REPORT COM2039

Aksel Tahir, at01053

All of the work below is written by me

1. Matrix Multiplication

1. After completing labs 2 and 3, I have both matrix multiplication algorithms working - One uses shared device memory, the other global device memory. The matrices of the shared multiplicator are AS, BS and the resulting CS. The matrices of the global memory multiplicator are AG, BG and resulting CG. The matrices of both labs are created with (16 16 16) arguments

AS															
1.00	1.00	0.00	1.00	2.00	1.00	1.00	0.00	0.00	1.00	2.00	1.00	2.00	1.00	2.00	1.00
0.00	0.00	1.00	1.00	2.00	2.00	0.00	0.00	2.00	2.00	2.00	1.00	1.00	1.00	2.00	0.00
0.00	0.00	2.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	2.00	2.00	1.00	2.00	2.00
2.00	0.00	2.00	1.00	1.00	2.00	2.00	0.00	2.00	2.00	1.00	1.00	0.00	0.00	2.00	0.00
2.00	2.00	1.00	0.00	1.00	2.00	0.00	0.00	0.00	0.00	2.00	0.00	2.00	2.00	0.00	2.00
1.00	0.00	0.00	2.00	2.00	0.00	0.00	2.00	2.00	1.00	0.00	0.00	2.00	0.00	1.00	1.00
1.00	0.00	0.00	2.00	0.00	1.00	2.00	1.00	2.00	0.00	2.00	2.00	0.00	0.00	1.00	2.00
1.00	2.00	2.00	1.00	0.00	2.00	1.00	2.00	1.00	0.00	0.00	1.00	0.00	2.00	0.00	2.00
0.00	1.00	0.00	1.00	2.00	0.00	0.00	2.00	1.00	2.00	2.00	2.00	1.00	0.00	2.00	2.00
0.00	2.00	0.00	0.00	1.00	2.00	1.00	0.00	0.00	2.00	2.00	1.00	2.00	0.00	1.00	2.00
2.00	1.00	2.00	2.00	1.00	2.00	1.00	1.00	2.00	1.00	0.00	1.00	2.00	2.00	0.00	0.00
2.00	1.00	0.00	1.00	1.00	2.00	2.00	1.00	2.00	2.00	0.00	1.00	2.00	1.00	2.00	1.00
0.00	1.00	2.00	2.00	0.00	0.00	0.00	0.00	2.00	1.00	1.00	1.00	2.00	2.00	2.00	1.00
1.00	1.00	0.00	0.00	0.00	0.00	2.00	0.00	2.00	0.00	0.00	2.00	2.00	2.00	1.00	2.00
1.00	0.00	2.00	2.00	2.00	1.00	0.00	2.00	0.00	2.00	1.00	0.00	0.00	1.00	2.00	2.00
2.00	0.00	2.00	0.00	1.00	2.00	1.00	1.00	2.00	1.00	0.00	1.00	1.00	0.00	1.00	2.00

BC															
0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	1.00	1.00	1.00
1.00	0.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	1.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	1.00
0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00
1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
1.00	1.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00
0.00	1.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00	1.00
1.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00
1.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00
0.00	1.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00
0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00
1.00	1.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00
1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00
1.00	1.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	1.00

CS															
8.00	9.00	5.00	5.00	8.00	8.00	10.00	12.00	9.00	5.00	11.00	11.00	7.00	10.00	9.00	5.00

10.00	9.00	5.00	7.00	10.00	5.00	8.00	12.00	7.00	5.00	10.00	9.00	7.00	8.00	6.00	3.00
9.00	8.00	4.00	5.00	11.00	6.00	9.00	12.00	5.00	7.00	6.00	7.00	5.00	7.00	6.00	6.00
11.00	9.00	4.00	9.00	9.00	5.00	5.00	11.00	4.00	4.00	7.00	8.00	7.00	6.00	4.00	2.00
9.00	7.00	6.00	3.00	6.00	7.00	7.00	14.00	9.00	6.00	9.00	9.00	4.00	11.00	9.00	6.00
4.00	10.00	4.00	3.00	10.00	8.00	9.00	7.00	6.00	4.00	7.00	9.00	4.00	7.00	6.00	5.00
10.00	11.00	2.00	7.00	10.00	3.00	6.00	11.00	4.00	5.00	7.00	9.00	6.00	8.00	5.00	7.00
10.00	12.00	7.00	4.00	11.00	5.00	8.00	13.00	4.00	8.00	7.00	5.00	5.00	8.00	7.00	10.00
7.00	13.00	4.00	5.00	12.00	8.00	11.00	11.00	10.00	6.00	12.00	12.00	8.00	9.00	8.00	8.00
8.00	9.00	3.00	4.00	5.00	6.00	7.00	13.00	11.00	5.00	10.00	10.00	6.00	10.00	8.00	5.00
9.00	12.00	8.00	5.00	11.00	7.00	10.00	14.00	5.00	5.00	7.00	9.00	5.00	9.00	8.00	7.00
10.00	13.00	4.00	6.00	10.00	9.00	10.00	13.00	7.00	7.00	9.00	10.00	8.00	10.00	8.00	6.00
10.00	10.00	7.00	7.00	12.00	5.00	10.00	12.00	6.00	5.00	10.00	8.00	6.00	9.00	6.00	7.00
9.00	11.00	3.00	5.00	9.00	5.00	8.00	13.00	5.00	5.00	6.00	7.00	5.00	11.00	7.00	7.00
9.00	11.00	7.00	5.00	13.00	7.00	9.00	9.00	7.00	8.00	12.00	10.00	8.00	6.00	6.00	7.00
10.00	9.00	3.00	6.00	10.00	6.00	5.00	12.00	5.00	6.00	5.00	8.00	4.00	7.00	4.00	4.00

AG															
0.00	0.00	0.00	1.00	2.00	0.00	2.00	0.00	1.00	1.00	0.00	0.00	2.00	1.00	1.00	2.00
0.00	0.00	2.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	2.00	1.00	2.00	0.00	0.00	0.00
1.00	0.00	2.00	1.00	2.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	2.00	1.00	0.00	0.00
1.00	2.00	1.00	1.00	1.00	2.00	2.00	1.00	0.00	1.00	2.00	0.00	2.00	0.00	1.00	0.00
2.00	0.00	0.00	2.00	1.00	1.00	0.00	0.00	2.00	2.00	1.00	2.00	0.00	1.00	2.00	1.00
2.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	2.00	2.00	2.00	0.00	1.00	1.00
0.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	2.00	2.00	2.00	0.00	2.00	0.00	0.00
0.00	1.00	2.00	2.00	2.00	0.00	2.00	2.00	1.00	1.00	2.00	0.00	0.00	0.00	2.00	0.00
2.00	1.00	1.00	0.00	2.00	2.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	1.00	0.00	2.00
0.00	2.00	2.00	2.00	0.00	1.00	2.00	2.00	1.00	1.00	0.00	1.00	0.00	1.00	2.00	2.00
2.00	1.00	2.00	2.00	0.00	0.00	1.00	2.00	2.00	2.00	2.00	0.00	0.00	0.00	0.00	1.00
1.00	2.00	1.00	1.00	1.00	0.00	1.00	2.00	0.00	2.00	1.00	0.00	0.00	0.00	2.00	2.00
2.00	0.00	2.00	0.00	1.00	1.00	0.00	1.00	0.00	0.00	1.00	2.00	0.00	1.00	0.00	2.00
0.00	2.00	1.00	2.00	0.00	0.00	2.00	1.00	2.00	0.00	1.00	0.00	2.00	2.00	1.00	1.00
2.00	0.00	2.00	0.00	2.00	2.00	1.00	0.00	0.00	0.00	0.00	2.00	2.00	1.00	1.00	0.00
1.00	0.00	0.00	2.00	1.00	0.00	1.00	1.00	1.00	2.00	2.00	0.00	1.00	1.00	0.00	1.00

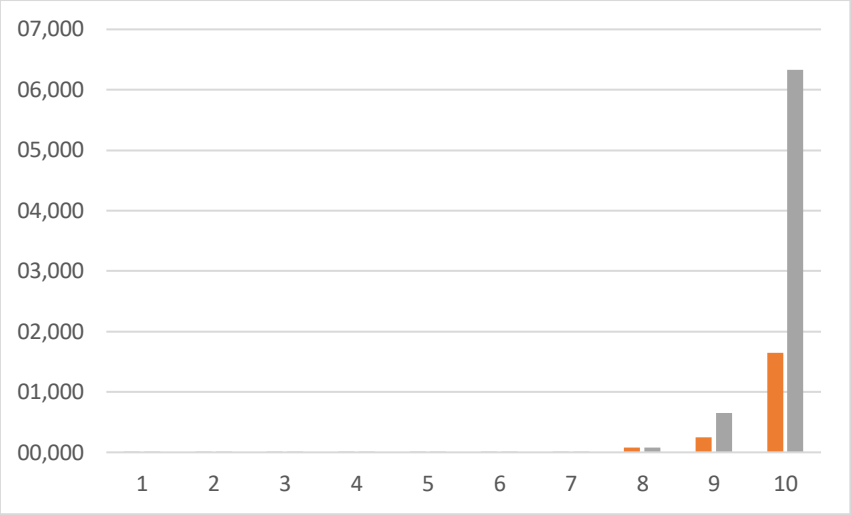
BG															
0.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	0.00	0.00
0.00	1.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00
0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00
0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00	1.00	0.00	1.00
1.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00
0.00	0.00	0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00
0.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00	1.00	1.00
0.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00
1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00
1.00	0.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00
1.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
1.00	1.00	1.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00
0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00
1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00
1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.00	0.00

0.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00
CG															
6.00	8.00	5.00	5.00	8.00	9.00	8.00	5.00	7.00	5.00	1.00	10.00	7.00	6.00	7.00	5.00
3.00	4.00	3.00	4.00	6.00	5.00	6.00	2.00	5.00	2.00	2.00	3.00	6.00	4.00	4.00	3.00
4.00	8.00	3.00	7.00	8.00	10.00	6.00	4.00	5.00	5.00	4.00	5.00	9.00	8.00	6.00	5.00
5.00	9.00	6.00	7.00	13.00	12.00	12.00	8.00	9.00	7.00	4.00	6.00	11.00	9.00	10.00	8.00
11.00	10.00	5.00	9.00	14.00	9.00	11.00	11.00	12.00	8.00	6.00	8.00	7.00	11.00	6.00	6.00
6.00	10.00	5.00	7.00	11.00	10.00	7.00	6.00	10.00	4.00	5.00	6.00	12.00	7.00	6.00	5.00
8.00	4.00	6.00	4.00	11.00	2.00	7.00	5.00	4.00	3.00	5.00	4.00	4.00	5.00	6.00	4.00
8.00	12.00	6.00	8.00	13.00	14.00	13.00	8.00	6.00	6.00	2.00	6.00	11.00	10.00	6.00	7.00
5.00	11.00	6.00	9.00	13.00	9.00	9.00	8.00	8.00	6.00	6.00	7.00	9.00	10.00	7.00	6.00
6.00	14.00	10.00	12.00	17.00	13.00	15.00	10.00	11.00	8.00	5.00	7.00	10.00	11.00	7.00	8.00
6.00	13.00	7.00	9.00	15.00	12.00	13.00	9.00	8.00	5.00	4.00	6.00	10.00	9.00	4.00	4.00
6.00	11.00	8.00	9.00	14.00	11.00	12.00	8.00	8.00	5.00	3.00	6.00	12.00	8.00	6.00	5.00
5.00	10.00	6.00	9.00	12.00	6.00	6.00	6.00	7.00	4.00	7.00	5.00	9.00	7.00	4.00	4.00
6.00	11.00	6.00	8.00	13.00	13.00	12.00	6.00	10.00	7.00	3.00	7.00	9.00	10.00	6.00	6.00
6.00	9.00	5.00	8.00	11.00	10.00	6.00	8.00	9.00	7.00	7.00	7.00	9.00	10.00	9.00	7.00
7.00	8.00	4.00	6.00	10.00	8.00	9.00	6.00	6.00	5.00	3.00	7.00	7.00	6.00	5.00	4.00

The lab 2 input matrices use the default random seed, while the lab 3 matrices use a time-based random value generator.

2. Both source codes have timing events set up.
3. I ran both of the multiplicators with arguments starting from 16 to 8192, doubling the value each time. The below tables show the results:

run No.	matrix size	lab 3 time	lab 2 time
1	16	0.0246	0.0236
2	32	0.0276	0.0246
3	64	0.0299	0.0266
4	128	0.0307	0.0399
5	256	0.0870	0.1539
6	512	0.5450	1.0680
7	1024	4.0995	9.1212
8	2048	74.7550	74.2204
9	4096	248.1410	654.9499
10	8192	1645.8027	6335.7197



The two multiplicator timings set side by side
Grey – Lab 2
Orange Lab 3
Lower values mean better performance

4. The difference between the two applications is, as previously stated, that lab2 uses global as opposed to shared memory. The hypothesis is that the first program must access the global memory module for every calculation it makes, and the time it takes to do this adds up with bigger and bigger matrices, resulting in a very well quantifiable change in the speed of the algorithm. The shared memory algorithm works by splitting up the input matrices valuables into submatrices that are stored in the shared memory modules, for which the individual threads have much faster access.
   A good explanation can be found here:
   https://stackoverflow.com/questions/14093692/whats-the-difference-between-cuda-shared-and-global-memory

   The results we get, showing that with each bigger matrix the time difference between the two algorithms gets greater and greater, prove the hypothesis.

2. Reduce

The source code:

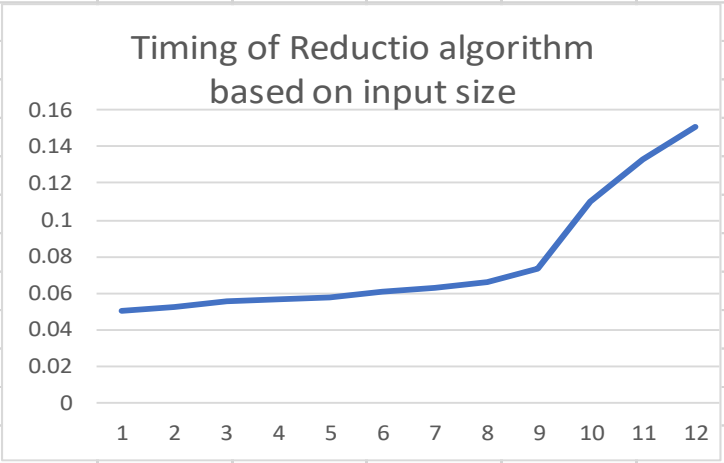
```
#include <cuda.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <numeric>
#define N 8192
#define BLOCKSIZE 16

__global__ void reduce(float* out, float* in);

float randomiser(const float min, const float max) {
    if (max == min) return min;
    else if (min < max) return (max - min) * ((float)rand() / RAND_MAX) + min;
    else return 0;
}

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

```
    // Timing events
    cudaEvent_t eventStart, eventStop;
    cudaEventCreate(&eventStart);
    cudaEventCreate(&eventStop);

    // Some variables
    int outSize = ceil(N / BLOCKSIZE);                  //determined by the total array&block s
izes
    float *deviceInput, *deviceOutput;                  //dIO pointers
    float hostInput[N];
    float hostOutput[outSize];
    int gridSize = ceil((float) N / (float) BLOCKSIZE);
    size_t size = N * sizeof(float);
    size_t size_o = gridSize * sizeof(float);

    // More outputs
    printf("size of block:  %d\n", BLOCKSIZE);
    printf("size of grid:   %d\n", gridSize);

    //Fill up the input array with random float variables from 1 to 100
    //Output some elements, randomised with time srand
    time_t t;
    srand((unsigned) time(&t));
    printf("First 5 elements: ");
    for (int i = 0; i < N; i++) {
        hostInput[i] = randomiser(1.0f, 100.0f);
    }
    for (int i=0; i<4; i++) {
        printf("%f, ", hostInput[i]);
    }
    printf("%f...And so on. \n", hostInput[4]);

    //set device memory, set grid block dimensions
    cudaMalloc((void**) &deviceInput, size);
    cudaMalloc((void**) &deviceOutput, size_o);
    cudaMemcpy(deviceInput, hostInput, size, cudaMemcpyHostToDevice);
    dim3 threadsPerBlock(BLOCKSIZE);
    dim3 blocks(gridSize);

    //first round of reduce, then sync
    cudaEventRecord(eventStart);
    reduce<<<blocks, threadsPerBlock>>>(deviceInput, deviceOutput);
    cudaError_t cudaError = cudaThreadSynchronize();
    cudaEventRecord(eventStop, 0);
    cudaEventSynchronize(eventStop);

    float milliseconds = 0;     //i use this to print the runtimes
    float ms = 0;               //i use this simply simply as a pointer

    //print time it took for the first run
    cudaEventElapsedTime(&ms, eventStart, eventStop);
    milliseconds += ms;
    printf("This run took %f ms\n", milliseconds);
```

```
    ms = 0;

    // get result from device  and remove items from mem
    cudaError = cudaMemcpy(hostOutput, deviceOutput, size_o, cudaMemcpyDeviceToHost);
    cudaFree(deviceOutput);
    cudaFree(deviceInput);

    //set device memory, reduce again
    size = sizeof(float);
    cudaMalloc((void**) &deviceInput, size_o);
    cudaMalloc((void**) &deviceOutput, size);
    cudaMemcpy(deviceInput, hostOutput, size_o, cudaMemcpyHostToDevice);
    cudaEventRecord(eventStart);
    reduce<<<1, gridSize>>>(deviceInput, deviceOutput);

    // Wait for GPU to finish, output result
    cudaThreadSynchronize();
    cudaEventRecord(eventStop);
    cudaEventSynchronize(eventStop);
    float* final_reduction = (float*) malloc(size);
    cudaError = cudaMemcpy(final_reduction, deviceOutput, size, cudaMemcpyDeviceToHost);
    printf("Input array reduced to: %f\n", *final_reduction);

    //output time it took
    cudaEventElapsedTime(&ms, eventStart, eventStop);
    milliseconds += ms;
    printf("Final time: %f ms \n", milliseconds);

    cudaFree(deviceInput);
    cudaFree(deviceOutput);
    return 0;
}

//reduction kernel. Takes
__global__ void reduce(float* inputElement, float* outputElement) {

    int arrayElementID = threadIdx.x + blockDim.x * blockIdx.x,
        threadIDx = threadIdx.x;

    for (unsigned int i = blockDim.x/2; i > 0; i /= 2) {
        if (threadIDx < i) {
            inputElement[arrayElementID] += inputElement[arrayElementID + i];
        }
        __syncthreads();
    }

    if (threadIDx == 0) outputElement[blockIdx.x] = inputElement[arrayElementID];
}
```

1. Timings:

run number	size	ms
1	16	0.050176
2	32	0.052224
3	64	0.055296
4	128	0.056144
5	256	0.057885
6	512	0.060384
7	1024	0.062542
8	2048	0.065536
9	4096	0.073728
10	8192	0.109568
11	16384	0.13312
12	32768	0.150736



2. The CPU-only algorithm might be faster for small input sizes simply because CPUs generally run on faster clocks, but the bigger the input size gets, the more advantages can be seen from using a GPU-only algorithm for reduction.

3. Not halving the input array in two would cause a more naïve approach to calculating the reduction. It's also a very inefficient way of distributing tasks for each thread. Repeatedly partitioning the array into two parts helps reduce the number of active warps by a half, as well as optimising for less thread divergence.

3. Scan

Source Code:

```
#include <cuda.h>
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#define BLOCK_SIZE 32

void scan(int N);

__global__ void scanKernel(float *input,int n,int sIndex=0);
__global__ void maxValArrKernel(float* sData,float* output);
__global__ void finScanKernel(float* initial,float* max);

int main(void)
{
    for (int i = 1; i < 20; i++)
        scan(1<<i);
    return 0;
}

void scan(int N){
    float *input;

    //begin event monitoring, allocate unified mem, init input data on host
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

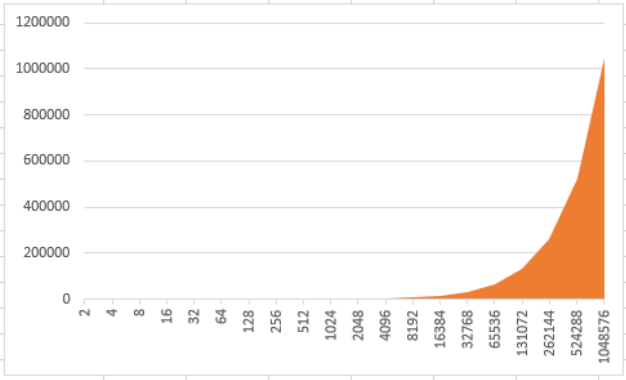
```
    cudaError_t err;

    cudaMallocManaged(&input, N * sizeof(float));

    for (int i = 0; i < N; i++)
        input[i] = 1.0f;

    // run first scan
    int grid_size = ((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
    cudaEventRecord(start);
    scanKernel<<<grid_size, BLOCK_SIZE>>>(input,BLOCK_SIZE);
    err = cudaDeviceSynchronize();                                  // Wait for GPU to fin
ish before accessing on host
    cudaEventRecord(stop);
    float timeElapsed = 0;
    cudaEventElapsedTime(&timeElapsed, start, stop);


    // run scan of the max output values, then run another scan on the kernell call
    float *scanMax;
    cudaMallocManaged(&scanMax, grid_size * sizeof(float));
    cudaEventRecord(start);
    maxValArrKernel<<<grid_size, BLOCK_SIZE>>>(input,scanMax);
    err = cudaDeviceSynchronize();
    cudaEventRecord(stop);
    float tmp=0;
    cudaEventElapsedTime(&tmp, start, stop);
    timeElapsed+=tmp;


    // run scan of the max output values again, this time with cross-block communication
    int tmpBlockSize=512;
    int scanGridSize=((grid_size + tmpBlockSize - 1) / tmpBlockSize);
    int sIndex=0;
    int endIndex=0;

    while(scanGridSize>0){
        scanGridSize=scanGridSize-
1;                                      //initial values
        endIndex=grid_size-(tmpBlockSize*(scanGridSize));

        cudaEventRecord(start);                                             //Timing and r
unning the scan Kernel.
        scanKernel<<<1, tmpBlockSize>>>(scanMax,grid_size,sIndex);

        cudaDeviceSynchronize();                                            //Synchronisin
g, getting elapsed time
        cudaEventRecord(stop);
        cudaEventElapsedTime(&tmp, start, stop);
        timeElapsed+=tmp;
        sIndex=endIndex;                                                    //setting the
startpoint for the next iteration as the endpoint of this one

        if(scanGridSize>0)
```

```
            scanMax[endIndex]+=scanMax[endIndex-1];
    }

    // Summing remaining values
    cudaEventRecord(start);
    finScanKernel<<<grid_size, BLOCK_SIZE>>>(input,scanMax);
    err = cudaDeviceSynchronize();
    cudaEventRecord(stop);
    cudaEventElapsedTime(&tmp, start, stop);
    timeElapsed+=tmp;

    printf("N - %d ,time elapsed - %f , final element - %f\n",N,timeElapsed,input[N-1]);

    //Free up mem
    cudaFree(input);
    cudaFree(scanMax);
    cudaDeviceReset();
}

__global__ void finScanKernel(float* initial,float* max){
    int thIdx = threadIdx.x + blockIdx.x * blockDim.x;

    if(blockIdx.x!=0)
        initial[thIdx]+=max[blockIdx.x-1];
}



__global__ void scanKernel(float *input,int n,int sIndex)
{
    int thIdx = threadIdx.x + blockIdx.x * blockDim.x+sIndex;
    int tid = threadIdx.x;

    __shared__ float tmp[1024];
    tmp[tid] = input[thIdx];
    __syncthreads();

    //offsetting the values by i so they match the previous ones
    //creating an initial scan
    for (int i = 1; i < n; i *= 2)
    {
        if (tid >= i)
            tmp[tid] += tmp[tid - i];
        __syncthreads();
    }
    input[thIdx] = tmp[tid];
}

__global__ void maxValArrKernel(float* input,float* output){
    int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
    int tid = threadIdx.x;

    if(tid==BLOCK_SIZE-1)
        output[blockIdx.x]=input[thIdx];
```

```
}
```

Timings:

ms	array size
0.354304	2
0.14336	4
0.134144	8
0.137216	16
0.130048	32
0.137216	64
0.141312	128
0.101376	256
0.140288	512
0.1536	1024
0.151552	2048
0.159744	4096
0.200704	8192
0.226304	16384
0.347136	32768
0.328704	65536
0.437248	131072
0.743424	262144
1.082368	524288
1.970176	1048576



4. Histogram

Below are the two code implementations for parallel histograms.

Simple histogram code:

```
//SIMPLE HISTOGRAM
#define BLOCK_SIZE 64

__global__ void simple_histogram(int* d_bins, const int* d_in, const int BIN_COUNT);

void histogramCPUcode(int Num) {

    //Some variables
    int N = Num;
    int* d_in;
    int* d_bins;
    int BIN_COUNT = 8;

    // Create timing events
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaError_t err;

    cudaMallocManaged(&d_bins, BIN_COUNT * sizeof(int));
    cudaMallocManaged(&d_in, N * sizeof(int));

    for (int i = 0; i < N; i++) {
        d_in[i] = i + 1;
    }

    for (int i = 0; i < BIN_COUNT; i++) {
        d_bins[i] = 0;
    }
```

```
    int grid_size = N / BLOCK_SIZE;

    cudaEventRecord(start);
    simple_histogram <<<grid_size, BLOCK_SIZE >> > (d_bins, d_in, BIN_COUNT);
    cudaDeviceSynchronize();

    // Wait for GPU to finish before accessing on host
    err = cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("%f\n", milliseconds);

    for (int i = 0; i < BIN_COUNT; i++) {
        printf("BIN NO. %d: Count = %d\n", i, d_bins[i]);
    }
}

int main(void) {
    int arraySize = 16;
    for (int i = 0; i < 16; i++) {
        printf("Size of array is: %d\n", arraySize);
        arraySize *= 2;
        histogramCPUcode(arraySize);
    }
}

__global__ void simple_histogram(int* d_bins, const int* d_in, const int BIN_COUNT) {
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;

    atomicAdd(&(d_bins[myBin]), 1);
}
```

And the shared memory histogram code:

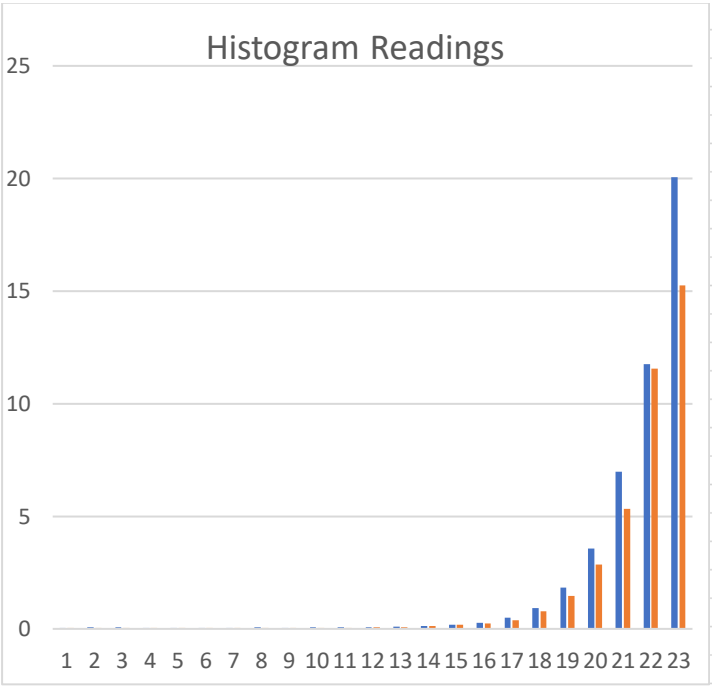
```
#include<stdio.h>
#include<numeric>
#include<stdlib.h>
#include<cuda.h>
#include <cuda_runtime.h>

#define BLOCK_SIZE 64

__global__ void shared_memory_histogram(int* d_bins, const int* d_in, const int BIN_COUNT)
;

void histogramCPUcode(int Num) {
    int N = Num;
```

```
    int* d_bins;
    int* d_in;
    int BIN_COUNT = 8;

    // Create timing events
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaError_t err;

    cudaMallocManaged(&d_bins, BIN_COUNT * sizeof(int));

    cudaMallocManaged(&d_in, N * sizeof(int));

    for (int i = 0; i < N; i++) {
        d_in[i] = i + 1;
    }

    for (int i = 0; i < BIN_COUNT; i++) {
        d_bins[i] = 0;
    }

    int grid_size = N / BLOCK_SIZE;

    cudaEventRecord(start);
    shared_memory_histogram <<<grid_size, BLOCK_SIZE >> > (d_bins, d_in, BIN_COUNT);
    cudaDeviceSynchronize();

    // Wait for GPU to finish before accessing on host
    err = cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Elapsed time was: %f ms \n", milliseconds);

    for (int i = 0; i < BIN_COUNT; i++) {
        printf("BIN NO. %d: Count = %d\n", i, d_bins[i]);
    }
}

int main(void) {
    int arraySize = 16;
    for (int i = 0; i < 16; i++) {
        printf("Size of array is: %d\n", arraySize);
        arraySize *= 2;
        histogramCPUcode(arraySize);
    }
}

__global__ void shared_memory_histogram(int* d_bins, const int* d_in, const int BIN_COUNT)
 {
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
```

```
    int myItem = d_in[myId];

    __shared__ int hist[8];
    hist[threadIdx.x] = 0;
    __syncthreads();
    int myBin = myItem % BIN_COUNT;
    atomicAdd(&(hist[myBin]), 1);
    __syncthreads();
    atomicAdd(&(d_bins[threadIdx.x]), hist[threadIdx.x]);
}
```

1. "Complete the parallel implementation of histogram using the Kernel that was outlined in Lecture 11." – See above
2. "Modify the implementation so that instead of atomicAdd() being called on global memory, it is called on partial results in local memory per block (as discussed in the handout for Lab Class 6)." – See above

3. I am testing the timings of these two algorithms on my own PC, since there is currently a blackout at the University laboratories (As of 08.06.2020). This is relevant because my timings appear to be much faster than what I was experiencing on the University GPUs, which run Quadro P4000 with a compute capability of 6.1, whereas the RTX 2060 I'm testing on has a CC of 7.5



runs	array size	Simple Histogram	Shared Mem Histogram
1	16	0.051232	0.049696
2	32	0.057344	0.053248
3	64	0.066688	0.044992
4	128	0.055744	0.045056
5	256	0.0512	0.045824
6	512	0.055296	0.043072
7	1024	0.0496	0.044928
8	2048	0.057152	0.045056
9	4096	0.054816	0.045024
10	8192	0.059488	0.047104
11	16384	0.063328	0.052448
12	32768	0.073728	0.060896
13	65536	0.105504	0.08
14	131072	0.125568	0.123008
15	262144	0.182208	0.1736
16	524288	0.275104	0.245152
17	1048576	0.492064	0.394848
18	2097152	0.937344	0.776192
19	4194304	1.84048	1.458816
20	8388608	3.554336	2.862304
21	16777216	6.979104	5.326976
22	33554432	11.745536	11.566816
23	67108864	20.057535	15.259648

The orange graph belongs to the shared memory histogram and it shows a clear improvement in time performance as opposed to the global memory algorithm. The flaw the both share though, is that no matter the memory type they use they're just not too efficient for arrays larger than a few thousand elements. Both algorithms show a trend towards [roughly] doubling performance times the larger the array gets.

5. Reflection

1. Matrix multiplication

Matrix multiplication is an incredibly useful and frequently used operation in linear algebra. The parallel implementation of matrix multiplication is often seen in machine learning algorithms.

The first step of our designing our matrix multiplication algorithm is to map out thread elements of the result matrix C. This map needs to scale for different matrix sizes, which is why matrix C is tiled with multiple

blocks – Bigger matrices require multiple blocks. Each of these threads calculates its corresponding element into C.

The biggest enemy of every computational algorithm is arithmetic intensity and computational complexity. In other words – time. To optimise our matrix multiplication, we need to maximise the number of operations performed per thread, while minimising the amount of memory accesses. Different types of memory have different speeds, with global memory being the slowest. Our algorithm needs to take this into account. To do this, we simply introduce a middleman between the global memory and the threads.

Since shared memory is only used by the threads in the block it's inside of, to use it effectively we need to divide the matrices into Sub-Matrices – each one the size of the block it's going in. This way each block can store them in their shared memory, do the necessary calculations on the device, and return a result to the host. Shared memory is faster because of the physical architecture of the GPU itself.

2. Reduce
   Parallel reduction is another frequently used algorithm – it's basically running an operator on a vector with n elements. Implementing a parallel reduction algorithm in the most abstract way consists of having a kernel that runs the operator on every adjacent element, then every fourth element, eighth, sixteenth and etc. until all elements of the original vector array are reduced to one value.

   The problems that occur with this implementation are mainly to do with, again, optimisation. Due to the way the algorithm works, we see that with the first iteration only half the threads need to be active. And then only 4^{th}, and so on, which can cause a performance downtick. Thread divergency needs to be accounted for. We need to be aware that using any amount of threads that are less than a warp is underutilizing the hardware. One of the optimisation steps is to use only sequential threads in the different iterations of the kernel.

   The usage of shared memory is also encouraged in this algorithm, since there's no reason to bottleneck the performance with global memory. The way we do it here is similar to the matrix multiplication implementation, we simply put the two values to be reduced by a thread into the shared memory.
3. Scan
   Parallel scan is very interesting since its parallelisation isn't immediately obvious. In abstract, scan takes the result of a previous computation to find the result of the next one. The Hillis and Steele Scan is a version of inclusive scan that's optimised very well for GPU computation.

   We first generate an initial scan, aka the first iteration, then run a scan on the maximal values of the output of that iteration. We do this again on the values of the second iteration. This part is done in a loop because to have a functioning overall scan, we require cross-block communication. Covering all elements in a block and adding the last one of each block to the next one will give us the needed result at the end of the iterations. The final part of the algorithm implementation is adding the sums of the remaining values in scanMax to the original array.

   A common concept amidst all the optimisation methods for these algorithms is looking for ways to use the shared memory as much as possible, instead of relying on global and unified memory.

4. Histogram
   Histograms are basically graphs that have a set of bins, each of which has a particular measurement. The dataset fed into the histogram is sorted into the bins, and we get the resulting graph.

   Implementing a histogram algorithm can be done via atomics – we add an atomic addition. Atomic operations are used when we don't want the operation to be interrupted by the other threads. The memory locations used by an atomicAdd are locked down during the read/write process. This limitation can be seen as not purely parallel, because it serialises the access to different threads, but it's necessary in some cases like this one.