CPEG 455 Course Project

Shane Cincotta

12/11/19

Problem 1

Abstract:

The goals of this problem can be broken into two sections:

1. Write a multithreaded program using the CUDA programming model to compute matrix multiplication

2. Write a multithreaded program using the CUDA programming model to compute matrix multiplication using the tiling algorithm.
    a. Measure and report the performance of two matrix sizes, N=512 and N=1024, and for each size, program your code with two tile sizes, 8 and 16.

Detailed Strategy:

To begin this problem, I first needed to create a program to calculate matrix multiplication, that is C=A*B, where A,B,C are three matrices with size NxN

```
/*
    Computes dot product of two matrices in GPU

    a = GPU device pointer to a m X n matrix
    b = GPU device pointer to a n X k matrix
    c = an m X k matrix to store the result

    assigns one thread to compute one element of matrix C. Each thread loads one row of matrix A and one column of matrix B from global memory,

    stores the result back to matrix C in the global memory

    the amount of computation = 2 * m * n * k flops

    the amount of global memory accesses = 2 * m * n * k
*/

__global__ void gpu_matrix_mult(int *a, int *b, int *c, int m, int n, int k){

    int row = blockIdx.y * blockDim.y + threadIdx.y;

    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int sum = 0;

    if( col < k && row < m) {
        for(int i = 0; i < n; i++) {
            sum += a[row * n + i] * b[i * k + col];
        }

        c[row * k + col] = sum;
    }
}
```

Fig 1 Function to calculate regular matrix multiplication

This function calculates the resulting matrix from an m*n and an n*k matrix. The result is stored in a matrix C. The GPU assigns a thread to compute an element of matrix C. Each thread loads a row of matrix A and a column of matrix B from global memory, computes the inner product and stores the result in matrix C (in global memory). The total floating point operations computed is 2*m*n*k, the amount of accesses of the global memory is 2*m*n*k.

I then began writing a cuda function to complete matrix multiplication using the tiling algorithm for square matrices.

```
__global__ void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_result, int n) {

    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;

    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    int tmp = 0;

    int idx;

    for(int j = 0; j < gridDim.x; j++){

        idx = row * n + j * BLOCK_SIZE + threadIdx.x;

        if(idx >= n*n){
           tile_a[threadIdx.y][threadIdx.x] = 0;
        }

        else{
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }

        idx = (j * BLOCK_SIZE + threadIdx.y) * n + col;

    if(idx >= n*n){
            tile_b[threadIdx.y][threadIdx.x] = 0;
        }
        else{
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }

    __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k){
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }

    if(row < n && col < n){
        d_result[row * n + col] = tmp;
    }
}
```

Fig 2 Function to calculate tiled matrix multiplication

The tiled version works by dividing the matrix into square tiles by taking the square root (thus the necessity for a square matrix). A divide and conquer approach is taken by computing the result from each of these nested matrices. In each iteration, each thread blocks loads a tile of matrix A and a tile of matrix B to shared memory (prevents bank conflicts). The result is computed and stored the result into a temp register. After all the iterations are completed, the thread block stores a tile of matrix C into global memory. The amount of floating point computations is till 2*m*n*k, but by using a tile size of B, the amount of global memory accesses is 2*m*n*k/B.

I then created two matrices, h_a & h_b, which I filled with random values, I multiplied these matrices to test my functions on.

```
//making a random matrix h_a
for (int i = 0; i < m; ++i){
    for (int j = 0; j < n; ++j) {
        h_a[i * n + j] = rand() % 1024;
    }
}


//making a random matrix h_b
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < k; ++j) {
        h_b[i * k + j] = rand() % 1024;
    }
}
```

Fig 3 Filling matrices with random values

I then copy matrix A and B from host to device memory as well as setting the correct dimensions.

```
// copy matrix A and B from host to device memory
cudaMemcpy(d_a, h_a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice);

unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

Fig 4 Copying matrices from host to device memory/setting dimensions

I am now ready to test my functions against each other.

```
//Can only run if the matrices are square because we need to compare the regular time to the square time
if(m == n && n == k){

    // start to count execution time of tiled GPU version
    cudaEventRecord(start, 0);

    //running it 100 times to get an average
    for(int i = 0; i <100; i++){
        gpu_square_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n);
    }

    //Stop recording time
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // compute time elapse on GPU function
    cudaEventElapsedTime(&gpu_square_elapsed_time_ms, start, stop);


    // start to count execution time of regular GPU version
    cudaEventRecord(start, 0);

    //can do this no matter what, again run it 100 times
    for(int i = 0; i <100; i++){
        gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_cc, m, n, k);
    }

    //Stop recording time
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // compute time elapse on standard GPU function
    cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);


    // Transfer results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_cc, d_cc, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
}

else{
    printf("Error: Please use a square matrix\n");
    exit(0);
}

//divide by 100 because we ran 100 times
printf("Average ime elapsed on standard matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n\n", m, n, n, k, gpu_square_elapsed_time_ms/100);
//divide by 100 because we ran 100 times
printf("Average time elapsed on tiled matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n\n", m, n, n, k, gpu_elapsed_time_ms/100);

printf("Average speedup Percentage from nontiled to tiled algorithim = %f%s\n", 100*gpu_elapsed_time_ms / gpu_square_elapsed_time_ms, "%");
```

Fig 5 Testing functions against each other

I begin testing by first checking if the matrices are square. Since I am comparing the tiled version to the non-tiled version, I need to guarantee that the matrices are square to be multiplied, otherwise they would not be able to run on the tiled version. I run the tiled version first with the amount of threads determined by the dimensions. I also run this 100 times so I can get an average of the time taken. I begin keeping track of time before the program enters the for loop, and stop keeping track of time immediately after the for loop ends. I store the elapsed time (ms) in the variable *gpu_square_elapsed_time_ms.* I repeat this process for the non-tiled version, except I store the results in the variable *gpu_elapsed_time_ms*. I then divide each time by 100 (since I ran them 100 times) to get the average time taken for each version. I then

calculate the percentage difference between each version. This is done by multiplying their ratio by 100.

I now know that my functions compile and run, but I need to validate the results. That is, I need to make sure that each version calculated the resulting matrix correctly. To achieve this, I ran a for loop which compares the values stored in each index of both matrices. If any values are different, I update a variable (*all_ok*) which then decides if the results are correct or not. If all the results match, *all_ok* will not be changed.

```
// validate results
int all_ok = 1;
for (int i = 0; i < m; ++i)
{
    for (int j = 0; j < k; ++j)
    {
        if(h_cc[i*k + j] != h_c[i*k + j])
        {
            all_ok = 0;
        }

    }
}
printf("\n");

if(all_ok)
{
    printf("all results are correct!!!\n");
}
else
{
    printf("incorrect results\n");
}
```

Fig 6 Validating results

Finally, I free the memory I allocated on the heap.

```
// free memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
cudaFree(d_cc);
cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);
cudaFreeHost(h_cc);
return 0;
```

Fig 7 Freeing memory on the heap

Results:

I then tested my code with a variety of inputs. First I tested my non-tiled version with two sizes, N=16 and N=32. The results were validated by computing the same calculation with the tiled version and comparing the resulting matrices.

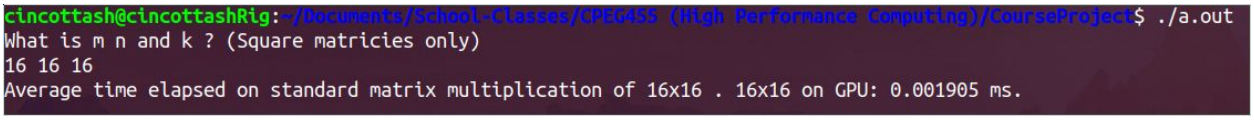

Fig 8 Computing and validating non-tiled version w/ N=16

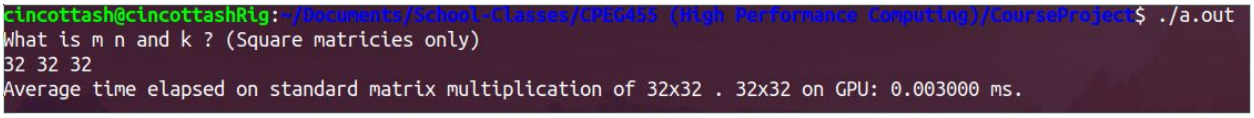

Fig 9 Computing and validating non-tiled version w/ N=32

I then tested my tiled version with two matrix sizes, N=512 and N=1024 as well as with two tile sizes, 8 and 16. I again validated my results by comparing to resulting matrix from the non-tiled version.

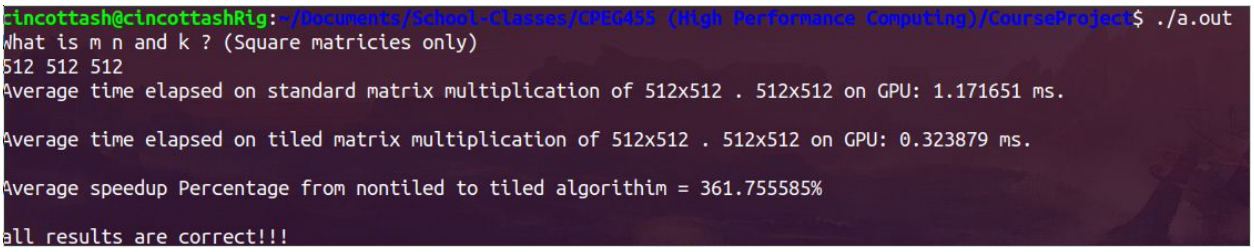

Fig 10 Computing and validating tiled version w/ N=512 and tile size = 8

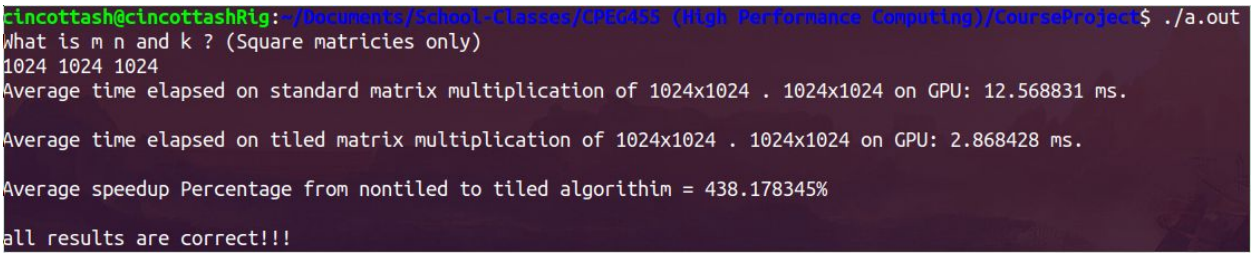

Fig 11 Computing and validating tiled version w/ N=1024 and tile size = 8

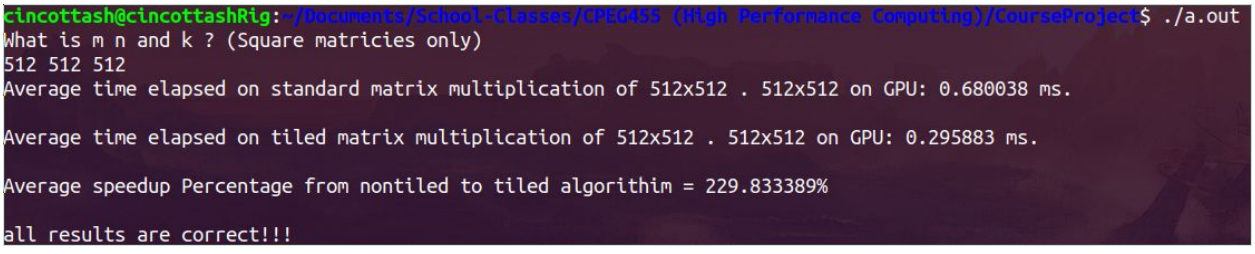

Fig 12 Computing and validating tiled version w/ N=512 and tile size = 16

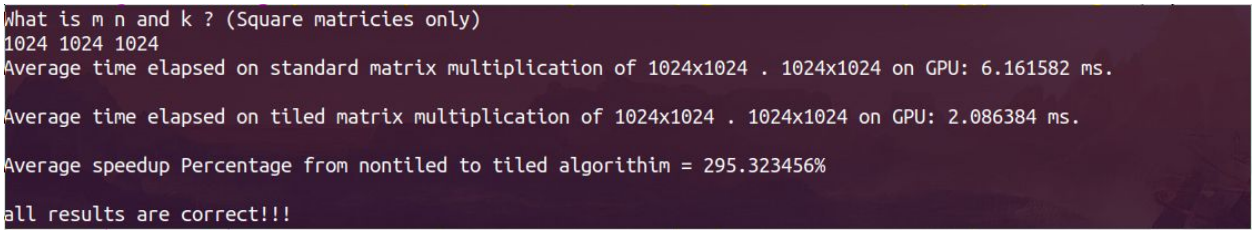

Fig 13 Computing and validating tiled version w/ N=1024 and tile size = 16

Conclusion:

After running my test cases, I was able to complete the objectives of this project. I successfully implemented matrix multiplication on my GPU with CUDA. In addition to this, I implemented a tiled version which resulted in a performance increase (with respect to time). The larger the matrix dimensions, the larger the performance increase of the tiled version compared to the non-tiled version. Also, using a smaller tile size resulted in a decrease in performance speedup.

Problem 2

Abstract:

After completing the first problem, I began to work on the 2nd problem (I initially was not aware that we had to complete both problems and thought problem 1 was the end of the project, I emailed the professor and he said to include what I have for problem 1 as well as including my attempt at problem 2).

This problem can be broken down into three parts:

1. Extend the version of 1.b with N=1024 by making one thread compute an NB*NB tile of matrix C.
    a. Additionally, try to change the number of threads NT*NT in a thread block.
    b. Also find the best NB and NT for N=1024
2. Unroll the innermost loop of the version a with factors 4 & 8.
    a. Measure the performance and explain the performance difference
3. For the unrolled versions from the problem 2.b, load all the elements of matrix A used in the tiled loops into shared memory, then replace all references to the original matrix A with the references to the shared memory copy of the matrix A.
    a. Additionally, create two versions of shared memory usage, one with bank conflicts and the other without conflicts.
        i. Explain why one has conflicts and the other doesn't
        ii. Measure the performance.

Detailed Strategy:

I began by trying to change the number of threads per thread block. After changing the number of threads per block, there was a clear difference in performance.

```
//can change this to change the number of threads per thread block, should be multiples of 32
dim3 dimBlock(64, 64);
```

Fig 1 Changing the number of threads per thread block

This lead to a clear difference in performance.

```
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.000340 ms.
all results are correct!!!
```

Fig 2 Performance difference after changing threads/thread blocks

I then attempted to make one thread compute an NB*NB tile of matrix C. However, my attempts at this transformation consistently lead to segmentation faults and I was not able to run my program without errors.

I then unrolled the innermost loop of my tiled algorithm with factors of 4 and 8. I achieved this by using *#pragma unroll factor* where *factor* is the amount I wish to unroll by. I also tested this against the resulting matrix from regular matrix multiplication function

```
for(int j = 0; j < gridDim.x; j++){

    idx = row * n + j * BLOCK_SIZE + threadIdx.x;

    if(idx >= n*n){
       tile_a[threadIdx.y][threadIdx.x] = 0;
    }

    else{
        tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
    }

    idx = (j * BLOCK_SIZE + threadIdx.y) * n + col;

if(idx >= n*n){
        tile_b[threadIdx.y][threadIdx.x] = 0;
    }
    else{
        tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
    }

__syncthreads();
    #pragma unroll 4
    for (int k = 0; k < BLOCK_SIZE; ++k){
        tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
    }
    __syncthreads();
}

if(row < n && col < n){
    d_result[row * n + col] = tmp;
}
```

Fig 3 Unrolling innermost for loop

I then ran my program with multiple n values (512, 1024), multiple unroll factors (4 and 8) as well as multiple tile sizes (8 and 16):

```
./a.out
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.398116 ms.
all results are correct!!!
```

Fig 4 Time taken with n=512, unroll factor=4, and tile size=8

```
./a.out
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.337763 ms.
all results are correct!!!
```

Fig 5 Time taken with n=512, unroll factor=4, and tile size=16

```
./a.out
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.318035 ms.
all results are correct!!!
```

Fig 6 Time taken with n=512, unroll factor=8, and tile size=8

```
./a.out
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.310698 ms.
all results are correct!!!
```

Fig 7 Time taken with n=512, unroll factor=8, and tile size=16

```
./a.out
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 4.231187 ms.
all results are correct!!!
```

Fig 8 Time taken with n=1024, unroll factor=4, and tile size=8

```
./a.out
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.645486 ms.
all results are correct!!!
```

Fig 9 Time taken with n=1024, unroll factor=4, and tile size=16

```
./a.out
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.930655 ms.
all results are correct!!!
```

Fig 10 Time taken with n=1024, unroll factor=8, and tile size=8

```
./a.out
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.197534 ms.
all results are correct!!!
```

Fig 11 Time taken with n=1024, unroll factor=8, and tile size=16

In every scenario, the performance was faster when the unroll factor was set to 8 as opposed to 4. Loop unrolling increases the programs speed by eliminating loop control instruction and loop test instructions such as pointer arithmetic. Loop unwinding erwrites loops as a repeated sequence of similar independent statements. For example, loop unrolling would produce the following transformation to a function *foo*:

```
int foo(void) {
    for (int i=0; i<5; i++)
        printf("Hello\n"); //prints hello 5 times

    return 0;
}

int bar(void)
{
    // unrolled the for loop in program foo
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");

    return 0;
}
```

Fig 12 How loop unrolling works

Unrolling a loop can increase speed but it also has disadvantages. You can clearly see that the unrolled version of function *foo* (*bar*) takes up more lines of code, around twice as much. This increase of program size can also lead to instruction cache misses which can affect performance. This explains why my unroll factor of 8 always lead to increase performance as contrasted with an unroll factor of 4.

In each iteration of my program, each thread block loads one tile of A and one tile of B from global memory to shared memory, performs the computation, and stores temporary result of C. My program avoids bank conflicts by preventing uncoalesced access. A coalesced memory transaction occurs when threads try to access memory at the same time. In order to prevent uncoalesced memory accesses, tiles are moved row by row to shared memory from global memory. The shared memory of the matrices corresponds to different memory banks, thus avoid uncoalesced memory accesses. If multiple threads use the same memory bank, those accesses will be effectively serialized. In the worst case, if all threads use the same memory bank, the memory accesses could take up to 32x longer.

A memory access can become uncoalesced when the memory is not sequential, it is spare or it is misaligned. Thus to create a bank conflict I will use an offset so the memory is not sequential.

```
__global__ void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_result, int n) {

    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y+offset;

    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x+offset;
```

Fig 13 Adding offset to force bank conflict

```
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.324559 ms.
incorrect results
```

Fig 14 Time taken with n=512, unroll factor=4, tile size=8 and offset=8

```
$ ./a.out
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.316065 ms.
incorrect results
```

Fig 15 Time taken with n=512, unroll factor=4, tile size=16 and offset=8

```
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.319720 ms.
incorrect results
```

Fig 16 Time taken with n=512, unroll factor=8, tile size=8 and offset=8

```
What is m n and k ? (Square matricies only)
512 512 512
Average time elapsed on tiled matrix multiplication of 512x512 . 512x512 on GPU: 0.269805 ms.
incorrect results
```

Fig 17 Time taken with n=512, unroll factor=8, tile size=16 and offset=8

```
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 4.399154 ms.
incorrect results
```

Fig 18 Time taken with n=1024, unroll factor=4, tile size=8 and offset=8

```
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.341509 ms.
incorrect results
```

Fig 19 Time taken with n=1024, unroll factor=4, tile size=16 and offset=8

```
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.687739 ms.
incorrect results
```

Fig 20 Time taken with n=1024, unroll factor=8, tile size=8 and offset=8

```
What is m n and k ? (Square matricies only)
1024 1024 1024
Average time elapsed on tiled matrix multiplication of 1024x1024 . 1024x1024 on GPU: 2.198666 ms.
incorrect results
```

Fig 21 Time taken with n=1024, unroll factor=8, tile size=16 and offset=8

Conclusion:

This problem had more issues than the first problem. I was not able to complete the first section entirely, as I had issues forcing a thread to compute an NB*NB tile of matrix C, but I was able to change the number of threads per thread block. I was able to show and explain the performance difference that resulted from unrolling threads. In addition, I was able to explain how bank conflicts arise, as well as creating a bank conflict in my own code.