Korea Advanced Institute of Science and Technology

School of Electrical Engineering

EE817 GPU Programming and Its Applications Spring 2018

Student Name: Dinh Vu

Student ID: 20184187

Homework 3

The computer used in this homework contains NVIDIA GeForce 1070 based on Pascal GP104 architecture.

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvidia-smi
Wed Apr 11 01:20:22 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.48                 Driver Version: 390.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1070    Off  | 00000000:01:00.0 Off |                  N/A |
|  8%   49C    P5    13W / 200W |      0MiB /  8119MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1070    Off  | 00000000:02:00.0 Off |                  N/A |
|  0%   50C    P5    24W / 200W |      0MiB /  8119MiB |      2%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 1. Graphic card information

1. Problem 1

Suppose you have a shared memory tile with dimension [32][32]. Pad a column to it and then draw an illustration showing the mapping between data elements and banks for a Kepler device in 8-byte access mode.

Assume each element of the shared memory tile is a 4-byte word and have index as Figure 1.1.

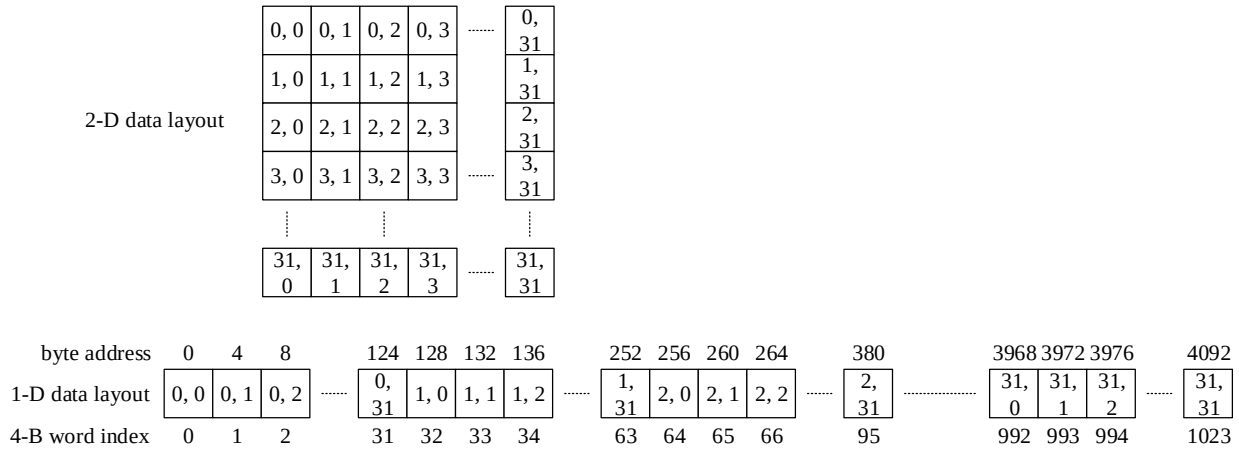

Figure 1.1. 2-D and 1-D data layout of the shared memory tile

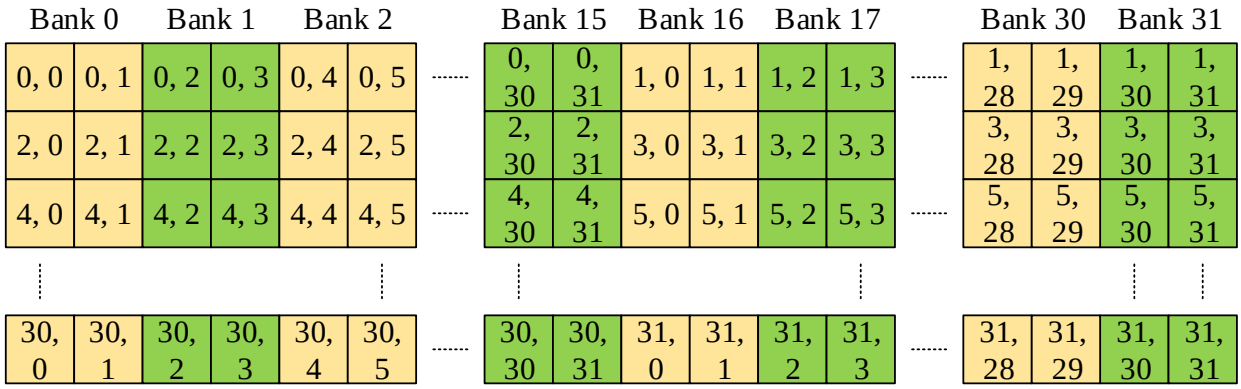

Figure 1.2. Bank mapping before padding

From byte address in Figure 1.1, before padding each element is mapped into bank with 8-byte access mode as shown in Figure 1.2. In order to avoid bank conflict, two 4-byte word is added at the end of each row of the shared memory tile, see Figure 1.3. After padding, each 4-byte word is relocated by Figure 1.4.

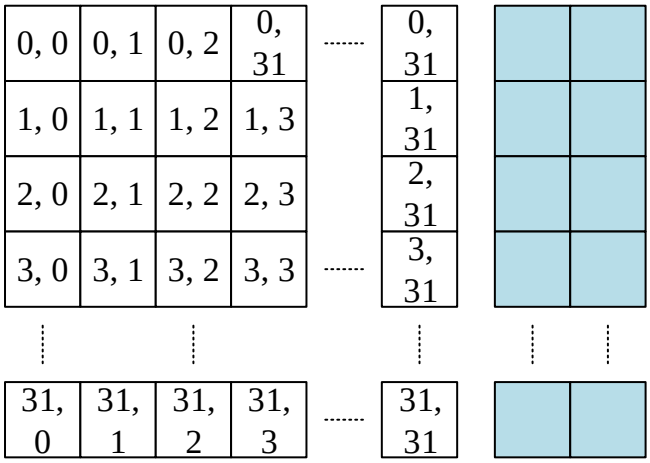

Figure 1.3. Shared memory padding with 2 columns

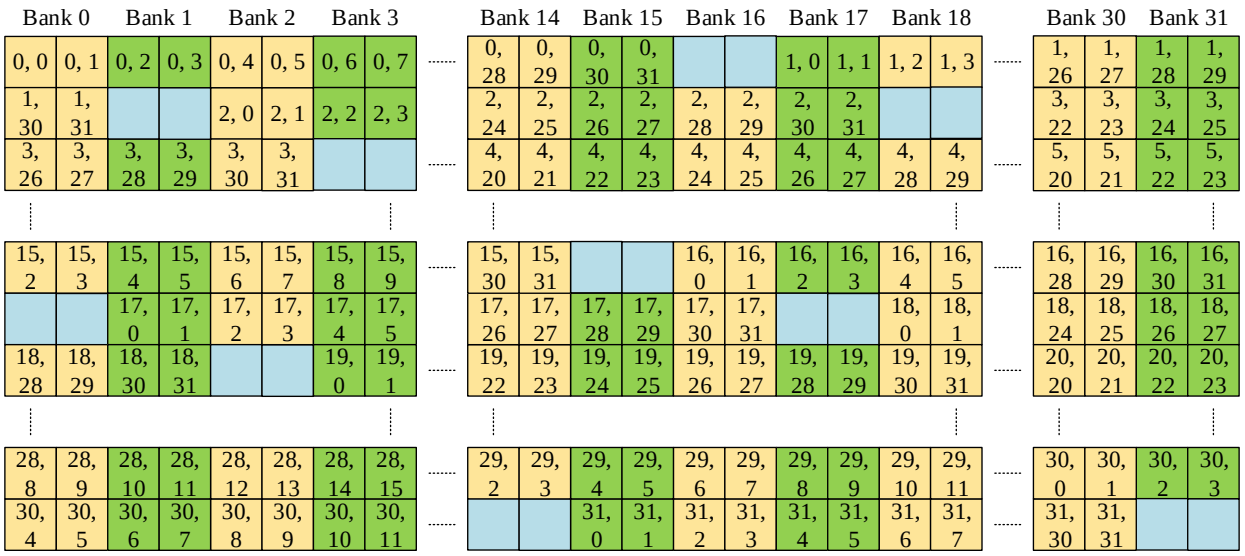

Figure 1.4. Shared memory mapping into bank after padding

The bank mapping rule is presented as follow:

Let row[idx] is the row having index is idx.

- If idx is even and $0 \leq idx \leq 14$, after saving row[idx], bank[idx + 16] is padd. Example: after storing row[0], bank[16] is padd and after row[2], bank[2] is padd.
- If idx is even and $16 \leq idx \leq 30$, after row[idx] is stored, bank[idx – 16] is padd. Example: after row[16] is stored, bank[0] is padd and after row[18] is stored, bank[2] is padd.
- If idx is odd then after relocating row[idx], bank[idx] is padd, such as after row 1, bank 1 is padded then after row 3, bank 3 is padded and so on.

2. Problem 2

The source code for problem 2 is checkSmemSquare.cu. In the line 154, the function cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte) is used to set 8-byte address access mode. The results is shown in Figure 2.

In the kernel setRowReadColDynPad() and setRowReadColPad(), there is one bank conflict in both write (load) and read (store). Because both of kernels write by row-order and use reading by column-order.

Two kernel setRowReadCol() and setRowReadCol() have the same efficiency. Bank conflict occurs one time in writing (store) due to write by row-order. However, there are 32 bank conflicts in reading (load) because of reading by column order.

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof --metrics shared_load_transactions_per_request,shared_store_transactions_per_request ./prob2
==19424== NVPROF is profiling process 19424, command: ./prob2
./prob2 at device 0: GeForce GTX 1070 with Bank Mode:8-Byte <<< grid (1,1) block (32,32)>>>
==19424== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==19424== Replaying kernel "setColReadCol(int*)" (done)
==19424== Replaying kernel "setRowReadRow(int*)" (done)
==19424== Replaying kernel "setRowReadCol(int*)" (done)
==19424== Replaying kernel "setRowReadColDyn(int*)" (done)
==19424== Replaying kernel "setRowReadColPad(int*)" (done)
==19424== Replaying kernel "setRowReadColDynPad(int*)" (done)
==19424== Profiling application: ./prob2
==19424== Profiling result:
==19424== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "GeForce GTX 1070 (0)"
    Kernel: setRowReadColDynPad(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.000000    1.000000    1.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: setRowReadCol(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request   32.000000   32.000000   32.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: setRowReadColDyn(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request   32.000000   32.000000   32.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: setRowReadRow(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    0.000000    0.000000    0.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: setColReadCol(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    0.000000    0.000000    0.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request   32.000000   32.000000   32.000000
    Kernel: setRowReadColPad(int*)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.000000    1.000000    1.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
```

Figure 2. The efficiency of shared memory in 8-byte mode

In both of kernel setRowReadRow() and setColReadCol(), bank conflict does not occur in writing (load) and there is only one time bank conflict because both of kernel using the same major-order for both reading and writing, row-major and column-major respectively.

3. Problem 3

3.1. Problem 3.1

The source code for problem 3.1 is matrixMulGmem.cu.

Each thread computes value of each element in matrix C.

Row of matrix A and the corresponding column of matrix B is read directly from global memory (line 44, 45 in the source code).

Then the value of each element in matrix with thread index is calculated parallel. Figure 3 below shows the programming strategy using only global memory

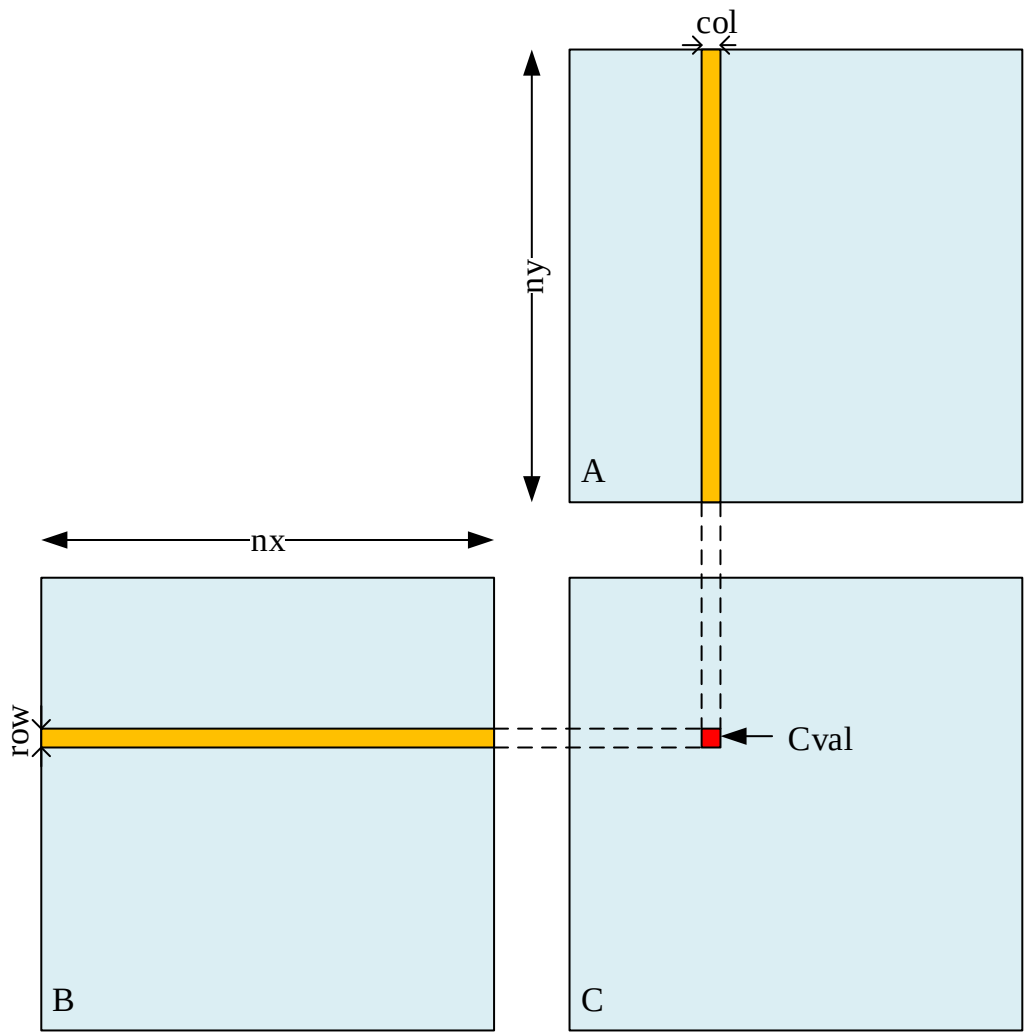

Figure 3. Matrix multiplication only using global memory

The execution time, the efficiency of global memory and the number of bank conflicts in shared memory are presented in Figure 4 and Figure 3.

The execution time of kernel matrixMulGmem() equals to 58.892 ms.

Because of without using shared memory, shared_store_transactions_per_request and shared_load_transactions_per_request equal to zero.

Writing to global memory (gst_efficiency) achieves fully efficiency 100 % due to mapping thread index to global address from line 35 to 37 in the source code file.

The efficiency of reading is not 100 % and about 82.5 % because of reading element of matrix B is not coalesced (see line 45 in the source code file).

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvcc -arch=sm_61 -o prob3_1 matrixMulGmem.cu
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof ./prob3_1
==19797== NVPROF is profiling process 19797, command: ./prob3_1
Matrix Multiplication OK!
==19797== Profiling application: ./prob3_1
==19797== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 78.09%  58.892ms         1  58.892ms  58.892ms  58.892ms  matrixMulGmem(int*, int*, int*, int)
 17.47%  13.176ms         1  13.176ms  13.176ms  13.176ms  [CUDA memcpy DtoH]
  4.44%  3.3448ms         2  1.6724ms  1.6553ms  1.6895ms  [CUDA memcpy HtoD]

==19797== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.86%  247.97ms         3  82.656ms  135.43us  247.69ms  cudaMalloc
 24.23%  80.249ms         3  26.750ms  1.7446ms  72.658ms  cudaMemcpy
  0.59%  1.9383ms         3  646.11us  216.03us  863.36us  cudaFree
  0.23%  754.67us       182  4.1460us     158ns  176.11us  cuDeviceGetAttribute
  0.07%  218.43us         2  109.21us  106.04us  112.39us  cuDeviceTotalMem
  0.02%  77.198us         2  38.599us  32.645us  44.553us  cuDeviceGetName
  0.01%  24.295us         1  24.295us  24.295us  24.295us  cudaLaunch
  0.00%  3.6330us         4     908ns     203ns  2.7940us  cudaSetupArgument
  0.00%  1.8110us         3     603ns     201ns  1.3330us  cuDeviceGetCount
  0.00%  1.7880us         6     298ns     177ns     488ns  cuDeviceGet
  0.00%  1.0910us         1  1.0910us  1.0910us  1.0910us  cudaConfigureCall
```

Figure 4. The execution time of matrix multiplication without shared memory

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvcc -arch=sm_61 -o prob3_1 matrixMulGmem.cu
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof --metrics shared_load_transactions_per_request,shared_store_transactions_per_request,gst_effici
ency,gld_efficiency ./prob3_1
==19688== NVPROF is profiling process 19688, command: ./prob3_1
==19688== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==19688== Replaying kernel "matrixMulGmem(int*, int*, int*, int)" (done)
Matrix Multiplication OK!
==19688== Profiling application: ./prob3_1
==19688== Profiling result:
==19688== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1070 (0)"
    Kernel: matrixMulGmem(int*, int*, int*, int)
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    0.000000    0.000000    0.000000
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    0.000000    0.000000    0.000000
          1                          gst_efficiency               Global Memory Store Efficiency     100.00%     100.00%     100.00%
          1                          gld_efficiency                Global Memory Load Efficiency      82.50%      82.50%      82.50%
```

Figure 5. The efficiency of global and shared memory in matrix multiplication without shared memory

3.2. Problem 3.2

The source code for problem 3.1 is matrixMulGSmem.cu. The method using shared memory tile is displayed in Figure 6. Each thread computes an element of matrix C. The chosen size of shared memory tile is TILE_WIDTH = 32.

By row-major order, row of matrix A will be load to tile ds_A, the corresponding column of matrix B will be load to tile ds_B (line 47-51 in the source code).

The value of each element in matrix C (Cval) equal to sum of product of each element in matrix A and B (line 55 in the source code).

Then this value Cval is load to global memory by locating the address from row and column of matrix (line 60 in the source code)

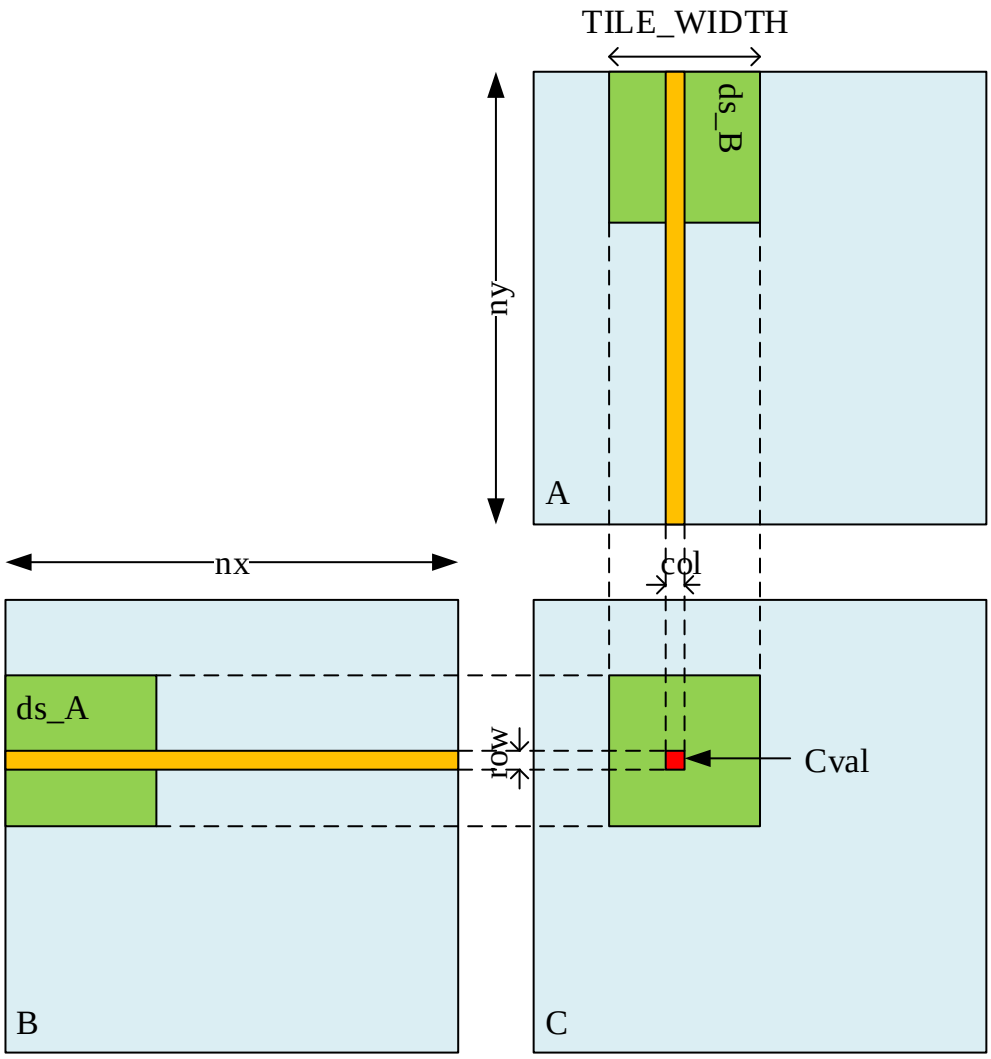

Figure 6. Using shared memory tile

The execution time, the efficiency of global memory and the number of bank conflicts in shared memory are presented in Figure 7 and Figure 8.

The execution time of matrix multiplication using shared memory without padding is 12.194 ms, equal quarter of without using shared memory.

The efficiency of global memory peaks 100 % in both writing and reading because of coalesce and align.

There is a bank conflict in writing shared memory. In the reading shared memory, the bank conflict is 1.2 because of the optimization variables in CUDA compiler.

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvcc -arch=sm_61 -o prob3_2 matrixMulGSmem.cu
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof ./prob3_2
==27106== NVPROF is profiling process 27106, command: ./prob3_2
Matrix Multiplication OK!
==27106== Profiling application: ./prob3_2
==27106== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 67.65%  12.194ms         1  12.194ms  12.194ms  12.194ms  matrixMulGSmem(int*, int*, int*, int)
 18.96%  3.4174ms         2  1.7087ms  1.6939ms  1.7235ms  [CUDA memcpy HtoD]
 13.39%  2.4127ms         1  2.4127ms  2.4127ms  2.4127ms  [CUDA memcpy DtoH]

==27106== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 93.68%  319.14ms         3  106.38ms  135.58us  318.87ms  cudaMalloc
  5.41%  18.418ms         3  6.1393ms  1.6672ms  14.839ms  cudaMemcpy
  0.57%  1.9493ms         3  649.78us  227.45us  863.26us  cudaFree
  0.24%  801.58us       182  4.4040us     171ns  177.87us  cuDeviceGetAttribute
  0.07%  226.45us         2  113.22us  113.17us  113.28us  cuDeviceTotalMem
  0.03%  95.835us         2  47.917us  38.058us  57.777us  cuDeviceGetName
  0.01%  26.935us         1  26.935us  26.935us  26.935us  cudaLaunch
  0.00%  2.9820us         4     745ns     215ns  2.1150us  cudaSetupArgument
  0.00%  2.4950us         6     415ns     193ns     973ns  cuDeviceGet
  0.00%  1.7460us         3     582ns     269ns  1.1760us  cuDeviceGetCount
  0.00%  1.2160us         1  1.2160us  1.2160us  1.2160us  cudaConfigureCall
```

Figure 7. The execution time of matrix multiplication using global and shared memory without padding

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof --metrics gst_efficiency,gld_efficiency,shared_store_transactions_per_request,shared_load_trans
actions_per_request ./prob3_2
==27190== NVPROF is profiling process 27190, command: ./prob3_2
==27190== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==27190== Replaying kernel "matrixMulGSmem(int*, int*, int*, int)" (done)
Matrix Multiplication OK!
==27190== Profiling application: ./prob3_2
==27190== Profiling result:
==27190== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1070 (0)"
    Kernel: matrixMulGSmem(int*, int*, int*, int)
          1                            gst_efficiency               Global Memory Store Efficiency     100.00%     100.00%     100.00%
          1                            gld_efficiency                Global Memory Load Efficiency     100.00%     100.00%     100.00%
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.200000    1.200000    1.200000
```

Figure 8. The efficiency of global and shared memory in matrix multiplication using glabal and shared memory without padding

3.3. Problem 3.3

The source code for problem 3.3 is matrixMulGSmemPadd.cu. The programming strategy is the same in problem 3.2. The only difference is adding a column in the left of shared memory tile because the access mode is 4-byte (see line 6, 36 and 37 in the source code).

The results are presented in Figure 9 and Figure 10. Figure 9 shows the execution time of matrix multiplication using shared memory with padding, 12.498 ms.

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvcc -arch=sm_61 -o prob3_3 matrixMulGSmemPadd.cu
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof ./prob3_3
==27761== NVPROF is profiling process 27761, command: ./prob3_3
Matrix Multiplication OK!
==27761== Profiling application: ./prob3_3
==27761== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 69.79%  12.498ms         1  12.498ms  12.498ms  12.498ms  matrixMulGSmemPadd(int*, int*, int*, int)
 18.82%  3.3711ms         2  1.6856ms  1.6839ms  1.6872ms  [CUDA memcpy HtoD]
 11.39%  2.0388ms         1  2.0388ms  2.0388ms  2.0388ms  [CUDA memcpy DtoH]

==27761== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.91%  175.08ms         3  58.360ms  152.70us  174.77ms  cudaMalloc
  9.59%  18.883ms         3  6.2943ms  1.8063ms  15.213ms  cudaMemcpy
  0.99%  1.9460ms         3  648.65us  222.40us  864.12us  cudaFree
  0.35%  687.98us       182  3.7800us     170ns  149.57us  cuDeviceGetAttribute
  0.11%  213.75us         2  106.88us  106.29us  107.46us  cuDeviceTotalMem
  0.04%  69.648us         2  34.824us  31.443us  38.205us  cuDeviceGetName
  0.01%  25.112us         1  25.112us  25.112us  25.112us  cudaLaunch
  0.00%  3.3910us         4     847ns     269ns  2.3970us  cudaSetupArgument
  0.00%  2.0630us         6     343ns     219ns     532ns  cuDeviceGet
  0.00%  1.5750us         3     525ns     220ns     983ns  cuDeviceGetCount
  0.00%  1.0790us         1  1.0790us  1.0790us  1.0790us  cudaConfigureCall
```

Figure 9. The execution time of matrix multiplication using shared memory with padding

```
20184187@eelab5:~/gpu_programming/hw/hw3$ nvprof --metrics gst_efficiency,gld_efficiency,shared_store_transactions_per_request,shared_load_trans
actions_per_request ./prob3_3
==27801== NVPROF is profiling process 27801, command: ./prob3_3
==27801== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==27801== Replaying kernel "matrixMulGSmemPadd(int*, int*, int*, int)" (done)
Matrix Multiplication OK!
==27801== Profiling application: ./prob3_3
==27801== Profiling result:
==27801== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1070 (0)"
    Kernel: matrixMulGSmemPadd(int*, int*, int*, int)
          1                            gst_efficiency                Global Memory Store Efficiency     100.00%     100.00%     100.00%
          1                            gld_efficiency                 Global Memory Load Efficiency     100.00%     100.00%     100.00%
          1     shared_store_transactions_per_request   Shared Memory Store Transactions Per Request    1.000000    1.000000    1.000000
          1      shared_load_transactions_per_request    Shared Memory Load Transactions Per Request    1.000000    1.000000    1.000000
```

Figure 10. The efficiency of global and shared memory in matrix multiplication using shared memory with padding