Author: Gwangho Kim
Date: Aug. 12, 2019

Comments on
http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda

Refer to the example in “3.2.3. Shared Memory” from
CUDA C Programming Guide
V10.1, May 2019

Set *tiled* submatrices

Build tiled submatrices As, Bs from matrices A, B, respectively.
Each thread in thread block copies one element from A for As, and one element from B for Bs.

Read matrices

There are two ways to fill submatrices. For each thread indexed by i, j,
Same order in matrix as
M[i][j]
Opposite order as
M[j][i]

This step is to *set* submatrices, so the order *does* not matter; there might be another way. The key is that which way is the *fastest* way to build submatrices.
5.3.2. Device Memory Accesses
Appendix H. Compute Capabilities

Build submatrices

Also, there are two ways to build submatrices.
Copy
Ms[i][j] = M[i][j]
Transpose
Ms[i][j] = M[j][i]

Set submatrices

There are four ways to set submatrices.

Way 1. Read M in the same order and copy it

M[i][j]
Ms[i][j] = M[i][j]

M:

0	1	2
3	4	5
6	7	8

Ms:[1]

0_0	1_1	2_2
3_3	4_4	5_5
6_6	7_7	8_8

Way 2. Read M in the same order but transpose it

M[i][j]
Ms[i][j] = M[j][i]

M:

0	1	2
3	4	5
6	7	8

Ms:

0_0	3_1	6_2
1_3	4_4	7_5
2_6	5_7	8_8

Way 3. Read M in the opposite order and copy it

M[j][i]
Ms[i][j] = M[i][j]

M:

0	1	2

[1] A subscript is the reading order.

3	4	5
6	7	8

Ms:

0_0	1_3	2_6
3_1	4_4	5_7
6_2	7_5	8_8

Way 4. Read M in the opposite order and transpose it

M[j][i]
Ms[i][j] = M[j][i]

M:

0	1	2
3	4	5
6	7	8

Ms:

0_0	3_3	6_6
1_1	4_4	7_7
2_2	5_5	8_8

Examples

There are three types of kernels in the above link:

matrixMul_tiling

```
BS(tx, ty) = B[b + wB * tx + ty]; // BS[tx][ty]; build BS in opposite order, and copy it
...
Csub += AS(ty, k) * BS(k, tx);
```

This is “Way 3”.

matrixMul_coalescing

```
BS(tx, ty) = B[b + wB * ty + tx]; // BS[tx][ty]; build BS in same order, but transpose it
…
Csub += AS(ty, k) * BS(tx, k);
```

This is “Way 2”.

matrixMul_noBankConflict

```
BS(ty, tx) = B[b + wB * ty + tx]; // BS[ty][tx]; build BS in same order, and copy it
…
Csub += AS(ty, k) * BS(k, tx);
```

This is “Way 1”.

Another one

```
BS(ty, tx) = B[b + wB * tx + ty]; // BS[ty][tx]; build BS in opposite order, but transpose it
...
Csub += AS(ty, k) * BS(tx, k);
```

This is “Way 4”.

Example in 3.2.3. Shared Memory

```
Bs[row][col] = GetElement(Bsub, row, col); // build Bs in same order, and copy it
…
Cvalus += As[row][e] * Bs[e][col];
```

This is “Way 1”.

Performance in GFLOPS

With “GeForce GTX 960M” with compute compatibility 5.0

Way	GLOPS
1	110
2	53
3	98
4	56

It says

    1 > 3 >> 2 > 4

In other words,

    Copy >> Transpose (major)
    Same > Opposite (minor)

Why?

H.4.3. Shared Memory (for compute compatibility 5.x) says

> *Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.*

> *A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank): In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).*

C = A B

C[i][j] = ∑ A[i][k] * B[k][j]
A[BLOCK][BLOCK]
B[BLOCK][BLOCK]

Each thread starts summations as follows:

    Thread[0][0]
        A[0][0] * B[0][0] + A[0][1] * B[1][0] + …

    Thread[0][1]:
        A[0][0] * B[0][1] + A[0][1] * B[1][1] + …

    ……
    Thread[0][15]:
        A[0][0] * B[0][15] + A[0][1] * B[1][15] + …

At given time, up to 16 threads,
    There is only one read for A[0][0], and
    B[0][0], B[0][1], …, B[0][15]
Because all are read from different banks, there is no bank conflict for B except A[0][0].
    There is two-way bank conflict only for bank=0.

    In fact, A[0][0] is broadcast to all threads.

However, there are 32 threads in a warp.

    Thread[1][0]:
        A[1][0] * B[0][0] + A[1][1] * B[1][0] + …
    Thread[1][1]:
        A[1][0] * B[0][1] + A[1][1] * B[1][1] + …
    ……
    Thread[1][15]
        A[1][0] * B[0][15] + A[1][1] * B[1][15] + …

For A, there are multicast. For B, there are two-way bank conflicts.

What if BLOCK=32?

    Thread[0][16]:
        A[0][0] * B[0][16] + A[0][1] * B[1][16] + …
    Thread[0][17]:
        A[0][0] * B[0][17] + A[0][1] * B[1][17] + …
    ……
    Thread[0][31]:
        A[0][0] * B[0][31] + A[0][1] * B[1][31] + …

For A, there is a broadcast while all B are bank conflict free.

C = A B^T

C[i][j] = ∑ A[i][k] * B[k][j] = ∑ A[i][k] * B^T[j][k]

    Thread[0][0]
        A[0][0] * B[0][0] + A[0][1] * B[0][1] + …

    Thread[0][1]:
        A[0][0] * B[1][0] + A[0][1] * B[1][1] + …

    ……
    Thread[0][15]:
        A[0][0] * B[15][0] + A[0][1] * B[15][1] + …

At given time,

    A[0][0]: broadcast
    B[0][0], B[1][0], …, B[15][0]: all are in the same bank; 16-way bank conflict

Inconsistency in results

The link says they got the performance at this order:

    1 >> 2 > 3