Lab 2

George Onwubuya

September 25, 2018

1 Output Files

1.1 Output File 1

```
Setting up the problem...0.020886 s
    A: 1000 x 1000
    B: 1000 x 1000
    C: 1000 x 1000
Allocating device variables...0.176401 s
Copying data from host to device...0.004717 s
Launching kernel...0.022178 s
Copying data from device to host...0.004555 s
Verifying results...TEST PASSED


Setting up the problem...0.081405 s
    A: 2000 x 2000
    B: 2000 x 2000
    C: 2000 x 2000
Allocating device variables...0.157044 s
Copying data from host to device...0.017920 s
Launching kernel...0.163624 s
Copying data from device to host...0.015868 s
Verifying results...TEST PASSED


Setting up the problem...0.318687 s
    A: 4000 x 4000
    B: 4000 x 4000
    C: 4000 x 4000
Allocating device variables...0.155963 s
Copying data from host to device...0.071036 s
Launching kernel...1.308468 s
Copying data from device to host...0.049722 s
```

```
Verifying results...TEST PASSED


Setting up the problem...1.281662 s
    A: 8000 x 8000
    B: 8000 x 8000
    C: 8000 x 8000
Allocating device variables...0.162413 s
Copying data from host to device...0.288608 s
Launching kernel...10.493221 s
Copying data from device to host...0.186492 s
Verifying results...TEST PASSED


Setting up the problem...0.121365 s
    A: 4000 x 2000
    B: 2000 x 2000
    C: 4000 x 2000
Allocating device variables...0.157077 s
Copying data from host to device...0.025333 s
Launching kernel...0.327244 s
Copying data from device to host...0.022017 s
Verifying results...TEST PASSED
```

1.2 Output File 2

```
Setting up the problem...0.008318 s
    A: 1000 x 500
    B: 500 x 500
    C: 1000 x 500
Allocating device variables...0.179471 s
Copying data from host to device...0.001774 s
Launching kernel...0.005649 s
Copying data from device to host...0.002503 s
Verifying results...TEST PASSED


Setting up the problem...0.030971 s
    A: 2000 x 1000
    B: 1000 x 1000
    C: 2000 x 1000
Allocating device variables...0.153657 s
Copying data from host to device...0.005067 s
Launching kernel...0.043961 s
```

```
Copying data from device to host...0.005322 s
Verifying results...TEST PASSED


Setting up the problem...0.122200 s
    A: 4000 x 2000
    B: 2000 x 2000
    C: 4000 x 2000
Allocating device variables...0.157203 s
Copying data from host to device...0.027051 s
Launching kernel...0.326948 s
Copying data from device to host...0.018884 s
Verifying results...TEST PASSED


Setting up the problem...0.486641 s
    A: 8000 x 4000
    B: 4000 x 4000
    C: 8000 x 4000
Allocating device variables...0.156578 s
Copying data from host to device...0.098028 s
Launching kernel...2.614291 s
Copying data from device to host...0.102813 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	0.020886	0.176401	0.022178	0.004717	0.004555
2000	0.81405	0.157044	0.163624	0.017920	0.015868
4000	0.318687	0.155963	1.308468	0.071036	0.049722
8000	1.281662	0.162413	10.493221	0.288608	0.186492

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

Execution Time (seconds) for Each Process					
Elements (m,k,n)	Setting Up	DeviceVar	Kernel	HostToDevice	DeviceToHost
1000, 500, 500	0.008318	0.179471	0.005649	0.001774	0.002503
2000, 1000, 1000	0.030971	0.153657	0.043961	0.005067	0.005322
4000, 2000, 2000	0.122200	0.157203	00.326948	.027051	0.018884
8000, 4000, 4000	0.486641	0.156578	2.614291	0.098028	0.102813

2.3 Comments

For the the square matrices each process time followed a similar pattern except for the process of allocating of 'device variables' which shows no correlation or change as the number of elements increase. As the number of elements increase the time taken setting up the problem, allocating device variables, launching the kernel and copying data from the host to the device and vice versa also increase. There is a noticeable direct proportional relationship between these processes and the number of elements. The same conclusion can be made for the rectangular matrices. It can also be observed that the time taken to allocate device variables are approximately the same regardless of the number of elements.

3 Answers

3.1 (i)

The elements in matrix A are loaded m times and in B are loaded n times.

3.2 (ii)

If storing to the global memory is ignored then for each element the global memory is accessed twice. On the first load a floating point operation used used to multiply the elements from the corresponding row and column and on the second load a floating-point operation is used to perform addition. There are therefore 2 global memory accesses and 2 floating point operations and therefore the memory access to floating-point compute ratio is 1:1.

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                # All matrices are 1000 x 1000"
           "\n    Usage: ./sgemm <m>            # All matrices are m x m"
```

```
            "\n    Usage: ./sgemm <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 memory to device");

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

    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 from
        ↪  device");

    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>

```

```
__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
     *
     ********************************************************************/

    // INSERT KERNEL CODE HERE
    int row, col;

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

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


    if(( row < m) && (col < n))
    {
        float acc = 0;

        for(int index = 0; index < k; index++)
        {
           acc = acc + A[row * k + index] * B[index * n + col];
        }

        C[row * n + col] = acc;

    }
}

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 ---------------------

    const unsigned int BLOCK_SIZE = 16; // Use 16x16 thread blocks

    //INSERT CODE HERE

    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);

}
```