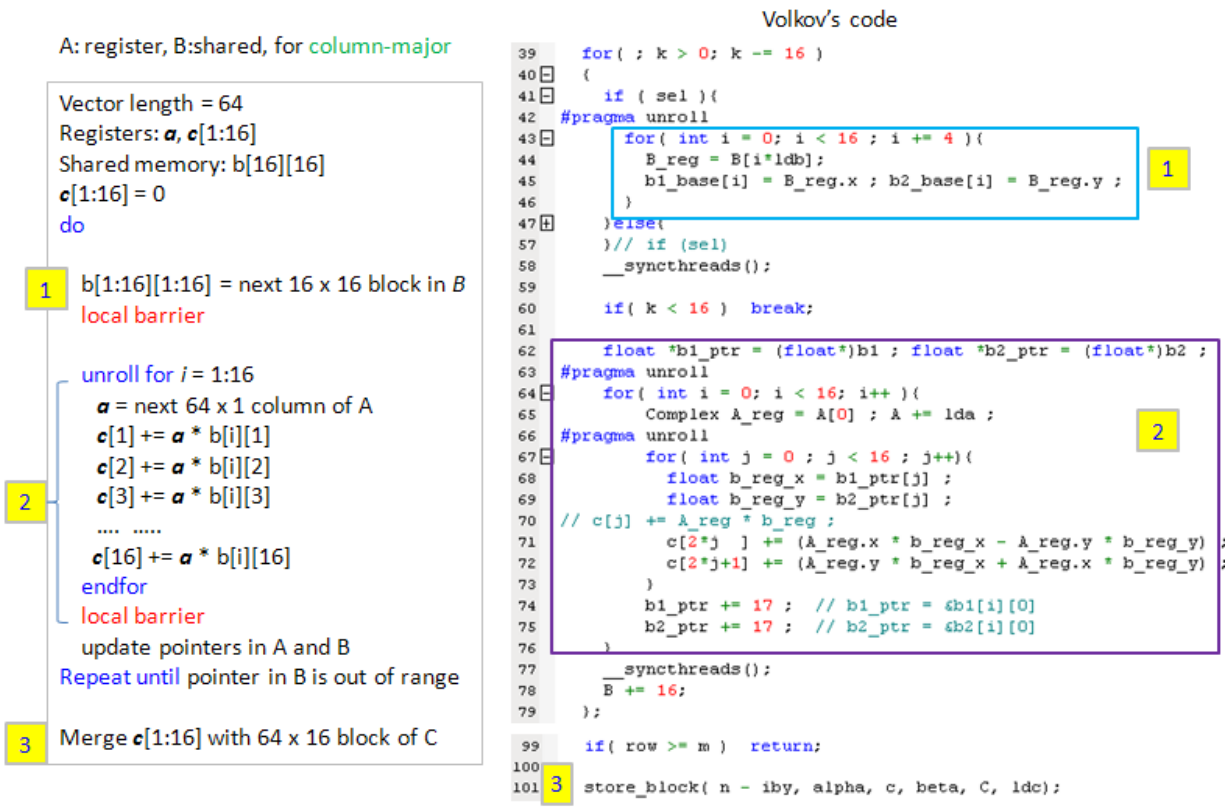


Figure 4: program structure of Volkov's code

Furthermore in order to simplify discussion, we use carton picture to describe grid information of Volkov's code as you see in Figure 5 (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: $mnk\dfrac{1}{b_M} = \dfrac{mnk}{64}$, and

(3) number of flop of $c = a \cdot b + c$: $8mnk$.

$c.x \leftarrow a.x \times b.x - a.y \times b.y + c.x$ 2 MUL and 2ADD

$c.y \leftarrow a.x \times b.y + a.y \times b.x + c.y$ 2 MUL and 2ADD

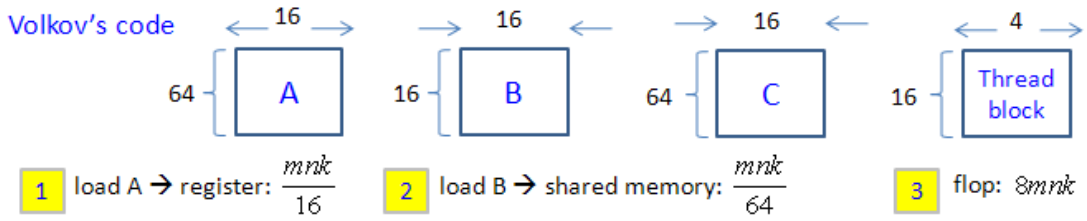


Figure 5: carton picture of grid information of Volkov's code.

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. From experience in SGEMM [3], we know compiler ***nvcc*** always choose shared-memory MAD. This property also holds in CGEMM. ***nvcc*** uses 2 MUL, 2 MAD and 2 ADD to implement "complex c+=a*b " in Figure 6. Our expectation is to hide 2 MUL operations by dual issue, then effective flop count is only 2 MAD and 2ADD per "complex c+=a*b ". We will discuss this in section 2.3.

(I)

$c.x = a.x \times b.x - a.y \times b.y + c.x$

$c.y = a.x \times b.y + a.y \times b.x + c.y$

(II)

$reg \leftarrow a.y \times b.y$

$reg \leftarrow a.x \times b.x - reg$

$c.x \leftarrow c.x + reg$

$reg \leftarrow a.y \times b.x$

$reg \leftarrow a.x \times b.y + reg$

$c.y = c.y + reg$

(III)

```
mul.rn.f32 $r49, s[$ofs1+0x0020], $r7
mad.rn.f32 $r49, s[0x0060], $r6, -$r49
add.rn.f32 $r47, $r47, $r49
mul.rn.f32 $r49, s[0x0060], $r7
mad.rn.f32 $r49, s[$ofs1+0x0020], $r6, $r49
add.half.rn.f32 $r46, $r46, $r49
```

Figure 6: *(I)* operation of "complex c+=a*b". *(II)* corresponding operations in Volkov's code, the execution sequence is MUL, MAD, ADD. *(III)* assembly representation of *(II)* from decuda.

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, one "complex *c*+=*a***b* " has 8 flops, so we define flop count as $Gflop/s = \frac{8mnk}{time(s)}$

Definition 2: let ***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 7 shows Gflop/s over $N = 5:4096$ and Table 3 shows Gflop/s on specific dimension $N = 256, 512, 1024, 2048, 4096$. Clearly, Volkov's code has better performance than CUBLAS, about 20% improvement. There are two points that CUBLAS is not good. First, compared with SGEMM, performance of CUBLAS on CGEMM is poor than performance on SGEMM which reaches 344Gflop/s. This is not reasonable since CGEMM can utilize power of dual issue but SGEMM cannot. Second, both SGEMM and CGEMM in CUBLAS have large variation on graph of Gflop/s. We believe that such variation comes from number of memory transactions which strongly depends on leading dimension of matrices, even in SGEMM, Volkov's code has fluctuation on Gflop/s. However CGEMM is more compute-intensive than SGEMM, if one adopts the same algorithm, why does not CGEMM have uniform performance? (Volkov's code has uniform performance on CGEMM.)

Hence it is reasonable to take Volkov's code as baseline since performance of Volkov's is more uniform than performance of CUBLAS. Someone may ask "Could you convince me that Volkov's code can be improved further, because it is 20% faster than CUBLAS?". There are two main reasons

(1) number of registers per thread of Volkov's code is 51, so number of active threads per SM is only 256. From experience on SGEMM [3], register usage of CGEMM of Volkov's code should be the same as register usage of SGEMM of method 1 in [3]. If we can organize register usage ourselves, then only 48 registers per thread are required to compile CGEMM. If register count is 48, then number of active threads can achieve 320.

(2) Volkov's code on CGEMM has the same performance as on SGEMM in Table 4. Again this is not reasonable because CGEMM should utilize dual issue. We will discuss theoretical bound of Volkov's code under dual issue in section 2.3

N	CGEMM,CUBLAS (Gflop/s)	CGEMM, Volkov (Gflop/s)	***R***
256	210.317	187.892	1.1194
512	260.288	265.591	0.9800
1024	273.432	329.555	0.8297
2048	276.788	342.854	0.8073
4096	277.708	348.489	0.7969

Table 3: CGEMM comparison between CUBLAS and Volkov's code for *N* = 256, 512, 1024, 2048, 4096 on TeslaC1060.

N	SGEMM, CUBLAS (Gflop/s)	SGEMM, 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: SGEMM comparison between CUBLAS and Volkov's code on TeslaC1060.

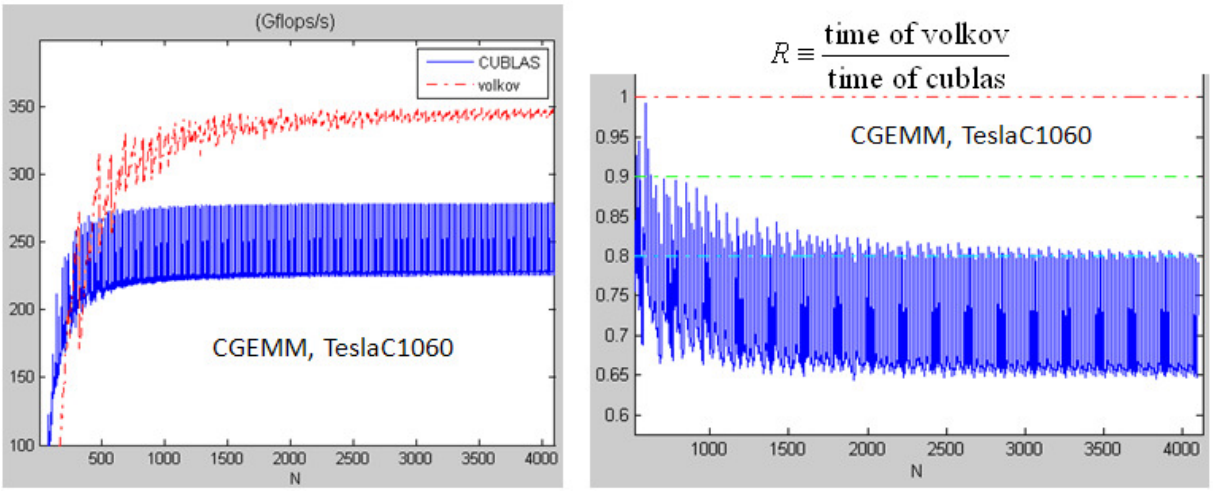


Figure 7: left panel is Gflop/s of CUBLAS (CUDA 2.3) and Volkov's code. Right panel is performance improvement. Platform is TeslaC1060.

2.3 theoretical performance of Volkov's code

CGEMM is compute-intensive, it is reasonable to neglect overhead of data transfer (or assume that we can hide memory latency in arithmetic pipeline) and focus on how many arithmetic operations are done. We have shown number of flops per "complex *c*+=*a***b* " in CGEMM is 8, or equivalently say 4 MADs. We can also estimate lower bound and upper bound of Volkov's code, the former is considered without dual issue (left panel in Figure 8) and the latter is considered with dual issue (right panel in Figure 8). Moreover we know cost of each operation shown in Table 2, let use define

(1) cost of "MAD dest, src1, src2, src3" as $T_{MAD,reg} = 4 cycle / warp$

(2) cost of "MAD dest, [smem], src2, src3" as $T_{MAD,smem} = 1.5\ T_{MAD,reg} = 6 cycle / warp$

(3) cost of "MUL dest, src1, src2" as $T_{MUL,reg} = 4 cycle / warp$

(4) cost of "MUL dest, [smem], src2" as $T_{MUL,smem} = 1.5\ T_{MUL,reg} = 6 cycle / warp$

(5) cost of "ADD dest, src1, src2" as $T_{ADD,reg} = 4 cycle / warp$

Also single precision performance without dual issue on TeslaC1060 is 624 Gflop/s, or say $\frac{1\ MAD}{T_{MAD,reg}} = 624 Gflop / s$

(1) without dual issue

peak performance is $\frac{4MAD}{2T_{MUL,smem}+2T_{MAD,smem}+2T_{ADD,reg}}=\frac{4}{8}\frac{MAD}{T_{MAD,reg}}=\frac{4}{8}\times 624Gflop/s=312Gflop/s$

(2) with dual issue

peak performance is $\frac{4MAD}{2T_{MAD,smem}+2T_{ADD,reg}}=\frac{4}{5}\frac{MAD}{T_{MAD,reg}}=\frac{4}{5}\times 624Gflop/s=499.2Gflop/s$

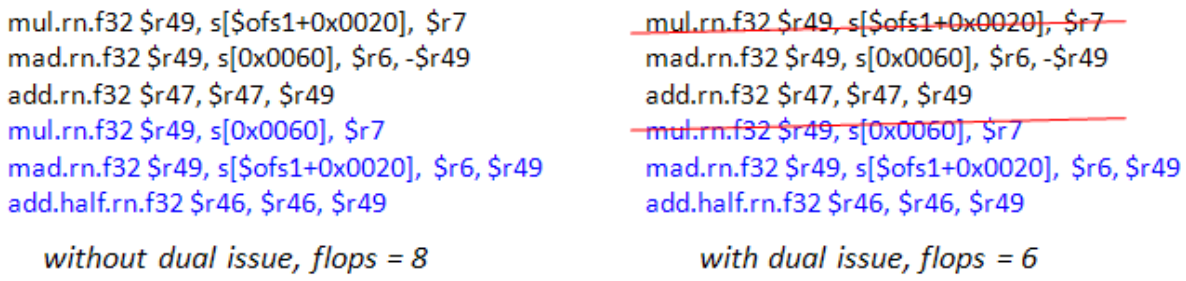


Figure 8: left panel (no dual issue): 8 flops per "complex *c+=a*b* ", including 2 MUL, 2 ADD, 2MAD.

right panel (with dual issue): 6 flops since 2 MUL is hidden in dual issue.

Remark 4: from Table 3, Volkov's code only reaches 348.489 Gflop/s which is near 312Gflop/s, this means that Volkov's code does not utilize dual issue very well. Of course we can remove effect of fewer active threads per SM so far.

2.4 MUL and MAD interleaving manually

If one looks at assembly code (from result of decuda, not PTX code) of Volkov's code, then ***nvcc*** does not produce regular pattern for rank-1 update

```
    for( int i = 0; i < 16; i++ ){
        Complex A_reg = A[0] ; A += lda ;
// rank-1 update for c[j] += a[i] * b[i][j]
        for( int j = 0 ; j < 16 ; j++){
            float b_reg_x = b1_ptr[j] ;
            float b_reg_y = b2_ptr[j] ;
// c[j] += A_reg * b_reg ;
            c[2*j   ] += (A_reg.x * b_reg_x - A_reg.y * b_reg_y) ;
            c[2*j+1] += (A_reg.y * b_reg_x + A_reg.x * b_reg_y) ;
        }
        b1_ptr += 17 ;    // b1_ptr = &b1[i][0]
        b2_ptr += 17 ;    // b2_ptr = &b2[i][0]
    }
```

and experimental result only shows 348.489 Gflop/s. It is reasonable to manipulate pattern of rank-1 update ourselves if we can write assembly instructions at will and ***cudasm*** can work well. As we know, if we want to activate dual issue,

then best approach is to interleave MUL and MAD [6]. In (a) of Figure 9, we re-write one "*c = a * b + c*",

```
c[2*j    ] += (A_reg.x * b_reg_x - A_reg.y * b_reg_y) ;
c[2*j+1] += (A_reg.y * b_reg_x + A_reg.x * b_reg_y) ;
```

, which has two MUL, two MAD and two ADD, to a sequence of "MUL, MAD, MUL, MAD, ADD, ADD". We call this method as "volkov + unroll 1", here "unroll" means that we unroll the loop of rank-1 update.

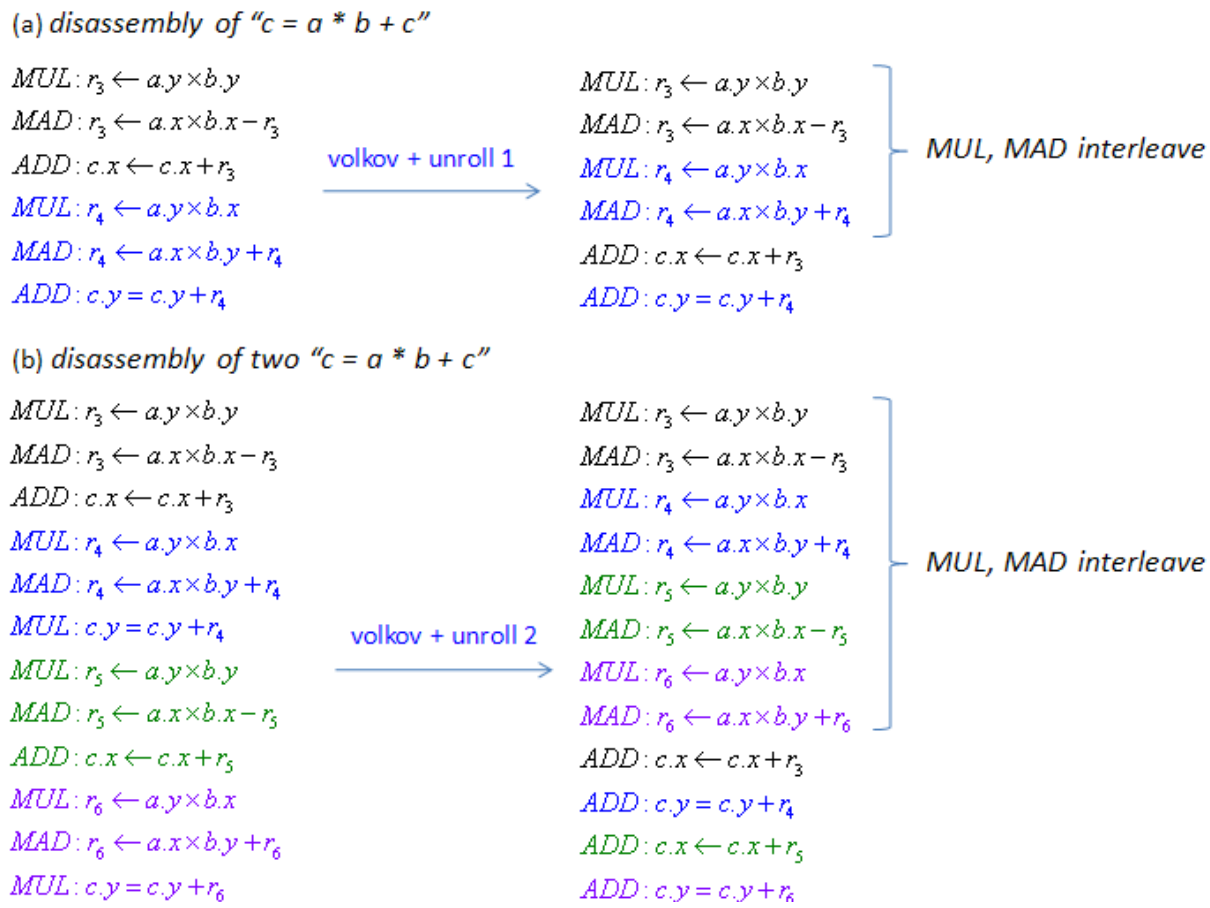


Figure 9: rearrange assembly instructions such that MUL and MAD are interleaved each other in a long sequence.

Similarly we can unroll two consecutive "*c = a * b + c*" . In (b) of Figure 9, we unroll the loop

```
// rank-1 update for c[j] += a[i] * b[i][j]
        for( int j = 0 ; j < 16 ; j++){
            float b_reg_x = b1_ptr[j] ;
            float b_reg_y = b2_ptr[j] ;
// c[j] += A_reg * b_reg ;
            c[2*j    ] += (A_reg.x * b_reg_x - A_reg.y * b_reg_y) ;
            c[2*j+1] += (A_reg.y * b_reg_x + A_reg.x * b_reg_y) ;
        }
```

two times and then MUL-MAD interleaving sequence becomes longer. We call this method as "volkov + unroll 2". We

expect that the longer a MUL-MAD interleaving sequence is, the faster the program is. In order to keep register count less than 64, we can unroll the loop 8 times, so "volkov + unroll 8" should be fastest, and near optimal performance 499.2 Gflop/s. However this conjecture is wrong, from Table 5, "volkov + unroll 8" is the slowest among all methods.

	volkov	Volkov + unroll 1	Volkov + unroll 2	Volkov + unroll 4	Volkov + unroll 8
Registers per thread	51	48	48	51	59
Active threads per SM	256	320	320	256	256
Gflop/s for $N = 4096$	348.489	347.991	347.615	325.859	325.628

Table 5: profile CGEMM, C = A*B for square matrices with dimension $N = 4096$ on TeslaC1060. We expect that the longer a MUL-MAD interleaving sequence is, the faster the program is. However our desired pattern "volkov + unroll 8" is slower than original volkov (optimization by nvcc).

I think that what disables dual issue is RAW (Read-After-Write) hazard. In David Kanter's article, NVIDIA's GT200: Inside a Parallel Processor, Shader Multiprocessor Architecture is depicted in Figure 10.

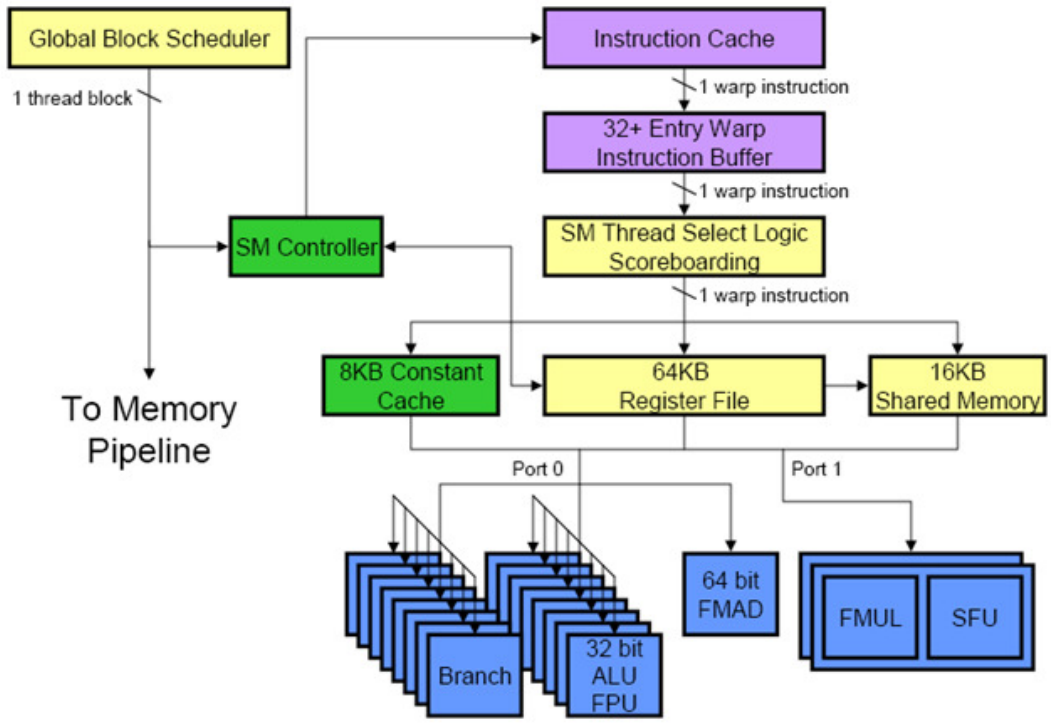


Figure 10: Shader Multiprocessor Architecture in [7]

It says that *warp instructions are fetched into a multithreaded instruction buffer, which probably contains 32 or 64 entries – one to two entries per warp in-flight*. In GT200 architecture, maximum number of active warps per SM is 32 (or equivalently maximum number of active threads per SM is 1024), in this paper we assume that instruction buffer has 64 entries such that each warp can put two consecutive instructions into this buffer. The author, David Kanter,, also describes two properties on execution of instructions,

Property 1: The instruction issue logic is responsible for selecting a warp instruction to issue **each cycle**. Each cycle the issue logic selects and forwards the highest priority 'ready to execute' warp instruction from the buffer.

Prioritization is determined with a **round-robin** algorithm between the 32 warps that also accounts for warp type, instruction type and other factors.

Property 2: A warp which has multiple ready instructions can continue to issue until the scoreboarding blocks further progress or another warp is selected for issue. (or called out-of-order completion). For example, a warp could issue a long latency memory instruction, followed by a computational instruction and in that case, the computation would end up writing back its results before the memory instruction.

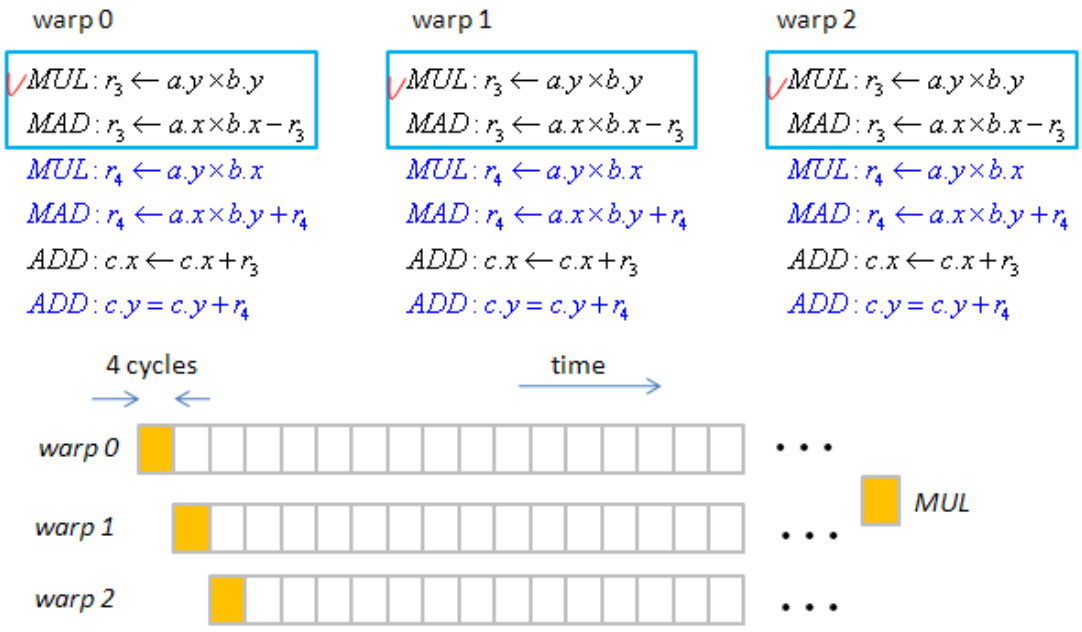


Figure 11: Gatt chart of three warps in one SM. blue rectangle is a window in instruction buffer which contains two consecutive instructions of each warp. The first instruction MUL, is executed warp-by-warp due to RAW hazard. At this time dual issue is disable.

For simplicity, let us assume pipeline latency is 12 cycles (it should be 24 cycles at least) and only three warps per SM. First we consider "volkov + unroll 1", first two instructions per warp are put into instruction buffer (blue rectangle in Figure 11) since we have assume that each warp can put two consecutive instructions into instruction buffer. It is easy to show that three warps execute their first MUL instruction in turn (according to property 1, round-robin), no dual issue is invoked due to RAW hazard.

After completion of first MUL instruction of three warps, we may expect dual issue if priority scheme of instruction issue logic is good. In Figure 12, we can combine MAD of warp 0 and MUL of warp 1 to be a dual-issue packet and executed at the same time. Similarly, MAD of warp 1 and MUL of warp 2 are executed in dual issue sense. Finally MAD of warp 2 and MUL of warp 0 are executed. Then we only use 24 cycles to execute 6 MUL and 3 MAD, that means that we save cost of 3 MUL. Hence peak performance of "volkov + unroll 1" is

$$\frac{4MAD}{1T_{MUL,smem} + 2T_{MAD,smem} + 2T_{ADD,reg}} = \frac{4}{6.5}\frac{MAD}{T_{MAD,reg}} = \frac{4}{6.5} \times 624Gflop/s = 384Gflop/s$$

However "volkov + unroll 1" only reaches 348Gflop/s, much smaller than 384 Gflop/s. Either our interpretation is

wrong or instruction issue logic does not do well.

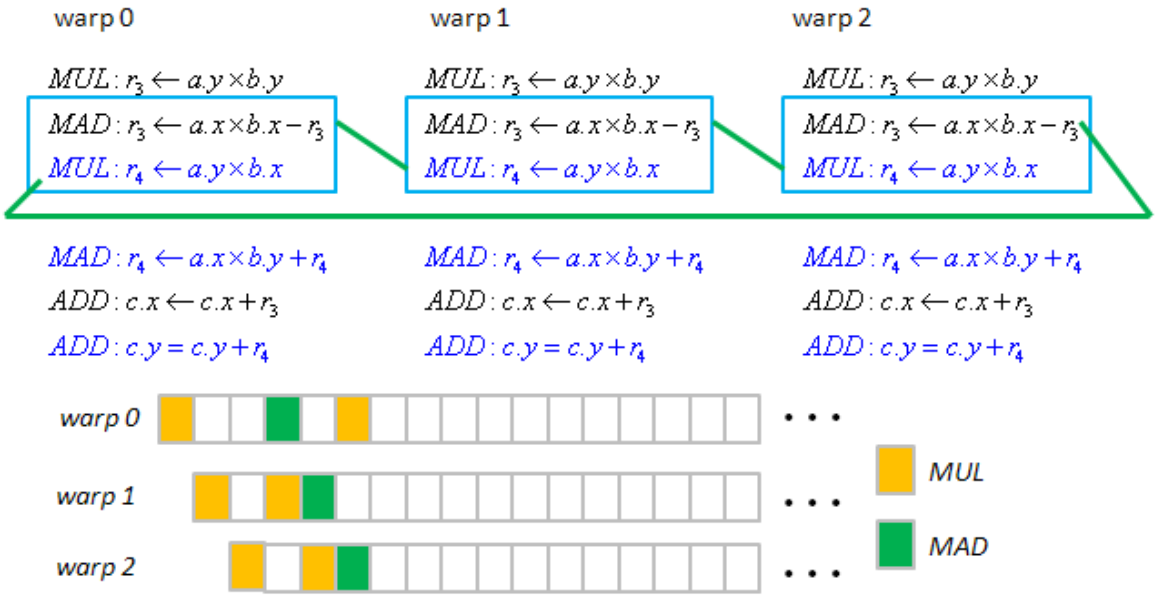


Figure 12: Gatt chart after completion of first MUL instruction. Now window moves one instruction further and dual issue can be invoked now if priority scheme is good enough. MAD of warp 0 can combine MUL of warp 1 to be a dual-issue packet and so on.

3 basic idea

From above discussion, we know Volkov's code is not optimal, combination of instructions and hazards must be taken care. Under manipulation of assembly code ourselves, register count per thread is not greater than 48, so number of active threads per SM is 320. We propose three ideas to improve CGEMM,

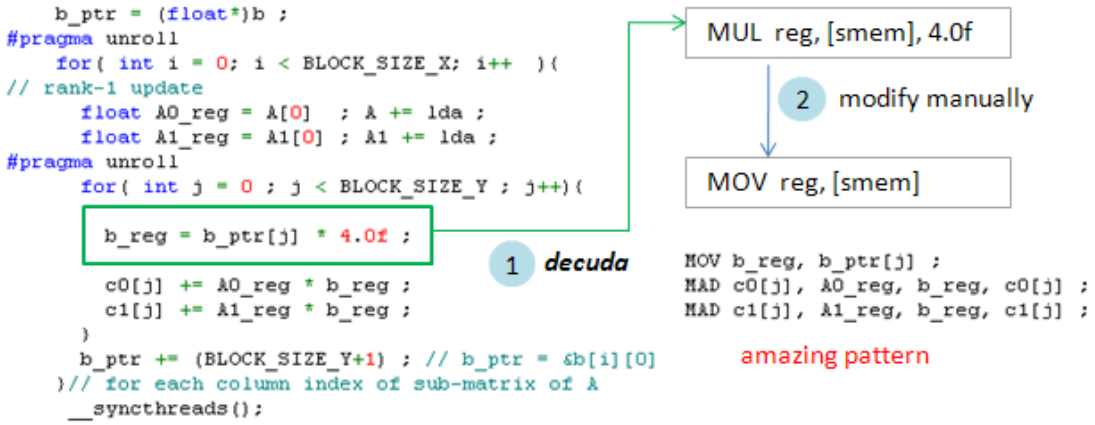


Figure 13: amazing pattern in [3], "MOV reg, [smem]" and followed by two "MAD dest, src1, src2, src3".

Idea 1: use amazing pattern, "MOV reg, [smem]" followed by two "MAD dest, src1, src2, src3", which is found in SGEMM [3]. In SGEMM, such pattern is deduced from "two $c = a*b + c$ shares one movement from shared memory to register". However in CGEMM, one "$c = a*b + c$" can produce this pattern if we modify

$c.x \leftarrow a.x \times b.x - a.y \times b.y + c.x$
$c.y \leftarrow a.x \times b.y + a.y \times b.x + c.y$
to
$c.x \leftarrow a.x \times b.x - (a.y \times b.y - c.x)$
$c.y \leftarrow a.x \times b.y + (a.y \times b.x + c.y)$
. We will describe this form in method 1 and method1_variant.

Idea 2. remove RAW hazard to increase possibility of dual issue. In order to achieve this goal, we only use one MUL per "*c = a*b +c"* (original Volkov's code use 2 MUL per "*c = a*b +c"*). method 2_varinat uses this idea.

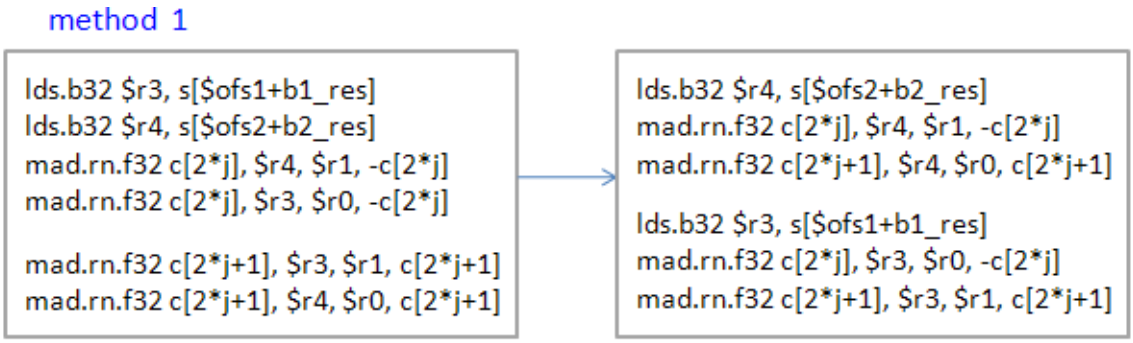
Idea 3: mix idea 1 and idea 2 to increase possibility of dual issue without RAW hazard. method 3 and method 4 use this idea.

Remark 5: Although we design method 1, method 2, method 3 and method 4, they have same block structure as Volkov's code in Figure 5 except pattern of rank-1 update.

4 idea 1: amazing pattern in method 1

If we write "complex *c = a*b +c"* as
$c.x \leftarrow a.x \times b.x - (a.y \times b.y - c.x)$
$c.y \leftarrow a.x \times b.y + (a.y \times b.x + c.y)$
, then two amazing patterns are generated. In Figure 14, we load element of matrix A to 64-bit register, say $(\$r_0, \$r_1) \leftarrow A[0]$ corresponding to C code "Complex A_reg = A[0]". Note that GT200 has only 32-bit registers, CUDA uses two consecutive 32-bit registers as one 64-bit register, however, one must be careful since ID of first 32-bit register must be odd or kernel launch failure occurs. Second we load sub-matrix of B from shared memory into registers by $(\$r_3, \$r_4) \leftarrow (b.x, b.y)$ which corresponds to C code " float b_reg_x = b1_ptr[j] ; float b_reg_y = b2_ptr[j] ; ". Then we can use four "MAD dest, src1, src2, src3" and two "MOV reg, [smem]" to implement one "complex *c = a*b +c"* which corresponds to C code

*c[2*j] += (A_reg.x * b_reg_x - A_reg.y * b_reg_y) ;*
*c[2*j+1] += (A_reg.y * b_reg_x + A_reg.x * b_reg_y) ;*



$$c.x \leftarrow a.x \times b.x - a.y \times b.y + c.x$$
$$= a.x \times b.x - (a.y \times b.y - c.x)$$
$$c.y \leftarrow a.x \times b.y + a.y \times b.x + c.y$$
$$= a.y \times b.x + (a.x \times b.y + c.y)$$

Figure 14: amazing pattern in method 1

Note that method 1 does not use any MUL operation, so theoretical peak performance of method 1 is

$$\frac{4MAD}{2\,load + 4T_{ADD,reg}} = \frac{4}{6}\frac{MAD}{T_{MAD,reg}} = \frac{4}{6}\times 624Gflop/s = 416Gflop/s\,.$$

4.1 performance of method 1

Performance of method 1 compared to Volkov's code is shown in Table 6 and Figure 15. Method 1 is very good, 20% faster than Volkov's code and moreover outperform against theoretical peak performance, 416 Gflop/s. Method 1 reaches 445 Gflop/s, this number cannot be interpreted except dual issue. So now we have a strong evidence to say amazing pattern, "MOV reg, [smem]" and followed by two "MAD dest, src1, src2, src3", would activate dual issue.

N	Volkov (Gflop/s)	Method 1 (Gflop/s)	R
256	195.053	193.956	0.9687
512	275.532	364.928	0.7278
1024	332.228	414.244	0.7956
2048	345.122	436.673	0.7852
4096	349.832	445.724	0.7818

Table 6: Comparison between Volkov's code and method 1 on CGEMM, C = A*B for square matrices. Platform is TeslaC1060. Method 1 outperforms Volkov's code, even breaks the limit of peak performance without considering dual issue.

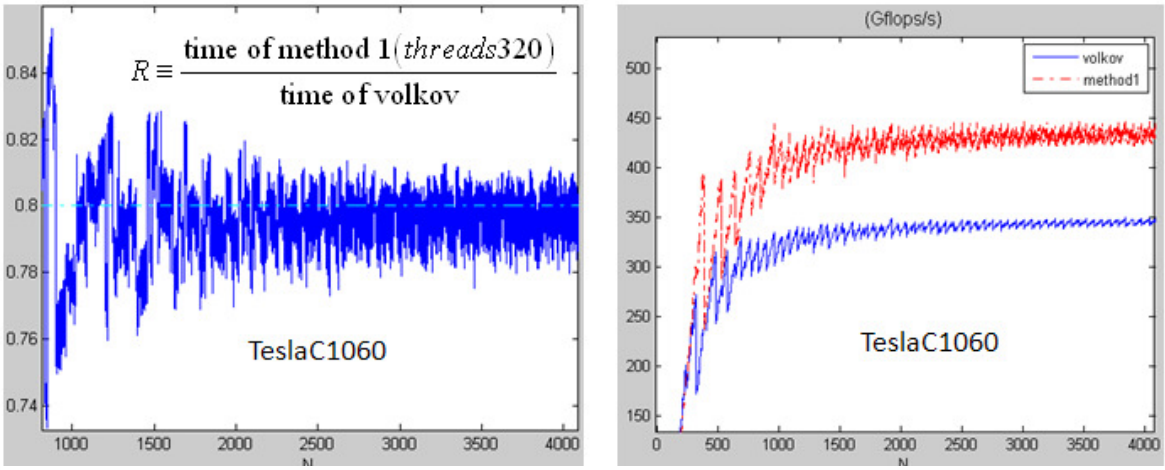


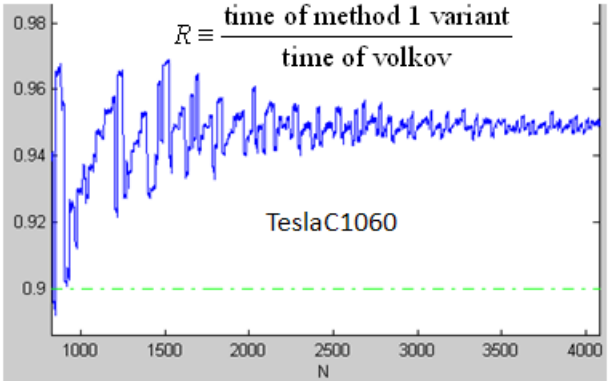
Figure 15: performance of method 1. it is 20% faster than Volkov's code on TeslaC1060.

4.2 method1_variant

In order to check dual issue of amazing pattern, we set up a reference experiment, called method1_varinat, which is the same as method 1 except using "MAD dest, [smem], src2, src3",

```
mad.rn.f32 c[2*j], s[$ofs2+b2_res], $r1, -c[2*j]
mad.rn.f32 c[2*j+1], s[$ofs1+b1_res], $r1, c[2*j+1]
mad.rn.f32 c[2*j], s[$ofs1+b1_res], $r0, -c[2*j]
mad.rn.f32 c[2*j+1], s[$ofs2+b2_res], $r0, c[2*j+1]
```

Theoretical peak performance of Method1_variant is $\frac{4MAD}{4T_{ADD,smem}} = \frac{4}{6}\frac{MAD}{T_{MAD,reg}} = 416Gflop/s$, the same as that of method 1. However Figure 16 shows bad performance of method1_variant, only 5% faster than Volkov's code.



N	Volkov (Gflop/s)	Method 1 variant (Gflop/s)	R
256	195.053	218.598	0.8595
512	275.532	329.961	0.8049
1024	332.228	353.154	0.9332
2048	345.122	363.862	0.9423
4096	349.832	367.029	0.9495

Figure 16: performance of method1_variant. it is almost the same speed as Volkov's code on TeslaC1060.

4.3 implementation of method 1

To design pattern of rank-1 update , we need to modify assembly code ourselves and use **decuda/cudasm** to complete the work. However the package ***decuda/cudasm*** is kept going and does not guarantee correctness for any combination of instructions, so we use the trick "*b_reg = b_ptr[j] * 4.0f* " in SGEMM [3] and then modify "MUL reg, [smem], 4.0f " to "MOV reg, [smem]", which is supported by **cudasm**. In this work we do more aggressively, follow the same procedure is depicted in Figure 17, and replace whole code segment X by W. W is designed ourselves and contains the pattern of rank-1 update. Of course, it is necessary to check validity of **cudasm** on W.

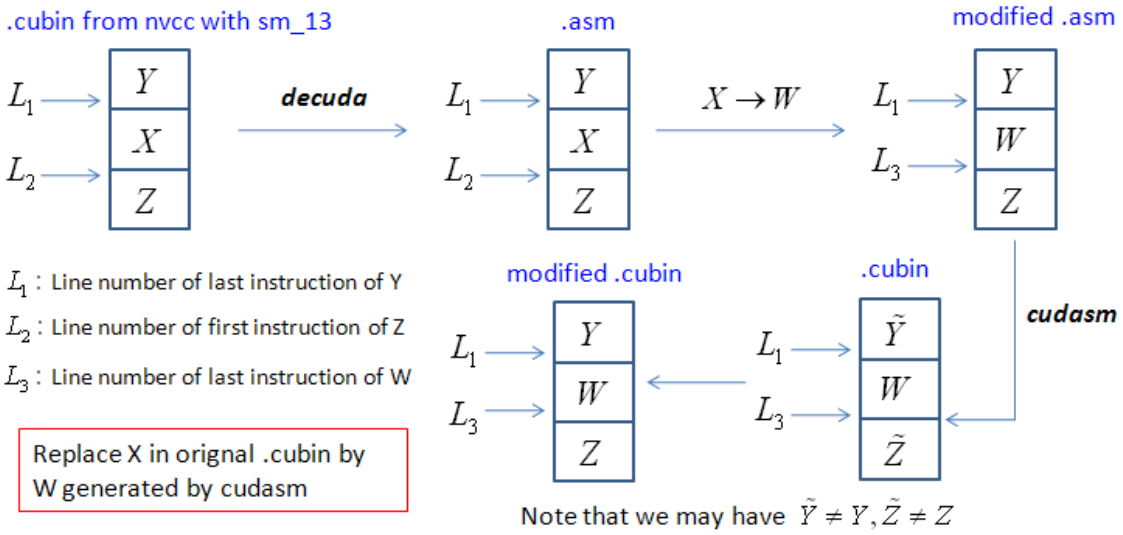


Figure 17: generic procedure of the workaround in method 1.

First we check what kind of codes ***cudasm*** cannot translate well. More precisely if ψ is (a segment of) binary code,

then we want to check if $cudasm(decuda(\psi)) \neq \psi$ or not. One thing must be mentioned: even $cudasm(decuda(\psi)) \neq \psi$, the code $cudasm(decuda(\psi))$ may work well (We have shown this in [3]). Our purpose is to find minimum set that $cudasm(decuda(\psi)) \neq \psi$ and $cudasm(decuda(\psi))$ does wrong job.

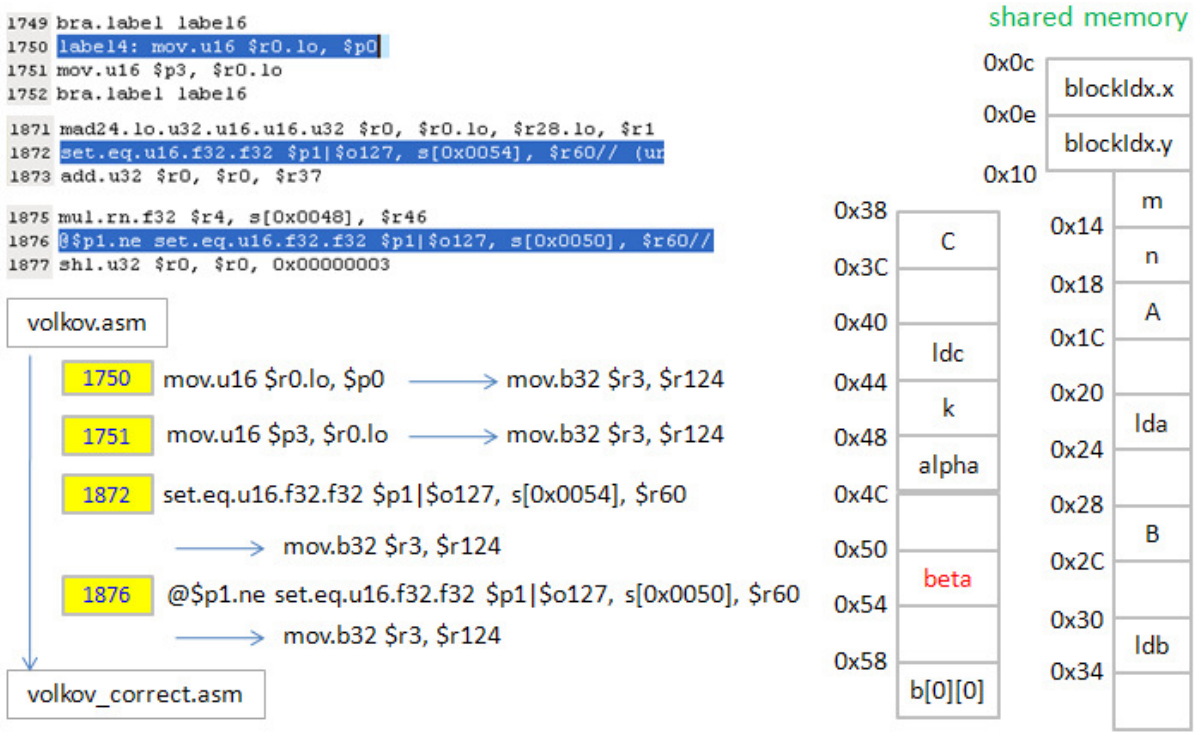


Figure 18: ***cudasm*** does not work at four assembly instructions

Step 1: We use ***cudasm*** to assemble *volkov.asm* file (method 1 is the same as Volkov's code except pattern of rank-1 update) generated by ***decuda***, then four errors occur, see Figure 18,

(1) Error on line 1750: Invalid argument types

(2) Error on line 1751: Invalid argument types

(3) Error on line 1872: Type conflict -- expected half register

(4) Error on line 1876: Type conflict -- expected half register

Fortunately line 1750, 1751, 1872, 1876 are in code segment Z ,not in code segment X , we can change these four instructions to any valid instructions (for example, line 937 "mov.u16 $r1.hi, $p0" is substituted by " mov.b32 $r3, $r124") such that **cudasm** can translate. We save these changes into new assembly file, called *volkov_correct.asm*. Finally we would correct these changes in machine code level.

Remark 6: line 1872 and line 1876 relate to instruction "if (beta == 0)" in *method1_variant.cu* because shared memory s[0x0050] is input parameter *beta*.

Step 2: Suppose binary code *from_nvcc.cubin* (Figure 19) is $nvcc(volkov.cu)$ without header and *from_cudasm.cubin* is $cudasm(volkov\_correct.asm)$ without header, then we compare these two binary codes by "diff" command. We test each different binary code segments and find corresponding assembly code in *from_decuda.asm* (Figure 20), then minimum set of error code ψ can be located. The difference has two parts.

Figure 19: green box is header of .cubin file, remove it and store remaining part into from_nvcc.cubin

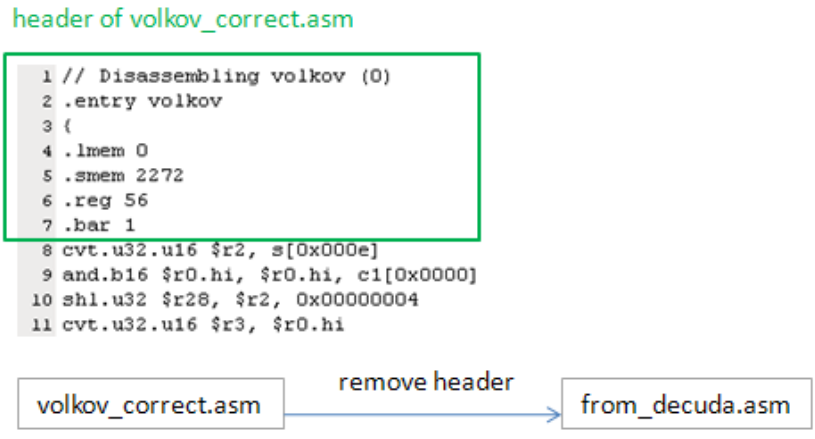


Figure 20: green box is header of .asm file, remove it and store remaining part into from_decuda.asm

Error 1: when load B into shared memory

The **cudasm** could not work for "MOV [smem], reg" which accounts for loading B into shared memory. It seems that

only one/two bits error. For example, "mov.b32 s[$ofs1+0x0000], $r4" corresponds to "0x04000001 0xe4210780" in *from_nvcc.cbin* but **cudasm** translates it to "0x00000001 0xe4210780".

Fortunately such error codes locate in segment Y, they don't affect our pattern of rank-1update.

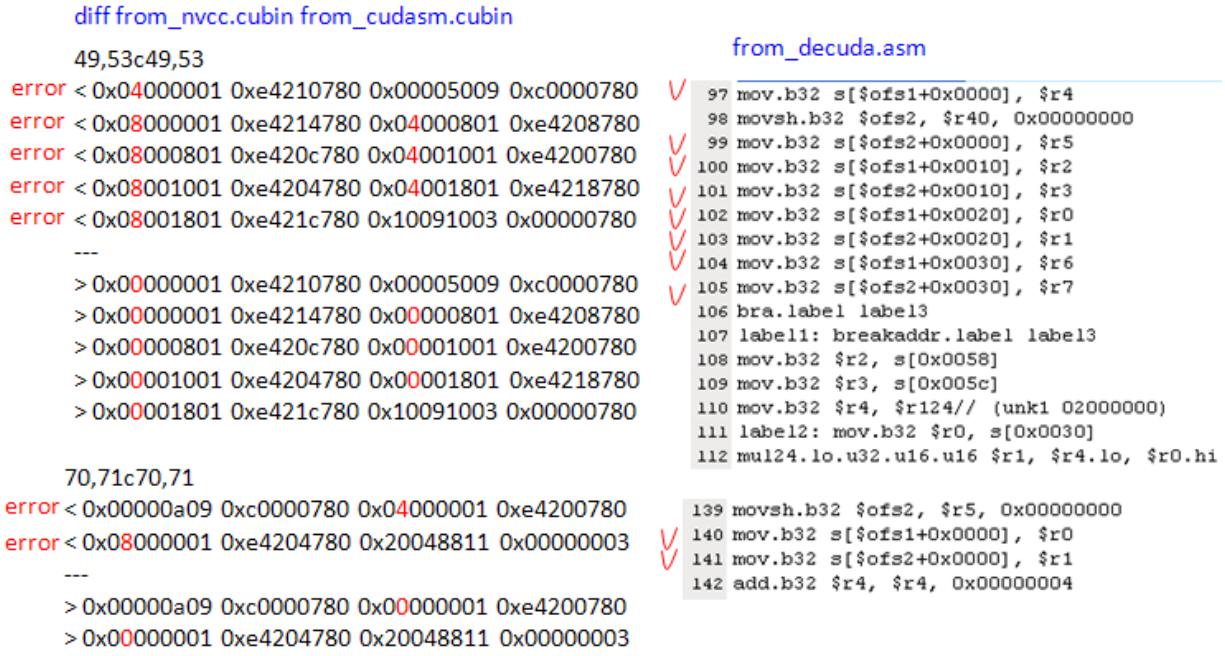


Figure 21: translation error of cudasm on "MOV [smem], reg". from_decuda.asm is volkov_correct.asm without header.

Error 2: correct " mov.b32 $r3, $r124" in step 1

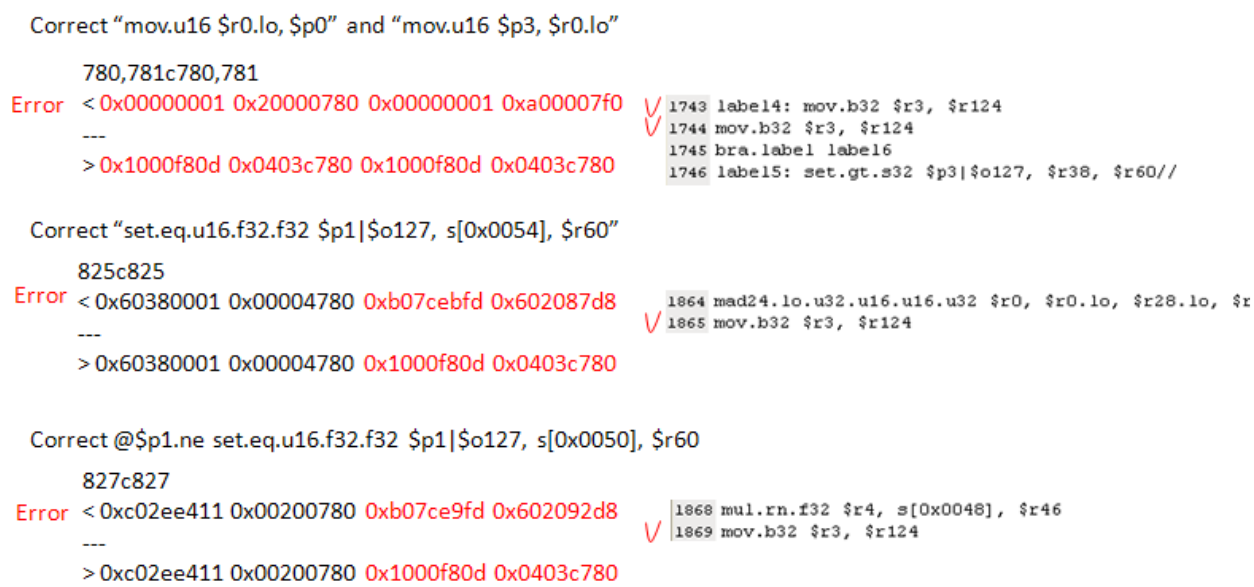


Figure 22: errors due to "mov.b32 $r3, $r124".

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 23 two synchronization commands, __synchthreads(), correspond to assembly code "bar.synch.u32 0x00000000" in line 146 and line 1735 in file *from_decuda.asm* respectively (binary code of __syncthread() is "0x861ffe03 0x00000000"). Clearly rank-1 update lies between two __synchthreads(), so code segment X must be line 147 ~ line 1736 in .asm file. To sum up, the result of decomposition of code segments Y, X, Z is listed in Table 7.

Besides we want to keep register count less than 48 such that number of active threads per SM is 320. This is subtle because **nvcc** uses 56 registers in code segment X but only uses 48 registers in code segments Y, Z. Hence we have no choice but traverse source code to find relationship between register ID and automatic variable in *volkov.cu* file.

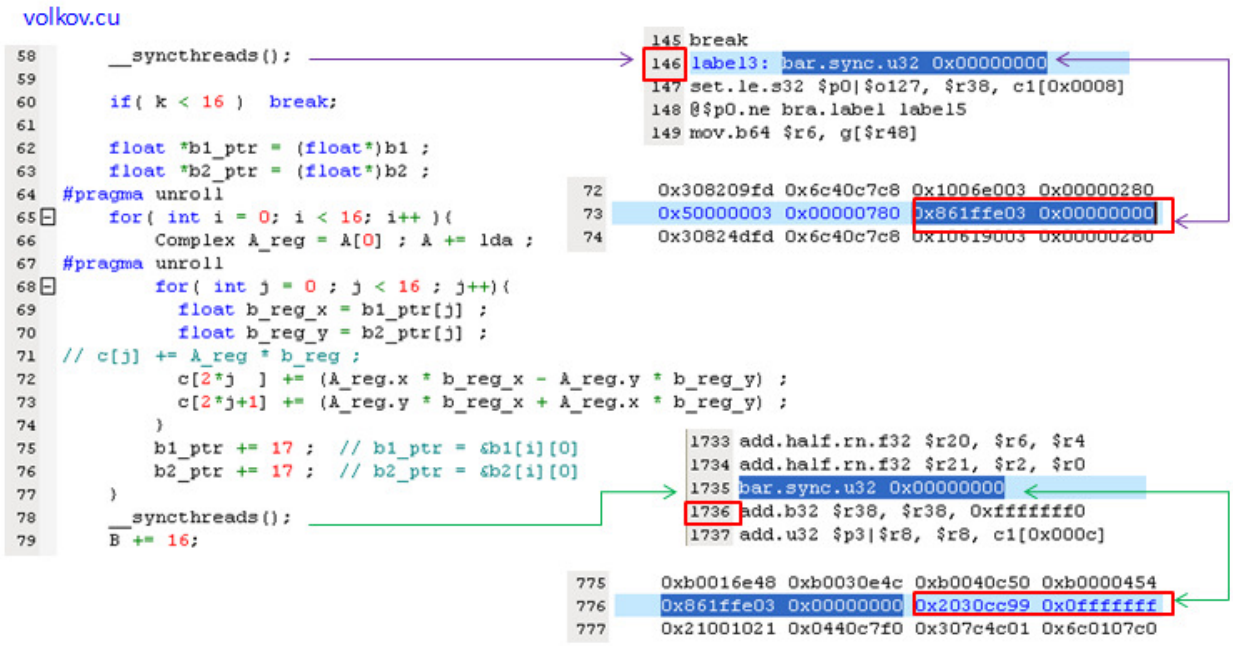


Figure 23: decompose *from_nvcc.cubin* and *from_decuda.asm* into three parts, *X, Y, Z.*

Code segment	from_nvcc.cubin	from_decuda.asm
Y	from_nvcc_1_73.cubin: line 1 ~ 73	from_decuda_1_146.asm: line 1 ~ 146 register count is up to 48
X	from_nvcc_74_776.cubin: line 74 ~ 776	from_decuda_147_1736.asm: line 147 ~ 1736 register count is up to 55
Z	from_nvcc_777_1000.cubin: line 777~1000	from_decuda_1737_2232.asm: line 1737~2232 register count is up to 48

Table 7: decompose binary file *from_nvcc.cubin* and assembly file *from_decuda.asm* into three parts, Y, X, Z according to landmarks in Figure 23 .

Step 4: relationship between register ID and automatic variable in file *volkov.cu*

Table 8 shows relationship between register ID and necessary automatic variables in code segment X. Necessary automatic variables of thread $((\text{threadIdx.x}, \text{threadIdx.y}), (\text{blockIdx.x}, \text{blockIdx.y}))$ are

(1) ***A***: pointer pointing to some element of matrix A,

(2) ***B***: pointer pointing to some element of matrix B,

(3) ***iby*** = blockIdx.y * 16,

(4) ***row*** = ibx + inx + iny*16 is row index,

(5) ***k*** sweeps all column indices of matrix A when do $C_{ij} = \sum_k A_{ik} B_{kj}$,

(6) b1_base = &b1[inx][iny],

(7) b2_base = &b2[inx][iny], and

(8) float c[32] represents sub-matrix of C, $C_{bm,bn}$

All other registers are temporary, ***nvcc*** uses them in loop-unrolling. Before discussing the pattern of rank-1 update, we need to re-bind register r48 (pointer A) to register r45 because threshold of 320 active threads per SM is 48 registers. Such re-binding must be consistent in code segments *Y* and *Z*. Fortunately, register r45 is not used in code segments *Y* and *Z*, we can do such re-binding, see Figure 24. Besides we need three registers to do

Complex A_reg = A[0] ;

A += lda ;

"complex" is 64-bit, two consecutive 32-bit registers represents *A_reg*, say $(r_0, r_1) = A\_reg$. In order to save computation of "lda*sizeof(complex)", we store the value to register r2. Finally relationship between register ID and automatic variables is shown in Table 9.

regID	map	regID	map	regID	Map	regID	map	regID	map
0		10	C[20]	20	C[30]	30	C[12]	40	b2_base
1		11	C[21]	21	C[31]	31	C[11]	41	C[5]
2		12	C[22]	22	C[19]	32	C[10]	42	C[4]
3		13	C[23]	23	C[18]	33	C[9]	43	C[3]
4		14	C[24]	24	C[17]	34	C[8]	44	C[2]
5		15	C[25]	25	C[16]	35	C[7]	45	
6		16	C[26]	26	C[15]	36	C[6]	46	C[1]
7		17	C[27]	27	C[14]	37	row	47	C[0]
8	B	18	C[28]	28	Iby	38	k	48	A
9	?	19	C[29]	29	C[13]	39	b1_base		

Table 8: relationship between register ID and necessary automatic variables in code segment X. Although nvcc uses 56 registers in code segment X, but only necessary variables are up to r48.

Remark 7: register r9 is an exception, we don't know its functionality but it appears in code segment Y and Z.

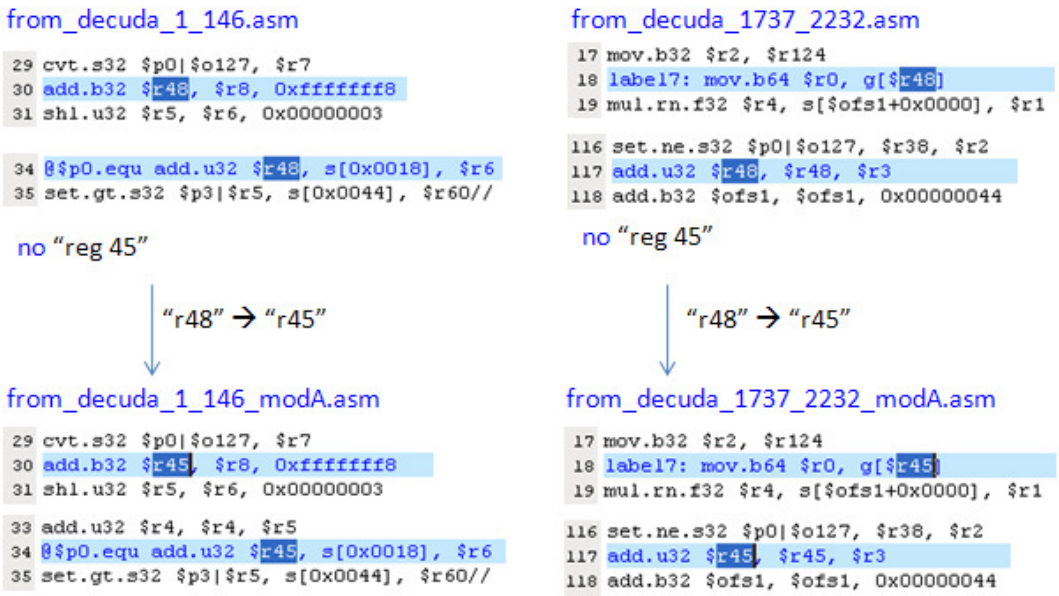


Figure 24: re-bind pointer A from register r48 to register r45 in code segments ***Y*** and ***Z***.

regID	map	regID	map	regID	Map	regID	map	regID	map
0	A_reg.x	10	C[20]	20	C[30]	30	C[12]	40	b2_base
1	A_reg.y	11	C[21]	21	C[31]	31	C[11]	41	C[5]
2	lda*8	12	C[22]	22	C[19]	32	C[10]	42	C[4]
3		13	C[23]	23	C[18]	33	C[9]	43	C[3]
4		14	C[24]	24	C[17]	34	C[8]	44	C[2]
5		15	C[25]	25	C[16]	35	C[7]	45	A
6		16	C[26]	26	C[15]	36	C[6]	46	C[1]
7		17	C[27]	27	C[14]	37	row	47	C[0]
8	B	18	C[28]	28	iby	38	k		
9	?	19	C[29]	29	C[13]	39	b1_base		

Table 9: re-bind pointer A from r48 to r45 and use three registers r0, r1, r2 to represent A_reg and lda*sizeof(complex). Now there are only 48 registers per thread, so number of active threads per SM is 320.

Step 5: design pattern of rank-1 update

Code segment ***X*** (from_decuda_147_1736.asm) contains rank-1 update and four assembly codes, two before rank-1 update and two after rank-1 update, shown in Figure 25. The pattern of rank-1 update is described in Figure 14, here we need to explain two "MOV reg, [smem]",

lds.b32 $r4, s[$ofs2+b2_res]

lds.b32 $r3, s[$ofs1+b1_res]

These two instructions come from C code

*float *b1_ptr = (float*)b1 ;*

*float *b2_ptr = (float*)b2 ;*

....

float b_reg_x = b1_ptr[j] ; // b1_ptr = &b1[i][0]

float b_reg_y = b2_ptr[j] ; // b2_ptr = &b2[i][0]

, i.e. $r_3 \equiv b\_reg\_x$ and $r_4 \equiv b\_reg\_y$. First one must know that __shared__ float b1[16][17] starts at 0x60 and __shared__ float b2[16][17] starts at 0x4a0, then

$$\&b1[i][j] = 0x60 + (17i + j) \times sizeof\,(float)$$

$$\&b2[i][j] = 0x4a0 + (17i + j) \times sizeof\,(float)$$

We use two special registers, $\$ofs1$ and $\$ofs2$[3], to encode LOAD instruction, "MOV reg, [smem]". Decompose address $\&b1[i][j]$ and $\&b2[i][j]$ by $\&b1[i][j] = \$ofs1 + b1\_res$ and $\&b2[i][j] = \$ofs2 + b2\_res$ respectively by $b1\_res \equiv \&b1[i][j] \mod 0x80$ and $b2\_res \equiv \&b2[i][j] \mod 0x80$. Then "*float b_reg_x = b1_ptr[j] ;*" is translated into "*lds.b32 $r3, s[$ofs1+b1_res]*" and "*float b_reg_y = b2_ptr[j] ;*" is translated into "*lds.b32 $r4, s[$ofs2+b2_res]*".

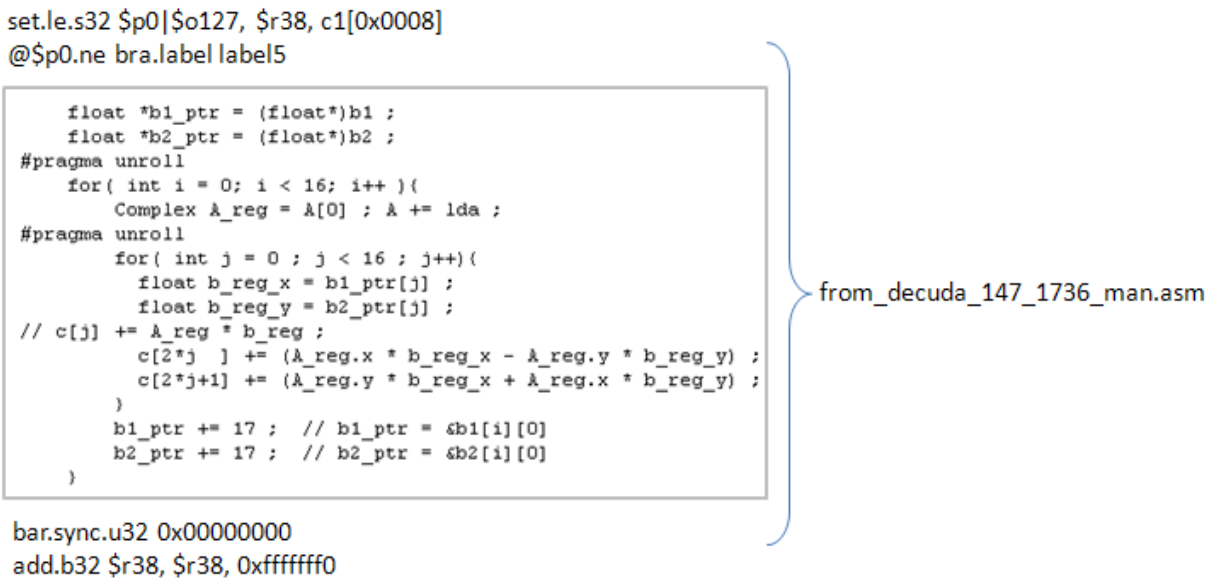


Figure 25: design pattern of rank-1 update, which is enclosed by 4 assembly codes.

Remark 8: Such pattern can be generated by function *rank1_update_method1()* in /method1/rank1_update_method1.cpp

Step 6: merge code segment ***X*** containing assembly code of desired pattern with code segments *Y* and *Z* to a file,

[3] Addressing memory is quite interesting. It appears that there are four offset registers (on current hardware) $ofs1, $ofs2, $ofs3 and $ofs4. These are used to add an offset to the immediate offsets encoded in the instructions. http://github.com/laanwj/decuda/raw/master/README

from_decuda_ldsb32.asm, see Figure 26. Also 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

and remove header of " *from_decuda_ldsb32_cudasm.cubin* " to be "*from_decuda_ldsb32_cudasm_noHeader.cubin*".

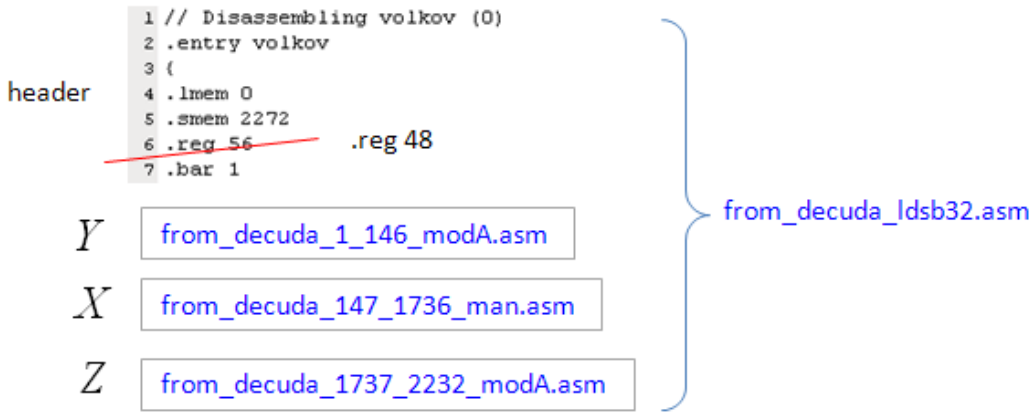


Figure 26: combine header and code segments *Y, X, Z* into a new assembly file, which translate rank-1 update as desired pattern.

Step 7: correct error codes in step 2 (see Figure 21 and Figure 22)

Step 8: merge header of "*volkov.cubin"* and "*from_decuda_ldsb32_cudasm_noHeader.cubin*" to final executable binary file "*method1.cubin*" (Figure 27). "*method1.cubin*" can be loaded into application via driver API.

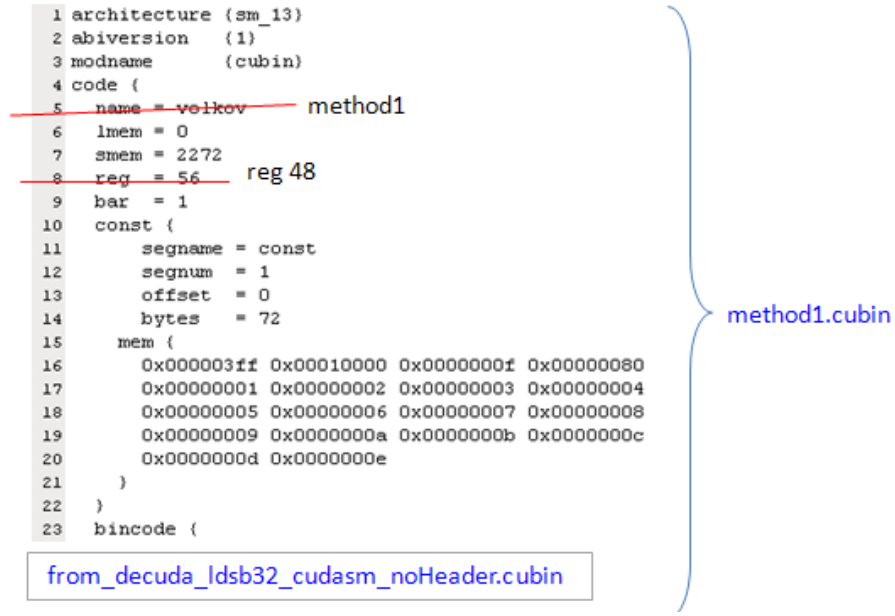


Figure 27: combine header of "*volkov.cubin"* and final executable binary code "*from_decuda_ldsb32_cudasm_noHeader.cubin*" into method1.cubin

5 Idea 2: remove RAW hazard to increase possibility of dual issue (method 2 variant)

We only use one MUL per "$c = a*b + c$" (original Volkov's code use 2 MUL per "$c = a*b + c$") in method 2 and its variant, see Figure 28. The main difference between method 2 and method2_variant is RAW (Read-After-Write) hazard, proper rearrangement of instructions in method2_varinat removes RAW hazard but theoretical peak performance of method2_variant is smaller than that of method 2.

Theoretical peak performance of method 2 with dual issue is

$$\frac{4MAD}{load + ADD + 2T_{MAD,reg} + T_{MAD,smem}} = \frac{4}{5.5}\frac{MAD}{T_{MAD,reg}} = 453.82Gflop/s$$

Theoretical peak performance of method 2 without dual issue is

$$\frac{4MAD}{load + ADD + 2T_{MAD,reg} + T_{MAD,smem} + T_{MUL,smem}} = \frac{4}{7}\frac{MAD}{T_{MAD,reg}} = 356.57Gflop/s$$

Theoretical peak performance of method 2 variant with dual issue is

$$\frac{4MAD}{load + ADD + 3T_{MAD,reg}} = \frac{4}{6}\frac{MAD}{T_{MAD,reg}} = 416Gflop/s$$

Theoretical peak performance of method 2 variant without dual issue is

$$\frac{4MAD}{load + ADD + 3T_{MAD,reg} + T_{MUL,reg}} = \frac{4}{7}\frac{MAD}{T_{MAD,reg}} = 356.57Gflop/s$$

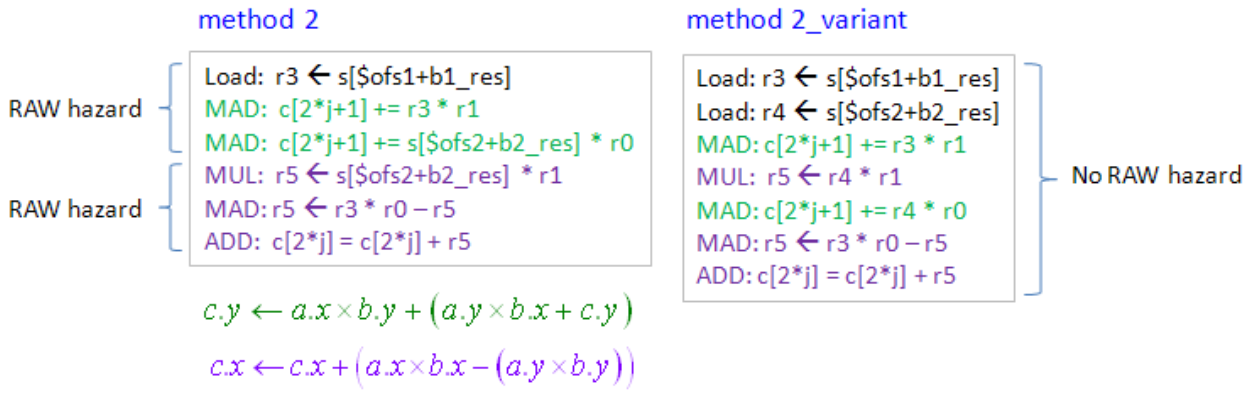


Figure 28: pattern of method 2 and its variant. The difference is RAW (Read-After-Write) hazard.

Performance of method 2 and its variant are shown in Table 10 and Figure 29. Clearly, performance of method 2 is near peak performance without dual issue. That means that RAW hazard of method 2 stops dual issue. However performance of method 2 variant is near peak performance with dual issue.

N	Volkov (Gflop/s)	method 2 (Gflop/s)	R (method2/volkov)	nethod2 variant (Gflop/s)	R (method2 variant/volkov)
256	187.892	169.412	1.1091	179.885	1.0445
512	265.591	310.015	0.8567	326.418	0.8137
1024	329.555	349.521	0.9429	369.045	0.8930

2048	342.854	362.175	0.9467	384.923	0.8907
4096	348.489	369.914	0.9421	394.145	0.8842

Table 10: performance of method 2 and its variant on TeslaC1060.

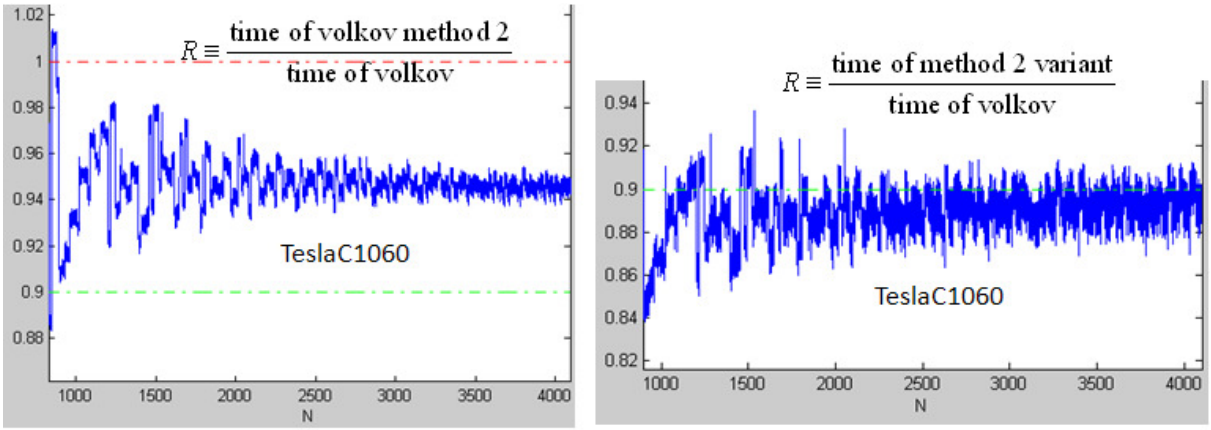


Figure 29: left panel: performance of method 2on TeslaC1060.

Right panel: performance of method2_variant.

6 Idea 3: mix idea 1 and idea 2 to increase possibility of dual issue without RAW hazard

6.1 method 3: mix "volkov + unroll 1" and "method 1"

We have shown bad performance (348Gflop/s) of "volkov + unroll 1" in section 2.4 and conjecture that RAW hazard destroys dual issue. On the other hand, method 1 outperforms (445.7Gflop/s) due to amazing pattern. We may ask one question, could we combine "volkov + unroll 1" and "method 1" to remove RAW hazard and keep advantage of amazing pattern? The answer is method 3 in Figure 30.

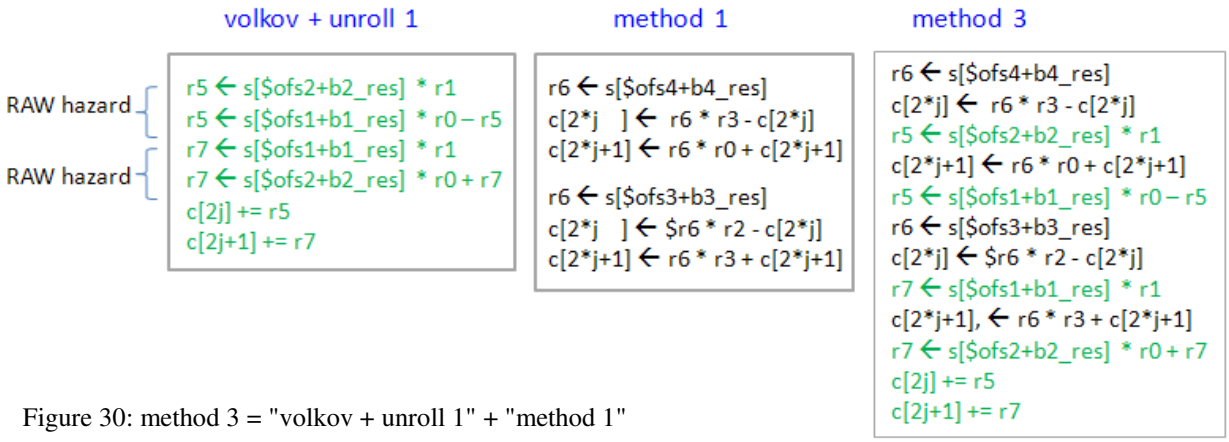


Figure 30: method 3 = "volkov + unroll 1" + "method 1"

In order to interleave "volkov + unroll 1" and "method 1", we need more registers, including

$(r_0, r_1) \equiv A[0]$ is used in "volkov + unroll 1",

$(r_2, r_3) \equiv A[lda]$ is used in "method 1", and

$r_4 \equiv lda \times sizeof\left(complex\right)$.

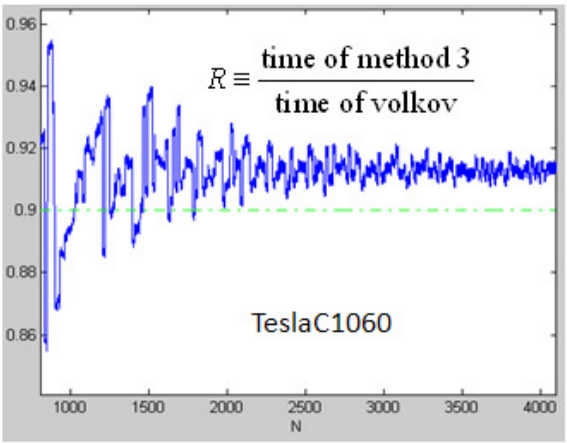
Obviously, peak performance of method 3 is between "volkov + unroll 1" and "method 1".

peak performance of "volkov + unroll 1" with dual issue is $\frac{4}{5}\frac{MAD}{T_{MAD,reg}} = 499.2Gflop/s$ and

peak performance of "method 1" is $\frac{4}{6}\frac{MAD}{T_{MAD,reg}} = 416Gflop/s$, then

peak performance of "method 3" with dual issue is $\frac{4+4}{5+6}\frac{MAD}{T_{MAD,reg}} = 453.82Gflop/s$

However method 3 only reaches 382 Gflop/s from Figure 31, and 382 Gflop/s is smaller than 396.85 Gflop/s which is average of "volkov + unroll 1" and "method 1" ($\frac{348+445.7}{2} = 396.85Gflop/s$). This is unreasonable because RAW hazard is removed.



N	Volkov (Gflop/s)	Method 3 (Gflop/s)	R
256	187.892	192.465	0.9762
512	265.591	331.327	0.8016
1024	329.555	364.910	0.9031
2048	342.854	376.599	0.9104
4096	348.489	381.920	0.9125

Figure 31: performance of method 3 on TeslaC1060.

6.2 method 4: mix "method 2" and "method 1"

Similarly "method 2" has bad performance 369.9 Gflop/s due to RAW hazard. We interleave "method 2" and "method 1" in Figure 32. Also peak performance of method 4 is average of "method 2" and "method 1".

peak performance of "method 2" with dual issue is $\frac{4}{5.5}\frac{MAD}{T_{MAD,reg}} = 453.82Gflop/s$ and

peak performance of "method 1" is $\frac{4}{6}\frac{MAD}{T_{MAD,reg}} = 416Gflop/s$, then

peak performance of "method 4" with dual issue is $\frac{4+4}{5.5+6}\frac{MAD}{T_{MAD,reg}} = 443.08Gflop/s$

In Figure 33, method 4 only reaches 407 Gflop/s, which is exact average of "method 2" and "method 1".

T sum up, method 3 and method 4 do not achieve our goal, we have no idea about this phenomenon.

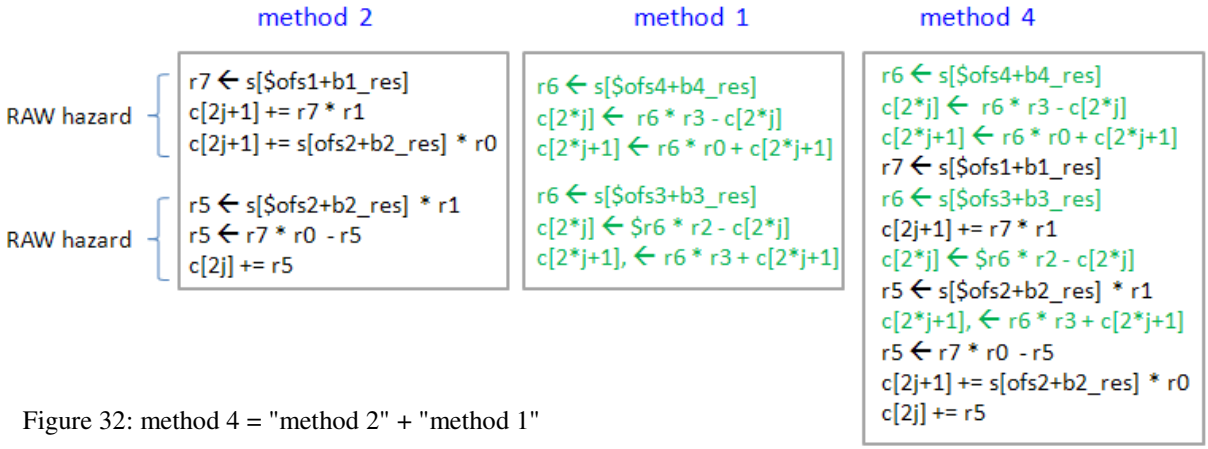
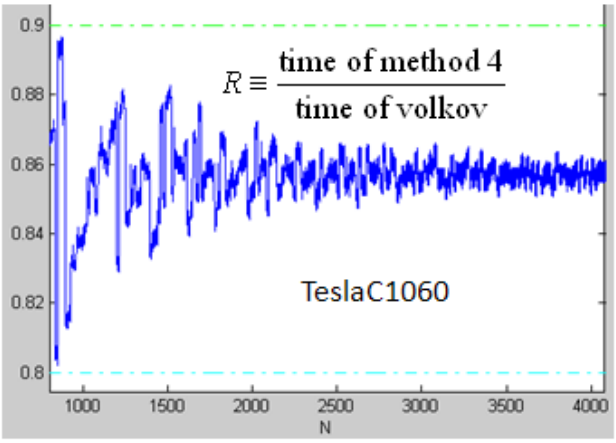


Figure 32: method 4 = "method 2" + "method 1"



N	Volkov (Gflop/s)	Method 4 (Gflop/s)	R
256	187.892	200.685	0.9757
512	265.591	351.212	0.8053
1024	329.555	387.220	0.9095
2048	342.854	401.559	0.9166
4096	348.489	407.128	0.9193

Figure 33: performance of method 4 on TeslaC1060.

7 conclusions

In this work, we use package **decuda/cudasm** to design pattern of rank-1 update in CGEMM under Volkov's work. Experimental result, method 1, supports dual issue of amazing pattern found in [3]. However we still cannot explain why method 3 and method 4 does not achieve expected performance.

Resource usage and computational cost of these five methods are listed in Table 11. Method 1 reaches 445.724 Gflop/s, slightly better than method 1 in [3], which attains 439467 Gflop/s. However we believe that CGEMM should have potential to reach 499 Gflop/s if we can solve RAW hazard.

algorithm	volkov	method 1	method 2 (variant)	method 3	method 4
Active threads per SM	256	320	320	320	320
Peak performance (Gflop/s) on TeslaC1060	499.2	416	416	453.82	434.08

Experimental Gflop/s on TeslaC1060	348.489	445.724	394.145	381.920	407
ranking	4	1	2	3	2

Table 11: resource usage and computational cost among five algorithms, volkov, method 1, method2_variant, method 3 and method 4.

Figure 34 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 1.155x than that of TeslaC1060. Figure 34 is almost the same as Figure 34 in [3]. Of course if we compare method 1 with CUBALS, then we have 37.69% improvement because CUBALS only reaches 277.7Gflop/s.

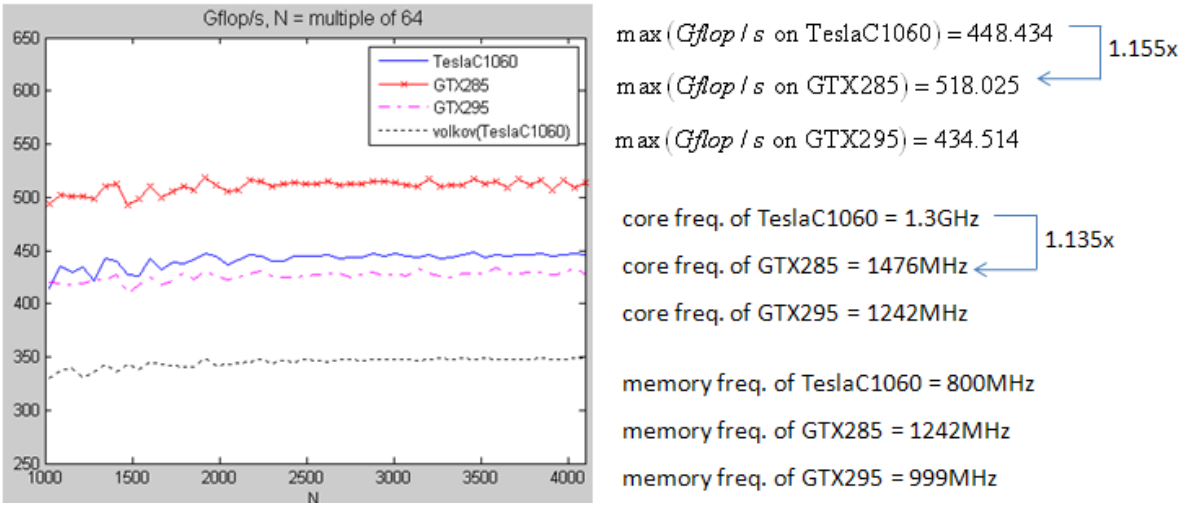


Figure 34: performance of method 1 over N = multiple of 64 on TeslaC1060, GTX285 and GTX295. The baseline is performance of Volkov's code on TeslaC1060.

In this work, we rely on effectiveness of package **decuda/cudasm** as we explain in section 4. Although we do many manual modification s in this work, it is more easy than what we do in SGEMM [3]. To sum up, I think there is a lot of degree of freedom on optimization if we take code pattern into account, including RAW elimination, combination of MUL and MAD, amazing pattern, ... etc.

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] Lung-Sheng Chien, Hand-Tuned SGEMM on GT200 GPU,

http://forums.nvidia.com/index.php?showtopic=159033

[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] thread on the forum discussing dual issue,

http://forums.nvidia.com/index.php?showtopic=103046&pid=570411&mode=threaded&show=&st=#entry570411

[7] David Kanter, instruction issue logic,

http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=8

[8] David Kanter , dual issue, http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=9

[9] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Henry Wong, Micro-benchmarking the GT200 GPU,

http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/microbenchmark_report.pdf