Lab 3

George Onwubuya

September 27, 2018

1 Output Files

1.1 Output File 1

```
Setting up the problem...0.021003 s
    A: 1000 x 1000
    B: 1000 x 1000
    C: 1000 x 1000
Allocating device variables...0.173379 s
Copying data from host to device...0.004189 s
Launching kernel...0.008233 s
Copying data from device to host...0.004426 s
Verifying results...TEST PASSED


Setting up the problem...0.081349 s
    A: 2000 x 2000
    B: 2000 x 2000
    C: 2000 x 2000
Allocating device variables...0.182460 s
Copying data from host to device...0.017313 s
Launching kernel...0.060868 s
Copying data from device to host...0.012964 s
Verifying results...TEST PASSED


Setting up the problem...0.321950 s
    A: 4000 x 4000
    B: 4000 x 4000
    C: 4000 x 4000
Allocating device variables...0.186792 s
Copying data from host to device...0.068122 s
Launching kernel...0.485511 s
Copying data from device to host...0.045862 s
```

```
Verifying results...TEST PASSED


Setting up the problem...1.289725 s
    A: 8000 x 8000
    B: 8000 x 8000
    C: 8000 x 8000
Allocating device variables...0.184046 s
Copying data from host to device...0.274219 s
Launching kernel...3.888804 s
Copying data from device to host...0.182651 s
Verifying results...TEST PASSED
```

1.2 Output File 2

```
Setting up the problem...0.009253 s
    A: 1000 x 500
    B: 500 x 500
    C: 1000 x 500
Allocating device variables...0.205076 s
Copying data from host to device...0.001798 s
Launching kernel...0.002227 s
Copying data from device to host...0.002273 s
Verifying results...TEST PASSED


Setting up the problem...0.031236 s
    A: 2000 x 1000
    B: 1000 x 1000
    C: 2000 x 1000
Allocating device variables...0.186218 s
Copying data from host to device...0.006258 s
Launching kernel...0.016270 s
Copying data from device to host...0.007320 s
Verifying results...TEST PASSED


Setting up the problem...0.119207 s
    A: 4000 x 2000
    B: 2000 x 2000
    C: 4000 x 2000
Allocating device variables...0.187957 s
Copying data from host to device...0.024576 s
Launching kernel...0.121746 s
```

```
Copying data from device to host...0.022181 s
Verifying results...TEST PASSED


Setting up the problem...0.481023 s
    A: 8000 x 4000
    B: 4000 x 4000
    C: 8000 x 4000
Allocating device variables...0.186625 s
Copying data from host to device...0.102642 s
Launching kernel...0.971854 s
Copying data from device to host...0.098304 s
Verifying results...TEST PASSED
```

2 Performance Analysis

2.1 Square Matrices (n x n)

Execution Time (seconds) for Each Process					
Elements(nxn)	Setting Up	DeviceVar	Kernel	HostToDevice	DeviceToHost
1000	0.021003	0.173379	0.008233	0.004189	0.004426
2000	0.081349	0.182460	0.060868	0.017313	0.012964
4000	0.321950	0.186792	0.485511	0.068122	0.045862
8000	1.289725	0.184046	3.888804	0.274219	0.182651

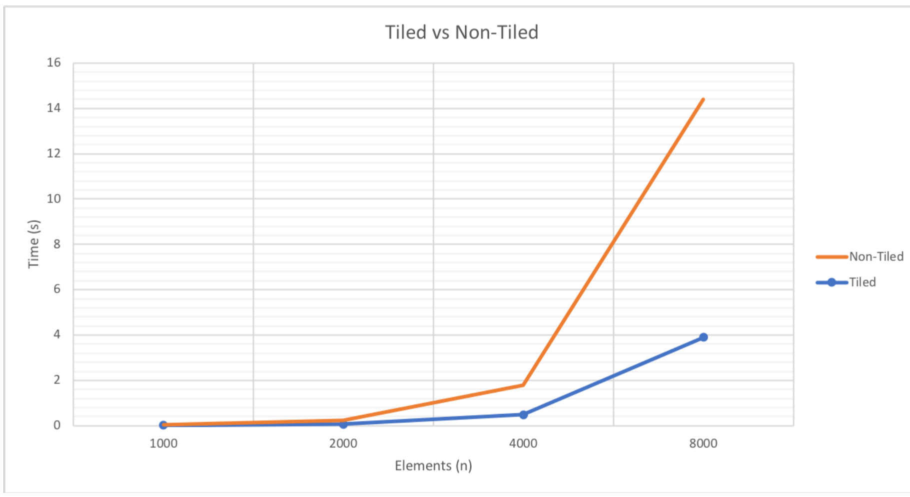
2.2 Rectangle Matrices (A = m x k and B = k x n)

Execution Time (seconds) for Each Process					
Elements (m,k,n)	Setting Up	DeviceVar	Kernel	HostToDevice	DeviceToHost
1000, 500, 500	0.009253	0.205076	0.002227	0.001798	0.002273
2000, 1000, 1000	0.031236	0.186218	0.016270	0.006258	0.007320
4000, 2000, 2000	0.119207	0.187957	0.121746	0.024576	0.022181
8000, 4000, 4000	0.481023	0.186625	0.971854	0.102642	0.098304

2.3 Square Matrix Comparison BetweenTiled vs Non-Tiled

Kernel Execution Times (seconds)		
Elements (n x n)	Tiled	Non-Tiled
1000	0.008233	0.022178
2000	0.060868	0.163624
4000	0.485511	1.308468
8000	3.888804	10.493221

2.4 Tiled vs Non-Tiled Graphical Representation



2.5 Comments

For the square matrices each process time followed a similar pattern except for the process of allocating of 'device variables'. The time taken to allocate device variables is approximately the same regardless of the number of elements. As the number of elements increase the time taken setting up the problem, launching the kernel, copying data from the host to the device and vice versa also increases. There is an observable direct proportional relationship between these processes and the number of elements. The same conclusion can be made for the rectangular matrices.

There is an exponential increase in the kernel execution times associated with the tiled and non-tiled. However the non-tiled line graph shows a sharp increase in kernel execution time as the number of elements increase. This can be seen by the large spike between 4000 and 8000 elements on the graph. The tiled line graph does not exhibit that spike, as the purpose of tiling is reduce the number of global memory accesses. The graph shows that tiling reduces the kernel execution time by approximately a factor of 2.7.

3 Answers

3.1 C(i)

The execution of nvcc –ptxas-options=”-v” kernel.cu revealed resource usage statistics which include 25 registers/thread, 2K (2048 bytes) of shared memory, 360 bytes of constant memory[0] and 4 bytes of constant memory[2]. There are obvious limits to the number of threads allocated to a single streaming processor as defined by hardware specifications. For the GeForce GTX 280 GPU that has a compute capability of 1.3 these specifications include: 512 threads/block, 16K shared memory/multiprocessor, 16K registers/multiprocessor, 8 blocks/multiprocessor, 1024 threads/multiprocessor and 124 registers/thread.

For this case, the block size is 16 x 16 = 256 threads and therefore the number of blocks/multiprocessor would be 1024/ 256 = 4. However, (256 threads x 25 reg/thread x 4 blocks/multiprocessor) > 16K registers/multiprocessor. The number of registers per multiprocessor is a limiting factor and therefore only 2 blocks can run on a single microprocessor which would yield (256 x 2 x 25) < 16K and hence the total number of threads that can be simultaneously scheduled for 30 single multiprocessors would be 256 x 2 x 30 = 15360.

4 Main

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ↪  ******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "kernel.cu"
#include "support.h"

int main (int argc, char *argv[])
{

    Timer timer;
    cudaError_t cuda_ret;
```

```

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

    float *A_h, *B_h, *C_h;
    float *A_d, *B_d, *C_d;
    size_t A_sz, B_sz, C_sz;
    unsigned matArow, matAcol;
    unsigned matBrow, matBcol;
    dim3 dim_grid, dim_block;

    if (argc == 1) {
        matArow = 1000;
        matAcol = matBrow = 1000;
        matBcol = 1000;
    } else if (argc == 2) {
        matArow = atoi(argv[1]);
        matAcol = matBrow = atoi(argv[1]);
        matBcol = atoi(argv[1]);
    } else if (argc == 4) {
        matArow = atoi(argv[1]);
        matAcol = matBrow = atoi(argv[2]);
        matBcol = atoi(argv[3]);
    } else {
        printf("\n    Invalid input parameters!"
      "\n    Usage: ./sgemm-tiled                # All matrices are 1000 x 1000"
      "\n    Usage: ./sgemm-tiled <m>            # All matrices are m x m"
      "\n    Usage: ./sgemm-tiled <m> <k> <n>    # A: m x k, B: k x n, C: m x n"
      "\n");
        exit(0);
    }

    A_sz = matArow*matAcol;
    B_sz = matBrow*matBcol;
    C_sz = matArow*matBcol;

    A_h = (float*) malloc( sizeof(float)*A_sz );
    for (unsigned int i=0; i < A_sz; i++) { A_h[i] = (rand()%100)/100.00; }

```

```
    B_h = (float*) malloc( sizeof(float)*B_sz );
    for (unsigned int i=0; i < B_sz; i++) { B_h[i] = (rand()%100)/100.00; }

    C_h = (float*) malloc( sizeof(float)*C_sz );

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));
    printf("    A: %u x %u\n    B: %u x %u\n    C: %u x %u\n", matArow, matAcol,
        matBrow, matBcol, matArow, matBcol);

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cuda_ret =  cudaMalloc((void**)&A_d, sizeof(float)*A_sz);
            if (cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    cuda_ret = cudaMalloc((void**)&B_d, sizeof(float)*B_sz);
        if (cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    cuda_ret = cudaMalloc((void**)&C_d, sizeof(float)*C_sz);
        if (cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy host variables to device ------------------------------------------

    printf("Copying data from host to device..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cuda_ret = cudaMemcpy(A_d, A_h, sizeof(float)*A_sz, cudaMemcpyHostToDevice);
        if (cuda_ret != cudaSuccess) FATAL("Unable to copy to device memory");

```

```
    cuda_ret = cudaMemcpy(B_d, B_h, sizeof(float)*B_sz, cudaMemcpyHostToDevice);
        if (cuda_ret != cudaSuccess) FATAL("Unable to copy to device memoy");


    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Launch kernel using standard sgemm interface ---------------------------
    printf("Launching kernel..."); fflush(stdout);
    startTime(&timer);
    basicSgemm('N', 'N', matArow, matBcol, matBrow, 1.0f, \
                A_d, matArow, B_d, matBrow, 0.0f, C_d, matBrow);

    cuda_ret = cudaDeviceSynchronize();
        if(cuda_ret != cudaSuccess) FATAL("Unable to launch kernel");
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy device variables from host ----------------------------------------

    printf("Copying data from device to host..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cuda_ret = cudaMemcpy(C_h, C_d, sizeof(float)*C_sz, cudaMemcpyDeviceToHost);
        if (cuda_ret != cudaSuccess) FATAL("Unable to copy to host memory");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Verify correctness -----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);

    verify(A_h, B_h, C_h, matArow, matAcol, matBcol);


```

```
    // Free memory -------------------------------------------------------------

    free(A_h);
    free(B_h);
    free(C_h);

    //INSERT CODE HERE
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
    return 0;

}
```

5 Kernel

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ******************************************************************************/

#include <stdio.h>

#define TILE_SZ 16

__global__ void mysgemm(int m, int n, int k, const float *A, const float *B, float* C) {

    /********************************************************************
     *
     * Compute C = A x B
     *   where A is a (m x k) matrix
     *   where B is a (k x n) matrix
     *   where C is a (m x n) matrix
     *
     * Use shared memory for tiling
     *
```

```
    ********************************************************************/

    // INSERT KERNEL CODE HERE
    unsigned int TiRow = threadIdx.y;
    unsigned int TiCol = threadIdx.x;
    unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ float As[TILE_SZ][TILE_SZ];
    __shared__ float Bs[TILE_SZ][TILE_SZ];

    float sum = 0;


    for(unsigned int TiNum = 0; TiNum < (k-1)/TILE_SZ+1; TiNum++){
          if((row < m) && (TiNum * TILE_SZ + TiCol) < k)
                    As[TiRow][TiCol]= A[row * k + TiNum * TILE_SZ + TiCol];
          else
                    As[TiRow][TiCol] = 0;

         if((TiNum * TILE_SZ + TiRow) < k && col < n)
                    Bs[TiRow][TiCol] = B[(TiNum * TILE_SZ + TiRow) * n + col];
        else
                    Bs[TiRow][TiCol] = 0;
          __syncthreads();

        //Calculate inner product for the tile
        //Checking for matrix size to lower power and practice green computing
        if(row < m && col < n)
                for(unsigned int TiElem = 0; TiElem < TILE_SZ; TiElem++)
                        sum = sum + As[TiRow][TiElem]*Bs[TiElem][TiCol];
        __syncthreads();

        }

        //Prevent writing of output to an undefined block
        if (row < m && col < n)
                C[row * n + col] = sum;
}
```

```

void basicSgemm(char transa, char transb, int m, int n, int k, float alpha, const float *A, int lda, const float *B, int ldb, float beta, float *C, int ldc)
{
    if ((transa != 'N') && (transa != 'n')) {
        printf("unsupported value of 'transa'\n");
            return;
    }

    if ((transb != 'N') && (transb != 'n')) {
        printf("unsupported value of 'transb'\n");
        return;
    }

    if ((alpha - 1.0f > 1e-10) || (alpha - 1.0f < -1e-10)) {
        printf("unsupported value of alpha\n");
        return;
    }

    if ((beta - 0.0f > 1e-10) || (beta - 0.0f < -1e-10)) {
        printf("unsupported value of beta\n");
        return;
    }

    // Initialize thread block and kernel grid dimensions ---------------------

   //INSERT CODE HERE
    const unsigned int BLOCK_SIZE = TILE_SZ; //use 16 x 16 thread blocks

    dim3 block(BLOCK_SIZE, BLOCK_SIZE ,1);
    dim3 grid((n + BLOCK_SIZE - 1)/BLOCK_SIZE, (m + BLOCK_SIZE -1)/BLOCK_SIZE, 1);


    // Invoke CUDA kernel -----------------------------------------------------

    //INSERT CODE HERE

    mysgemm<<< grid, block>>>(m, n, k, A, B, C);

}
```