ECE 5720 HW4
Hongliang Si (hs983)

Introduction

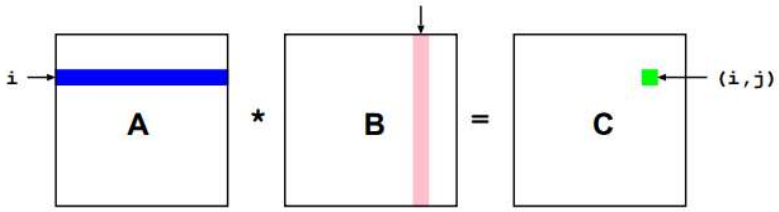
In this homework, the student explored the performance of using sheared memory in GPU with CUDA programming language, and the effect of bank conflict. The student measured the performance by calculating the matrix-matrix multiplication of two progressively large matrices. In the code, there are three versions of kernel: the naïve version, tiled version with bank conflict and the tiled version with no bank conflict.

Kernel 1: Naïve matrix-matrix multiplication on CUDA:

In this kernel, the computation is very straightforward. After we define the block size and matrix dimension, each thread with index of "i" and "j" will then access to the global memory and perform:

$$c_{i,j} = \sum_{k=1}^{N-1} a_{i,k} * b_{k,j}$$

Which will use the entire row i and column j.




```
i = blockIdx.y * blockDim.y + threadIdx.y;
j = blockIdx.x * blockDim.x + threadIdx.x;
```

(Grid , Block)

Figure 1: Naive matrix-matrix multiply [1]

Kernel 2: Tiled matrix-matrix multiplication on CUDA (Bank conflict):

In this kernel, the student divided the matrix in tiles and load to shared memory to save memory access time. After we define the tile size, matrix dimension and block size, each thread with index of "i" and "j" will then copy the data from global memory to shared memory and perform:

$$c_{i,j}{}^{(m)} = \sum_{k=mT}^{(m+1)T-1} a_{i,k} * b_{k,j} + c_{i,j}{}^{(m-1)}$$

However, since all threads will try to access the same column (bank) of the shared memory, this operation will have bank conflict and thus causes a slowdown.

Kernel 2: Tiled matrix-matrix multiplication on CUDA (No bank conflict):

The problem mentioned above can be solved by transposing the input matrix. Thus, we will be accessing the bank of both matrices in horizontal direction and bank conflict can be avoided.

[1] : Figure from the lecture note "CUDA III" page 27.

Discussion:

The figure 2 shows the comparison of the three kernels in runtime with different block sizes. As we can see below, the runtime of the naïve kernel is greater than the tiled kernels by degrees of magnitude as the block dimension increases. This shows that when we parallelize the matrix-matrix multiplication on a GPU, using tiles and shared memory will provide much more gain in the runtime efficiency.

On the other hand, this graph also shows the effect from bank conflict is less significant and will decrease as we increase the block dimension. The student believes that the since there are less blocks in the grid, therefore the total number of bank conflicts drops. Moreover, the improvement on GPU's architecture could also be shrinking the delay from bank conflicts.

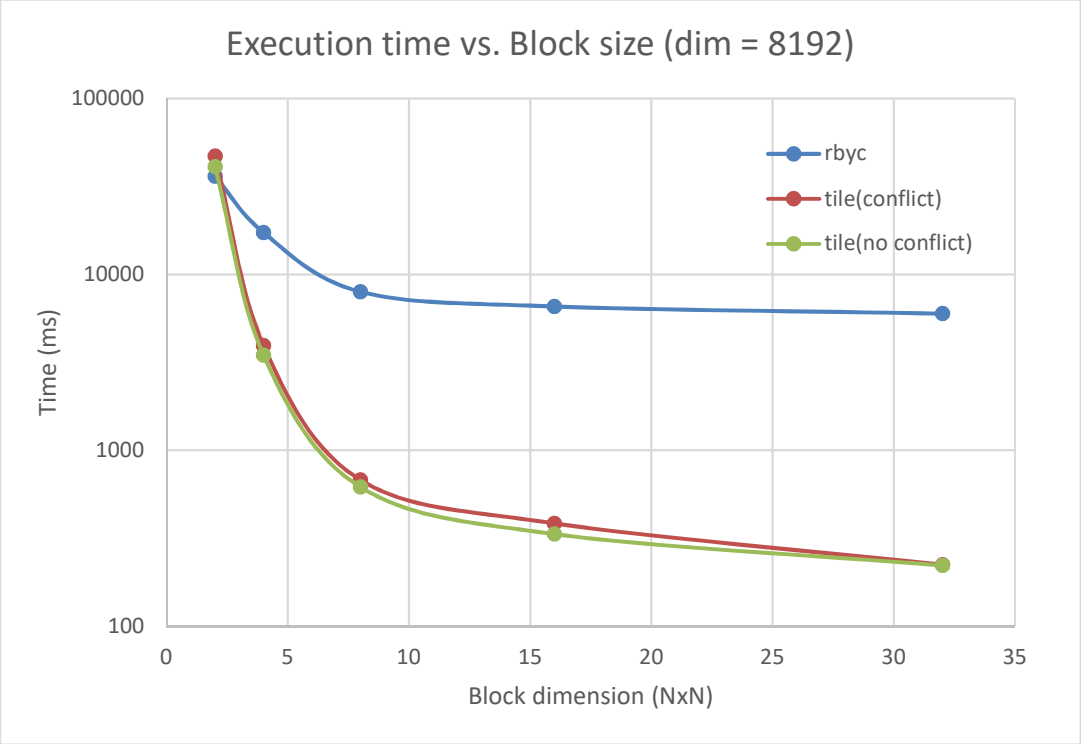


Figure 2: Runtime vs. block size

Table 1. Runtime data over fixed matrix dimension

dim	8192 (time in ms)		
block size	rbyc	tile(conflict)	tile(no conflict)
2	36193.00	47107.00	41036.00
4	17313.29	3947.09	3479.19
8	7971.72	678.45	618.25
16	6571.13	384.12	333.73
32	5983.24	223.31	221.36

In addition, as we increase the matrix dimension over a fixed block size, the effect from bank conflict is even less visible, since I was using a block size of 32. Again, the tiled kernels are way faster than the naïve kernel, by degrees of magnitude.

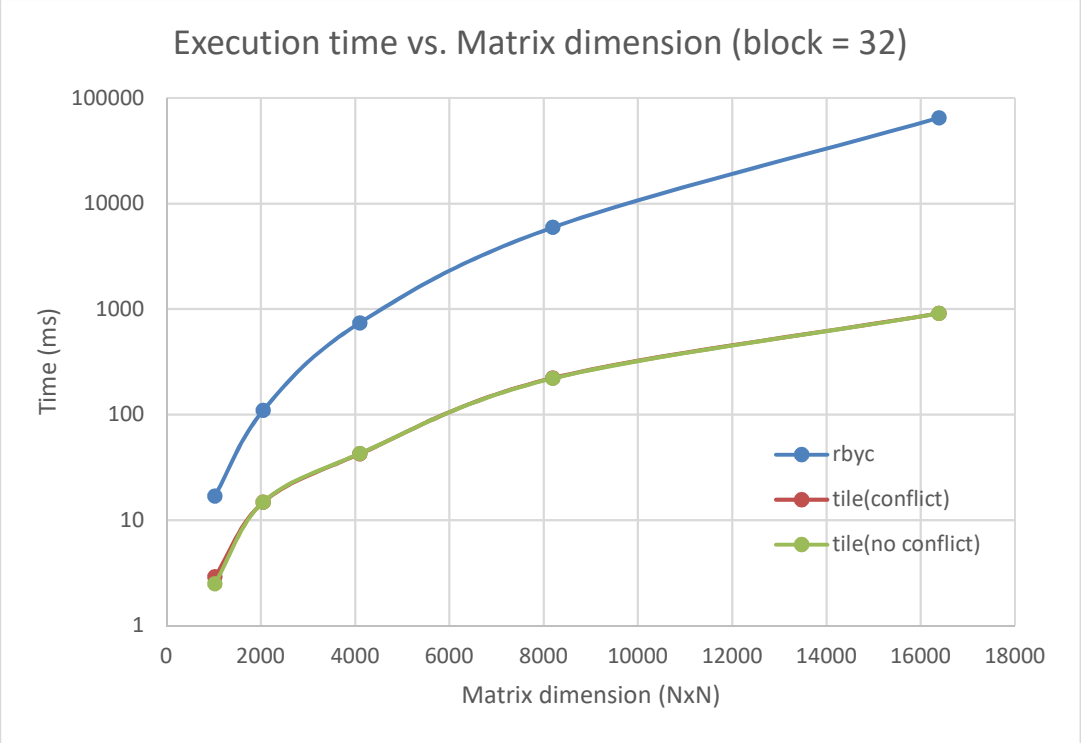


Figure 3: Runtime vs. matrix size

Table 1. Runtime data over fixed block dimension

block size	32 (time in ms)		
dim	rbyc	tile(conflict)	tile(no conflict)
1024	16.91	2.89	2.50
2048	109.43	14.76	14.77
4096	739.32	42.41	42.64
8192	5941.24	222.03	220.32
16384	64939.58	910.74	908.60

Appendix: hs983_hw4_1_to_3.c

```
/*****************************************
****************************
To Compile:
/usr/local/cuda-10.0/bin/nvcc -arch=compute_52 -o
file.out filename.cu
To run: ./file.out dim block_size
Input: dim - matrix dimension. Default: 1000
      block_size - block size. Default: 32

 ECE 5720  HW4
 matrix-matrix multiplication on CUDA
 Hongliang Si (hs983@cornell.edu)
 5/5/2019
*****************************************
****************************/
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

void check_output(float *A, int dim) {
 /* Print output for debug */
 int i,j;
 printf("\n");
 for(i = 0; i < dim; i++) {
  for(j = 0; j < dim; j++) {
   printf("%3.4f ", A[i*dim + j]);
  }
  printf(";\n");
 }
 printf("\n");
}

__global__ void MyKernel(float *d_a,float *d_b,float
*d_c,int dim){
//  navie method
 float partial = 0.0;

 int i = threadIdx.y + blockIdx.y * blockDim.y; //row i
of c
 int j = threadIdx.x + blockIdx.x * blockDim.x;
//Column j of c
 int k;
 i = i*dim;

 for(k = 0; k < dim; k++){
  partial+=d_a[i+k] * d_b[k*dim+j];
 }
 d_c[i+j] = partial;
}

__global__ void MyKernel2(float *d_a,float
*d_b,float *d_c,int dim){
 extern __shared__ float s[];  // declear a single
shared array.
 float *a_tile = s;          // Divide the shared array
into two.
 float *b_tile =
(float*)&a_tile[blockDim.x*blockDim.y];

 float partial = 0.0;
 int bx = blockIdx.x ; int by = blockIdx.y ;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int i = by * blockDim.y + ty; //row i of c
 int j = bx * blockDim.x + tx; //Column j of c
 int k,m;
 i = i * dim;
 int y = ty * blockDim.y;

 for(m = 0; m < dim/blockDim.x; m=m+blockDim.x) {
   a_tile[y+tx] = d_a[i + (m+tx)];     /* load coalesced
*/
   b_tile[y+tx] = d_b[(m+ty)*dim + j]; /* not
coalesced */
  __syncthreads();
  for(k = 0; k < blockDim.x; ++k)
    partial += a_tile[y+k] * b_tile[k*blockDim.y+tx];
/* A bank conflicts */
  __syncthreads();
  d_c[i+j] = partial;
 }
}

__global__ void MyKernel3(float *d_a,float
*d_b,float *d_cT,int dim){
 extern __shared__ float s[];  // declear a single
shared array.
 float *a_tile = s;          // Divide the shared array
into two
 float *bT_tile =
(float*)&a_tile[blockDim.x*blockDim.y];

 float partial = 0.0;
 int bx = blockIdx.x ; int by = blockIdx.y ;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int i = by * blockDim.y + ty; //row i of c
 int j = bx * blockDim.x + tx; //Column j of c
 int k,m;
 i = i * dim;
```

```
 int y = ty * blockDim.y;

 for(m = 0; m < dim/blockDim.x; m=m+blockDim.x) {
    a_tile[y+tx] = d_a[i + (m+tx)]; /* load coalesced */
   bT_tile[y+tx] = d_b[i + (m+tx)]; /* load coalesced
*/
   __syncthreads();
   for(k = 0; k < blockDim.x; ++k)
   /* No bank conflicts */
    partial +=
a_tile[ty+k*blockDim.x]*bT_tile[tx+k*blockDim.y];
   __syncthreads();
   d_cT[i+j] = partial;
 }
}

int main(int argc, char const *argv[]) {
 // Initiailize matrix dimension
 int dim = 1024,block_size = 32;
 int i, grid_size;
 if (argc > 1) {
  dim = atoi(argv[1]);
  block_size = atoi(argv[2]);
 }
 // declear host and device timer.
 srand(3);
 grid_size = dim / block_size;
 dim3 Block(block_size,block_size);
 dim3 Grid(grid_size,grid_size);
 struct timespec start,finish;
 int ntime, stime;
 float tot_time=0.0;

 // Populate matrice
 float *a  = (float*)malloc(sizeof(float)*dim*dim);
 float *bT = (float*)malloc(sizeof(float)*dim*dim);
 float *c  = (float*)malloc(sizeof(float)*dim*dim);
 float *d_a, *d_bT ,*d_c, limit=10.0; //d_bT for
transposed

 for(i = 0; i < dim*dim; i++){
  a[i]  = ((float)rand()/(float)(RAND_MAX)) * limit;
  bT[i] = ((float)rand()/(float)(RAND_MAX)) * limit;
 }

 // Allocate device memeory.
 cudaMalloc( (void**)&d_a,  dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_bT,
dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_c,  dim*dim*sizeof(float));

 // Initiailize timer & start recording.
 clock_gettime(CLOCK_REALTIME, &start);

 // Copy memory to device.

cudaMemcpy(d_a ,a ,dim*dim*sizeof(float),cudaMe
mcpyHostToDevice);

cudaMemcpy(d_bT,bT,dim*dim*sizeof(float),cudaM
emcpyHostToDevice);

 // Call CUDA kernel function.
 MyKernel<<<Grid, Block>>>(d_a,d_bT,d_c,dim);
 cudaMemcpy(c, d_c,
sizeof(float)*dim*dim,cudaMemcpyDeviceToHost);

 // Timer stop.
 cudaDeviceSynchronize();
 clock_gettime(CLOCK_REALTIME, &finish);
 ntime = finish.tv_nsec - start.tv_nsec;
 stime = (int)finish.tv_sec - (int) start.tv_sec;
 tot_time = ntime*1.0E-9 + stime;

 /* Print output for debug */
 printf("kernel#1 Time elapsed: %f ms. matrix
dimension: %d X %d\n",
 tot_time*1.0E3,dim,dim);

 // reset memory and timer.
 cudaFree(d_c); cudaFree(d_bT); cudaFree(d_a);

 /*----------------Tile method with bank conflicts:-------
----------------*/
 // Allocate memory again:
 cudaMalloc( (void**)&d_a,  dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_bT,
dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_c,  dim*dim*sizeof(float));

 // start timming.
 clock_gettime(CLOCK_REALTIME, &start);


cudaMemcpy(d_a ,a ,dim*dim*sizeof(float),cudaMe
mcpyHostToDevice);

cudaMemcpy(d_bT,bT,dim*dim*sizeof(float),cudaM
emcpyHostToDevice);

MyKernel2<<<Grid,Block,(2*Block.x*Block.y*sizeof(fl
oat))>>>(d_a,d_bT,d_c,dim);
 cudaMemcpy(c, d_c,
sizeof(float)*dim*dim,cudaMemcpyDeviceToHost);
```

```
 // Timer stop.
 cudaDeviceSynchronize();
 clock_gettime(CLOCK_REALTIME, &finish);
 ntime = finish.tv_nsec - start.tv_nsec;
 stime = (int)finish.tv_sec - (int) start.tv_sec;
 tot_time = ntime*1.0E-9 + stime;

 /* Print output for debug */
 printf("kernel#2 Time elapsed: %f ms. matrix
dimension: %d X %d\n",
 tot_time*1.0E3,dim,dim);

 // reset memory and timer.
 cudaFree(d_c); cudaFree(d_bT); cudaFree(d_a);

 /*----------------Tile method with no bank conflicts:---
-------------------*/
 // Allocate memory again:
 cudaMalloc( (void**)&d_a,  dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_bT,
dim*dim*sizeof(float));
 cudaMalloc( (void**)&d_c,  dim*dim*sizeof(float));

 // start timming.
 clock_gettime(CLOCK_REALTIME, &start);


cudaMemcpy(d_a ,a ,dim*dim*sizeof(float),cudaMe
mcpyHostToDevice);

cudaMemcpy(d_bT,bT,dim*dim*sizeof(float),cudaM
emcpyHostToDevice);

MyKernel3<<<Grid,Block,(2*Block.x*Block.y*sizeof(fl
oat))>>>(d_a,d_bT,d_c,dim);
 cudaMemcpy(c, d_c,
sizeof(float)*dim*dim,cudaMemcpyDeviceToHost);

 // Timer stop.
 cudaDeviceSynchronize();
 clock_gettime(CLOCK_REALTIME, &finish);
 ntime = finish.tv_nsec - start.tv_nsec;
 stime = (int)finish.tv_sec - (int) start.tv_sec;
 tot_time = ntime*1.0E-9 + stime;

 /* Print output for debug */
 printf("kernel#3 Time elapsed: %f ms. matrix
dimension: %d X %d\n",
 tot_time*1.0E3,dim,dim);

 // reset memory and timer.
 cudaFree(d_c); cudaFree(d_bT); cudaFree(d_a);
 free(a); free(bT); free(c);
 return 0;
}
```