COM2039 Parallel Computing Coursework

I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook.

I understand that the University may submit my work as a means of checking this, to the plagiarism detection service Turnitin® UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Signed:

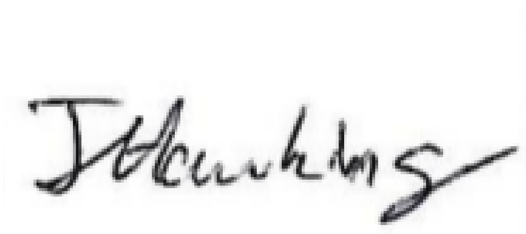


Table of Contents

Part 1: Matrix Multiplication

1. Example Matrices

LAB 2:

Matrix A
```
1.000000 1.000000 0.000000 1.000000 2.000000 1.000000 1.000000 0.000000 0.000000
1.000000 2.000000 1.000000 2.000000 1.000000 2.000000 1.000000
0.000000 0.000000 1.000000 1.000000 2.000000 2.000000 0.000000 0.000000 2.000000
2.000000 2.000000 1.000000 1.000000 1.000000 2.000000 0.000000
0.000000 0.000000 2.000000 0.000000 1.000000 1.000000 1.000000 1.000000 0.000000
0.000000 0.000000 2.000000 2.000000 1.000000 2.000000 2.000000
2.000000 0.000000 2.000000 1.000000 1.000000 2.000000 2.000000 0.000000 2.000000
2.000000 1.000000 1.000000 0.000000 0.000000 2.000000 0.000000
2.000000 2.000000 1.000000 0.000000 1.000000 2.000000 0.000000 0.000000 0.000000
0.000000 2.000000 0.000000 2.000000 2.000000 0.000000 2.000000
1.000000 0.000000 0.000000 2.000000 2.000000 0.000000 0.000000 2.000000 2.000000
1.000000 0.000000 0.000000 2.000000 0.000000 1.000000 1.000000
1.000000 0.000000 0.000000 2.000000 0.000000 1.000000 2.000000 1.000000 2.000000
0.000000 2.000000 2.000000 0.000000 0.000000 1.000000 2.000000
1.000000 2.000000 2.000000 1.000000 0.000000 2.000000 1.000000 2.000000 1.000000
0.000000 0.000000 1.000000 0.000000 2.000000 0.000000 2.000000
0.000000 1.000000 0.000000 1.000000 2.000000 0.000000 0.000000 2.000000 1.000000
2.000000 2.000000 2.000000 1.000000 0.000000 2.000000 2.000000
0.000000 2.000000 0.000000 0.000000 1.000000 2.000000 1.000000 0.000000 0.000000
2.000000 2.000000 1.000000 2.000000 0.000000 1.000000 2.000000
2.000000 1.000000 2.000000 2.000000 1.000000 2.000000 1.000000 1.000000 2.000000
1.000000 0.000000 1.000000 2.000000 2.000000 0.000000 0.000000
2.000000 1.000000 0.000000 1.000000 1.000000 2.000000 2.000000 1.000000 2.000000
2.000000 0.000000 1.000000 2.000000 1.000000 2.000000 1.000000
0.000000 1.000000 2.000000 2.000000 0.000000 0.000000 0.000000 0.000000 2.000000
1.000000 1.000000 1.000000 2.000000 2.000000 2.000000 1.000000
1.000000 1.000000 0.000000 0.000000 0.000000 0.000000 2.000000 0.000000 2.000000
0.000000 0.000000 2.000000 2.000000 2.000000 1.000000 2.000000
1.000000 0.000000 2.000000 2.000000 2.000000 1.000000 0.000000 2.000000 0.000000
2.000000 1.000000 0.000000 0.000000 1.000000 2.000000 2.000000
2.000000 0.000000 2.000000 0.000000 1.000000 2.000000 1.000000 1.000000 2.000000
1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 2.000000
```

Matrix B
```
0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000
1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000
0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000
0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000
1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 1.000000
```

1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000
0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000
1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000
0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000
0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000
0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000
0.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000
1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 1.000000 1.000000
1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000

Matrix C

8.000000 9.000000 5.000000 5.000000 8.000000 8.000000 10.000000 12.000000
9.000000 5.000000 11.000000 11.000000 7.000000 10.000000 9.000000 5.000000
10.000000 9.000000 5.000000 7.000000 10.000000 5.000000 8.000000 12.000000
7.000000 5.000000 10.000000 9.000000 7.000000 8.000000 6.000000 3.000000
9.000000 8.000000 4.000000 5.000000 11.000000 6.000000 9.000000 12.000000
5.000000 7.000000 6.000000 7.000000 5.000000 7.000000 6.000000 6.000000
11.000000 9.000000 4.000000 9.000000 9.000000 5.000000 5.000000 11.000000
4.000000 4.000000 7.000000 8.000000 7.000000 6.000000 4.000000 2.000000
9.000000 7.000000 6.000000 3.000000 6.000000 7.000000 7.000000 14.000000
9.000000 6.000000 9.000000 9.000000 4.000000 11.000000 9.000000 6.000000
4.000000 10.000000 4.000000 3.000000 10.000000 8.000000 9.000000 7.000000
6.000000 4.000000 7.000000 9.000000 4.000000 7.000000 6.000000 5.000000
10.000000 11.000000 2.000000 7.000000 10.000000 3.000000 6.000000 11.000000
4.000000 5.000000 7.000000 9.000000 6.000000 8.000000 5.000000 7.000000
10.000000 12.000000 7.000000 4.000000 11.000000 5.000000 8.000000 13.000000
4.000000 8.000000 7.000000 5.000000 5.000000 8.000000 7.000000 10.000000
7.000000 13.000000 4.000000 5.000000 12.000000 8.000000 11.000000 11.000000
10.000000 6.000000 12.000000 12.000000 8.000000 9.000000 8.000000 8.000000
8.000000 9.000000 3.000000 4.000000 5.000000 6.000000 7.000000 13.000000
11.000000 5.000000 10.000000 10.000000 6.000000 10.000000 8.000000 5.000000
9.000000 12.000000 8.000000 5.000000 11.000000 7.000000 10.000000 14.000000
5.000000 5.000000 7.000000 9.000000 5.000000 9.000000 8.000000 7.000000
10.000000 13.000000 4.000000 6.000000 10.000000 9.000000 10.000000 13.000000
7.000000 7.000000 9.000000 10.000000 8.000000 10.000000 8.000000
6.0000010.000000 10.000000 7.000000 7.000000 12.000000 5.000000 10.000000
12.000000 6.000000 5.000000 10.000000 8.000000 6.000000 9.000000 6.000000
7.000000 9.000000 11.000000 3.000000 5.000000 9.000000 5.000000 8.000000
13.000000 5.000000 5.000000 6.000000 7.000000 5.000000 11.000000 7.000000
7.000000 9.000000 11.000000 7.000000 5.000000 13.000000 7.000000 9.000000
9.000000 7.000000 8.000000 12.000000 10.000000 8.000000 6.000000 6.000000
7.000000
10.000000 9.000000 3.000000 6.000000 10.000000 6.000000 5.000000 12.000000
5.000000 6.000000 5.000000 8.000000 4.000000 7.000000 4.000000 4.000000

LAB3:

Matrix A

```
1.000000 1.000000 0.000000 1.000000 2.000000 1.000000 1.000000 0.000000 0.000000
1.000000 2.000000 1.000000 2.000000 1.000000 2.000000 1.000000
0.000000 0.000000 1.000000 1.000000 2.000000 2.000000 0.000000 0.000000 2.000000
2.000000 2.000000 1.000000 1.000000 1.000000 2.000000 0.000000
0.000000 0.000000 2.000000 0.000000 1.000000 1.000000 1.000000 1.000000 0.000000
0.000000 0.000000 2.000000 2.000000 1.000000 2.000000 2.000000
2.000000 0.000000 2.000000 1.000000 1.000000 2.000000 2.000000 0.000000 2.000000
2.000000 1.000000 1.000000 0.000000 0.000000 2.000000 0.000000
2.000000 2.000000 1.000000 0.000000 1.000000 2.000000 0.000000 0.000000 0.000000
0.000000 2.000000 0.000000 2.000000 2.000000 0.000000 2.000000
1.000000 0.000000 0.000000 2.000000 2.000000 0.000000 0.000000 2.000000 2.000000
1.000000 0.000000 0.000000 2.000000 0.000000 1.000000 1.000000
1.000000 0.000000 0.000000 2.000000 0.000000 1.000000 2.000000 1.000000 2.000000
0.000000 2.000000 2.000000 0.000000 0.000000 1.000000 2.000000
1.000000 2.000000 2.000000 1.000000 0.000000 2.000000 1.000000 2.000000 1.000000
0.000000 0.000000 1.000000 0.000000 2.000000 0.000000 2.000000
0.000000 1.000000 0.000000 1.000000 2.000000 0.000000 0.000000 2.000000 1.000000
2.000000 2.000000 2.000000 1.000000 0.000000 2.000000 2.000000
0.000000 2.000000 0.000000 0.000000 1.000000 2.000000 1.000000 0.000000 0.000000
2.000000 2.000000 1.000000 2.000000 0.000000 1.000000 2.000000
2.000000 1.000000 2.000000 2.000000 1.000000 2.000000 1.000000 1.000000 2.000000
1.000000 0.000000 1.000000 2.000000 2.000000 0.000000 0.000000
2.000000 1.000000 0.000000 1.000000 1.000000 2.000000 2.000000 1.000000 2.000000
2.000000 0.000000 1.000000 2.000000 1.000000 2.000000 1.000000
0.000000 1.000000 2.000000 2.000000 0.000000 0.000000 0.000000 0.000000 2.000000
1.000000 1.000000 1.000000 2.000000 2.000000 2.000000 1.000000
1.000000 1.000000 0.000000 0.000000 0.000000 0.000000 2.000000 0.000000 2.000000
0.000000 0.000000 2.000000 2.000000 2.000000 1.000000 2.000000
1.000000 0.000000 2.000000 2.000000 2.000000 1.000000 0.000000 2.000000 0.000000
2.000000 1.000000 0.000000 0.000000 1.000000 2.000000 2.000000
2.000000 0.000000 2.000000 0.000000 1.000000 2.000000 1.000000 1.000000 2.000000
1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 2.000000
```

Matrix B

```
0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000
1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000
0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000
0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000
1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 1.000000
1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000
0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000
```

1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000
0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000
0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000
0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000
0.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 1.000000 0.000000
1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000
1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 1.000000 1.000000
1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000

Matrix C

8.000000 9.000000 5.000000 5.000000 8.000000 8.000000 10.000000 12.000000
9.000000 5.000000 11.000000 11.000000 7.000000 10.000000 9.000000 5.000000
10.000000 9.000000 5.000000 7.000000 10.000000 5.000000 8.000000 12.000000
7.000000 5.000000 10.000000 9.000000 7.000000 8.000000 6.000000 3.000000
9.000000 8.000000 4.000000 5.000000 11.000000 6.000000 9.000000 12.000000
5.000000 7.000000 6.000000 7.000000 5.000000 7.000000 6.000000 6.000000
11.000000 9.000000 4.000000 9.000000 9.000000 5.000000 5.000000 11.000000
4.000000 4.000000 7.000000 8.000000 7.000000 6.000000 4.000000 2.000000
9.000000 7.000000 6.000000 3.000000 6.000000 7.000000 7.000000 14.000000
9.000000 6.000000 9.000000 9.000000 4.000000 11.000000 9.000000 6.000000
4.000000 10.000000 4.000000 3.000000 10.000000 8.000000 9.000000 7.000000
6.000000 4.000000 7.000000 9.000000 4.000000 7.000000 6.000000 5.000000
10.000000 11.000000 2.000000 7.000000 10.000000 3.000000 6.000000 11.000000
4.000000 5.000000 7.000000 9.000000 6.000000 8.000000 5.000000 7.000000
10.000000 12.000000 7.000000 4.000000 11.000000 5.000000 8.000000 13.000000
4.000000 8.000000 7.000000 5.000000 5.000000 8.000000 7.000000 10.000000
7.000000 13.000000 4.000000 5.000000 12.000000 8.000000 11.000000 11.000000
10.000000 6.000000 12.000000 12.000000 8.000000 9.000000 8.000000 8.000000
8.000000 9.000000 3.000000 4.000000 5.000000 6.000000 7.000000 13.000000
11.000000 5.000000 10.000000 10.000000 6.000000 10.000000 8.000000 5.000000
9.000000 12.000000 8.000000 5.000000 11.000000 7.000000 10.000000 14.000000
5.000000 5.000000 7.000000 9.000000 5.000000 9.000000 8.000000 7.000000
10.000000 13.000000 4.000000 6.000000 10.000000 9.000000 10.000000 13.000000
7.000000 7.000000 9.000000 10.000000 8.000000 10.000000 8.000000 6.000000
10.000000 10.000000 7.000000 7.000000 12.000000 5.000000 10.000000 12.000000
6.000000 5.000000 10.000000 8.000000 6.000000 9.000000 6.000000 7.000000
9.000000 11.000000 3.000000 5.000000 9.000000 5.000000 8.000000 13.000000
5.000000 5.000000 6.000000 7.000000 5.000000 11.000000 7.000000 7.000000
9.000000 11.000000 7.000000 5.000000 13.000000 7.000000 9.000000 9.000000
7.000000 8.000000 12.000000 10.000000 8.000000 6.000000 6.000000 7.000000
10.000000 9.000000 3.000000 6.000000 10.000000 6.000000 5.000000 12.000000
5.000000 6.000000 5.000000 8.000000 4.000000 7.000000 4.000000 4.000000

2. Adding Timing events

Timing events added to both lab2 and lab3

LAB2:

```
void MatrixMult(const Matrix h_A, const Matrix h_B, Matrix h_C)
      {

      cudaEvent_t start, stop;
      cudaEventCreate(&start);
      cudaEventCreate(&stop);
      // Load A and B into device memory
      Matrix d_A;
      d_A.width = h_A.width; d_A.height = h_A.height;
      size_t size = h_A.width * h_A.height * sizeof(float);
      cudaMalloc(&d_A.elements, size);
      cudaMemcpy(d_A.elements, h_A.elements, size, cudaMemcpyHostToDevice);

      Matrix d_B;
      d_B.width = h_B.width; d_B.height = h_B.height;
      size = h_B.width * h_B.height * sizeof(float);
      cudaMalloc(&d_B.elements, size);
      cudaMemcpy(d_B.elements, h_B.elements, size, cudaMemcpyHostToDevice);

      // Allocate C in Device memory
      Matrix d_C;
      d_C.width = h_C.width; d_C.height = h_C.height;
      size = h_C.width * h_C.height * sizeof(float);
      cudaMalloc(&d_C.elements, size);

      // Invoke Kernel
      dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
      dim3 dimGrid(d_B.width / dimBlock.x, d_A.height / dimBlock.y);
      cudaEventRecord(start);
      MatrixMultKern<<< dimGrid, dimBlock >>>(d_A, d_B, d_C);
      cudaEventRecord(stop);
      // Read C from Device to Host
      cudaMemcpy(h_C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
      cudaEventSynchronize(stop);
      float milliseconds = 0;
      cudaEventElapsedTime(&milliseconds, start, stop);
      printf("\n");
      printf("Elapsed time was: %f milliseconds", milliseconds);
      printf("\n");

      // Free Device Memory
      cudaFree(d_A.elements);
      cudaFree(d_B.elements);
      cudaFree(d_C.elements);


}
```

LAB3:

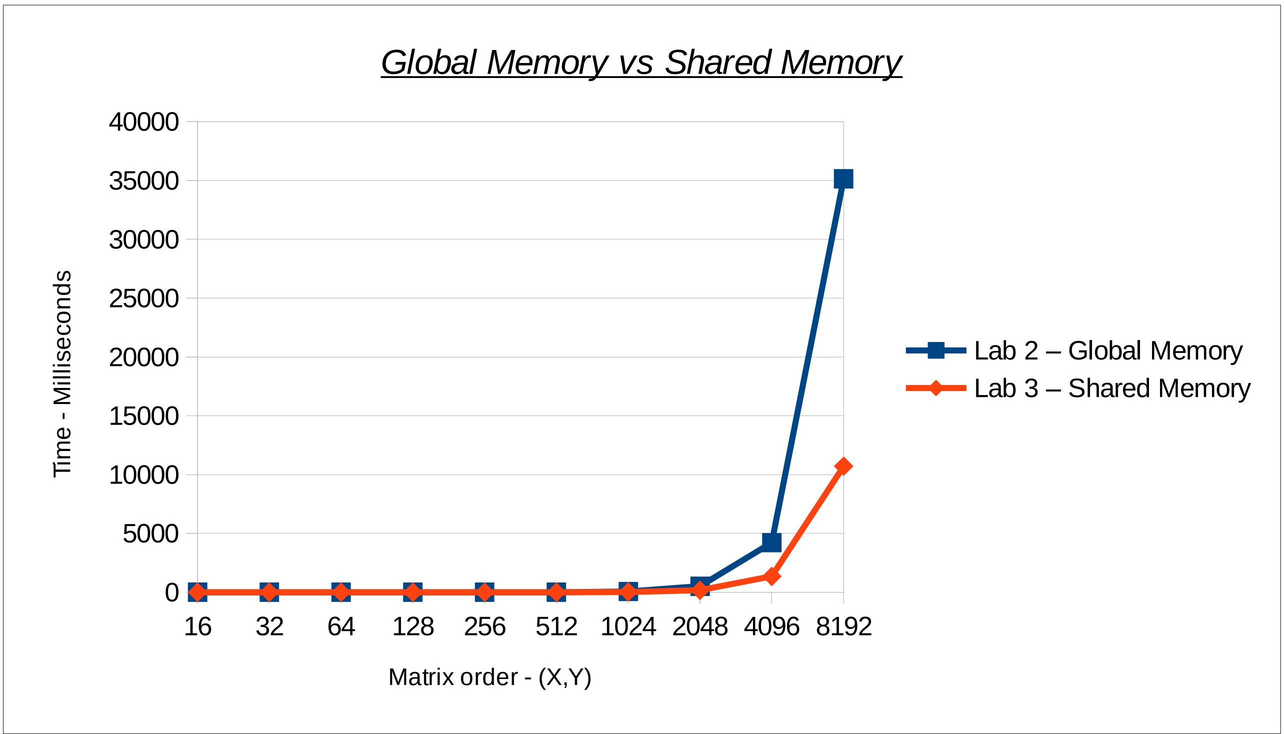
```
void MatrixMult(const Matrix h_A, const Matrix h_B, Matrix h_C) {

	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	// Load A and B to device memory
	Matrix d_A;
	d_A.width = d_A.stride = h_A.width;
	d_A.height = h_A.height;
	size_t size = h_A.width * h_A.height * sizeof(float);

	cudaError_t err = cudaMalloc(&d_A.elements, size);
	printf("CUDA malloc h_A: %s\n",cudaGetErrorString(err));
	cudaMemcpy(d_A.elements, h_A.elements, size, cudaMemcpyHostToDevice);

	Matrix d_B;
	d_B.width = d_B.stride = h_B.width;
	d_B.height = h_B.height;
	size = h_B.width * h_B.height * sizeof(float);

	err = cudaMalloc(&d_B.elements, size);
	printf("CUDA malloc h_B: %s\n",cudaGetErrorString(err));
	cudaMemcpy(d_B.elements, h_B.elements, size, cudaMemcpyHostToDevice);

	// Allocate C in device memory
	Matrix d_C;
	d_C.width = d_C.stride = h_C.width;
	d_C.height = h_C.height;
	size = h_C.width * h_C.height * sizeof(float);

	err = cudaMalloc(&d_C.elements, size);
	printf("CUDA malloc h_C: %s\n",cudaGetErrorString(err));

	// Invoke kernel
	dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
	dim3 dimGrid(h_B.width / dimBlock.x, h_A.height / dimBlock.y);

	cudaEventRecord(start);
	MultSharedKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

	err = cudaThreadSynchronize();
	cudaEventRecord(stop);
	printf("Run kernel: %s\n", cudaGetErrorString(err));

	// Read C from device memory
	err = cudaMemcpy(h_C.elements, d_C.elements, size,
	cudaMemcpyDeviceToHost);
	printf("Copy h_C off device: %s\n",cudaGetErrorString(err));
	cudaEventSynchronize(stop);
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, start, stop);
	printf("Elapsed time was: %f milliseconds", milliseconds);

	// Free device memory
	cudaFree(d_A.elements);
	cudaFree(d_B.elements);
	cudaFree(d_C.elements); }
```

3. Recording Timings for scaled matrices

Results

Matrix Order	Lab 2 Execution Time /Milliseconds	Lab 3 Execution Time /Milliseconds
(16,16)	0.017228	0.018816
(32,32)	0.019546	0.030912
(64,64)	0.030528	0.035072
(128,128)	0.108032	0.078272
(256,256)	0.747232	0.348704
(512,512)	5.83091	3.376256
(1024,1024)	64.21312	22.313087
(2048,2048)	512.1231	171.155685
(4096,4096)	4212.902332	1349.633545
(8192,8192)	35143.3221	10715.505859

Plot



4. Discussion

Hypothesis - *Shared Memory will give us a performance gain over Global Memory when performing matrix multiplication.*

Reject or confirm – From the results shown above the hypothesis can be confirmed. From the graph, we can see a clear increase in performance gain with shared memory as the size of the matrix increases. If we compare the time take for Global and shared memory for a **16*16** matrix there isn't much difference **"0.017228"** between and **"0.018816"**. However, if we compare the time taken for a **8192*8192** matrix there is a clear difference in execution time, global memory takes **"35143.3221"** whereas shared memory takes **"10715.505859"**. By the time global memory has executed the matrix, shared memory could've run it roughly three times which shows a noticeable performance gain.

Explanation of outcome – From memory hierarchy, we know that *Shared Memory* has a higher performance rate than *Global Memory*. We also know that shared memory is on-chip and gives all other threads in a given block access to a particular thread allowing us to reuse data instead of having to load it in every-time. Global memory is slightly different as it operates off-chip but provides all threads access to data for a given kernel execution. One important thing to note is that if we were only using the data once and there was no reuse of data between different threads in a block then shared memory would actually be slower. This is because when we copy data from global memory to shared memory, it still counts as a global transaction. Reading from shared memory is faster, but it doesn't matter because we've already read them from global memory thus the second step which involves reading from shared memory is just an extra step which reduces overall performance and is not needed. However, in this case, we are reusing data between different threads in a given block hence shared memory will be faster than global memory.This experiment was tested on "Nvidia GTX 745" which has a "Maxwell" GPU architecture, we know that Maxwell architecture has a compute capability of 5.0.Interestingly this was actually the first GPU from the 700 series to run the Maxwell architecture. With this architecture Nvidia focused on increasing the efficiency of the the GPU. When comparing the CPU and GPU there is little significant difference in execution times for matrix sizes up to 1000x1000, but there is a speed-up of around 3.5 times for matrices of size 8000x8000.The primary importance when it comes to using shared memory is the use of scheduling operations to reuse data. From looking at matrix multiplication, we know that inner products from "Matrix A" and "Matrix B" access the same data at the same time. If we highlight the inner products from A and B we form squares called tiles, and In this case, we're using tiles of size (2x2). Originally we had "2*(A.width) global memory accesses," but now we have "A.width global memory accesses + 2*(A.width) shared memory accesses". We know that global memory is 100 times slower than shared memory when it comes to memory access which shows we will have a clear performance gain.To speed things up further we use a technique called "Striding".Due to the fact that each submatrix is represented as a "1D serialization", we must skip across the unwanted values, In this case, the skips will be "numCol" in length. For example, if we had a block size of 16, we would need to skip 16 values across to get the next value. By combining this technique with the use of sub-matrices, we reuse a lot more data and reduce the number of memory access calls. From this research, we can conclude that shared memory will shows a clear performance gain over global memory when performing matrix multiplication which is also supported by our results.

Part 2: Reduce

1. Final Reduction on CPU

```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#include <time.h>
#include <inttypes.h>
#define BLOCK_SIZE 16 // Block size depicts the number of threads within a block
#define N 1048576 // Size of array

/**
 * Defining the kernel as a global function
 */
__global__ void reduceKernel(float *d_out, float *d_in);

/**
 * Main method which invokes calls on the kernel
 * Final reduction is done on the cpu.
 */
int main(void) {

	//Initialise timespec objects to differentiate between the beginning and
	the end of the cpu operation
	struct timespec begin, end;
	// A float allocates 4 bytes of memory, in this case we assign the size
	object with N*4 Bytes to store the reduce results.
	size_t size = N*sizeof(float);
	// Size_0 Stores the result of the reduce operation which is equal to
	N/BLOCK_SIZE for the first result.
	size_t size_o = size/BLOCK_SIZE;
	// Initialise a variable of size N which stores the values of the array
	float h_in[N];
	// Initialise a float of size N/BlockSize which stores the output of the
	reduce kernel.
	float h_out[N/BLOCK_SIZE];
	// Variables used when copying from device to host and vice versa
	float *d_in, *d_out;

	/**
	 * Initialise start and stop events for timing gpu
	 */
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	// Initialise cudaError variable to check if there our illegal memory
	access issues.
	cudaError_t err;

	//Iterate through N assigning values within h_in array 1,1,1,1.....
	for (int i = 0; i < N; i++){
		h_in[i] = 1.0f;
	}
```

```
/**
 *Loading h_in into device memory using allocated float d_in
 *Allocates the correct number of bytes for d_out
 */
cudaMalloc((void**)&d_in, size);
cudaMemcpy(d_in, h_in, size, cudaMemcpyHostToDevice);
cudaMalloc((void**)&d_out, size_o);

//grid_size stores the number of blocks needed to execute the kernel on a
given array size!
int grid_size = N/BLOCK_SIZE;
printf("Grid Size is: %d\n", grid_size);
printf("Block Size is: %d\n", BLOCK_SIZE);

//Setting Up 3dimensional Block-size(x,y,z)
dim3 threadsPerBlock(BLOCK_SIZE);
//Setting Up 3dimensional Grid-size(x,y,z)
dim3 blocks(grid_size);

//Start timing on gpu
cudaEventRecord(start);

/**
 * Call Kernel on given grid_size and number of threadsPerBlock
 * blocks = grid_size = N/BLOCK_SIZE
 * threadPerBlock = BLOCK_SIZE
 */
reduceKernel<<<blocks, threadsPerBlock>>>(d_out, d_in);

// Wait for GPU to finish before accessing on host
err = cudaDeviceSynchronize();

// Stop timing on gpu
cudaEventRecord(stop);
// Synchronise all timing events and total them
cudaEventSynchronize(stop);
// Initialise float to store time
float milliseconds = 0;
// Store time between start and stop events in milliseconds.
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Elapsed time on gpu was: %f milliseconds", milliseconds);
printf("\n");

// Checking if there was an error copying h_out of the host
err = cudaMemcpy(h_out, d_out, size_o, cudaMemcpyDeviceToHost);
// Printing the result to that check
printf("Copy h_out off device: %s\n",cudaGetErrorString(err));
printf("\n");

// Start timing cpu operations (final reduction sequence)
clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &begin);
// Initialise a float to store the final reduction
float final_reduction = 0.0f;
// Iterate through the final results and total them
for (int i = 0; i < grid_size; i++) {
      final_reduction += h_out[i];
}
// Finish timing cpu operations.
clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &end);
// Storing the time in nano seconds.
```

```
        uint64_t time = 1e9 * (end.tv_sec - begin.tv_sec) + (end.tv_nsec -
        begin.tv_nsec);

        /*
         * Converting the time to milliseconds
         * 1 million nano seconds equals 1 millisecond
         * hence time/1million converts it to milliseconds.
         */
        float clocktime = (time/1e6);

        // Print the time taken on the cpu in milliseconds
        printf("Elapsed time on cpu was: %f milliseconds", clocktime);
        printf("\n");
        // Total the time on the cpu and gpu
        float overalltime =  clocktime + milliseconds;
        // Print the overall time
        printf("CPU+GPU-Total Time:%f ", overalltime);
        printf("\n");
        // Print the final reduce value
        printf("And the final reduction is: %f\n", final_reduction);

        // Free the memory allocated on the gpu
        cudaFree(d_in);   // make sure all adds at one stage are done!
        cudaFree(d_out);
}


/**
 * Reduce kernel taking two parameters d_out and d_in
 * d_in takes the input of the kernel, on the first run this is N
 * d_out takes the output of the kernel, on the first run this is N/BLOCK_SIZE
 */
__global__ void reduceKernel(float* d_out, float* d_in) {

        // ID relative to whole array
        int myId = threadIdx.x + blockDim.x * blockIdx.x;
        // Local ID within the current block
        int tid = threadIdx.x;
        //initialisation of shared memory temporary array whose size is equal to
        the BLOCK_SIZE
        __shared__ float temp[BLOCK_SIZE];
        //assign the index of tid to the index of d_in at the value my_Id
        temp[tid] = d_in[myId];
        __syncthreads();

        /*
         * Do reduction in shared memory
         * It uses logical shifts to total up values
         * Makes sure that values are greater than or equal to 1
         * Must sync all threads to prevent a race condition
         */
        for (unsigned int s = blockDim.x/2; s >= 1; s >>= 1){
                if (tid < s){
                        temp[tid] += temp[tid + s];
                }
                __syncthreads();
        // make sure all adds at one stage are done!
        } // only thread 0 writes result for this block back to global memory
        if (tid == 0){
                d_out[blockIdx.x] = temp[tid];      }  }
```

1. Final Reduction on GPU

```
#include <stdio.h>
#include <math.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#include <iostream>
#define BLOCK_SIZE 32 // Block size depicts the number of threads within a block
#define N 1048608 // Size of array
//Defining the kernel as a global function
__global__ void reduceKernel(float *d_out, float *d_in);

int main(void) {

	// A float allocates 4 bytes of memory, in this case we assign the size
object with N*4 Bytes to store the reduce results.
	size_t size = N*sizeof(float);

	// Initialise variables to access memory on the gpu
	float *d_in, *d_out;

	/**
	 * Initialise start and stop events for timing gpu
	 */
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	//Allocate d_in and d_out into memory that is managed by unified memory
	cudaMallocManaged(&d_in, size);
	cudaMallocManaged(&d_out, size);

	//Iterate through N assigning values within d_in 1,1,1,1.....
	for (int i = 0; i < N; i++) {
			d_in[i] = 1.0f;
	}

	//grid_size stores the number of blocks needed to execute the kernel on a
given array size!
	int grid_size = ceil(N / BLOCK_SIZE);

	// This represents the number of times we need to call the kernel on a
given block and array size.
	float number0fIterations = (log(N) / log(BLOCK_SIZE));

	/*
	 * Iterate through the numberOfIterations
	 * Each time we iterate through we must update the grid size
	 * Which decreases by a given ratio (grid_size / BLOCK_SIZE)
	 * For Example if we had an array size of 512 and a block size of 16     *
	*Grid_size would equal 32, then for the second iteration it would equal 2
	 */
	cudaEventRecord(start);
	for (int i = 0; i < number0fIterations ; i++) {
		if(i == 0){
		reduceKernel<<<grid_size, BLOCK_SIZE>>>(d_out, d_in);
		cudaDeviceSynchronize();
```

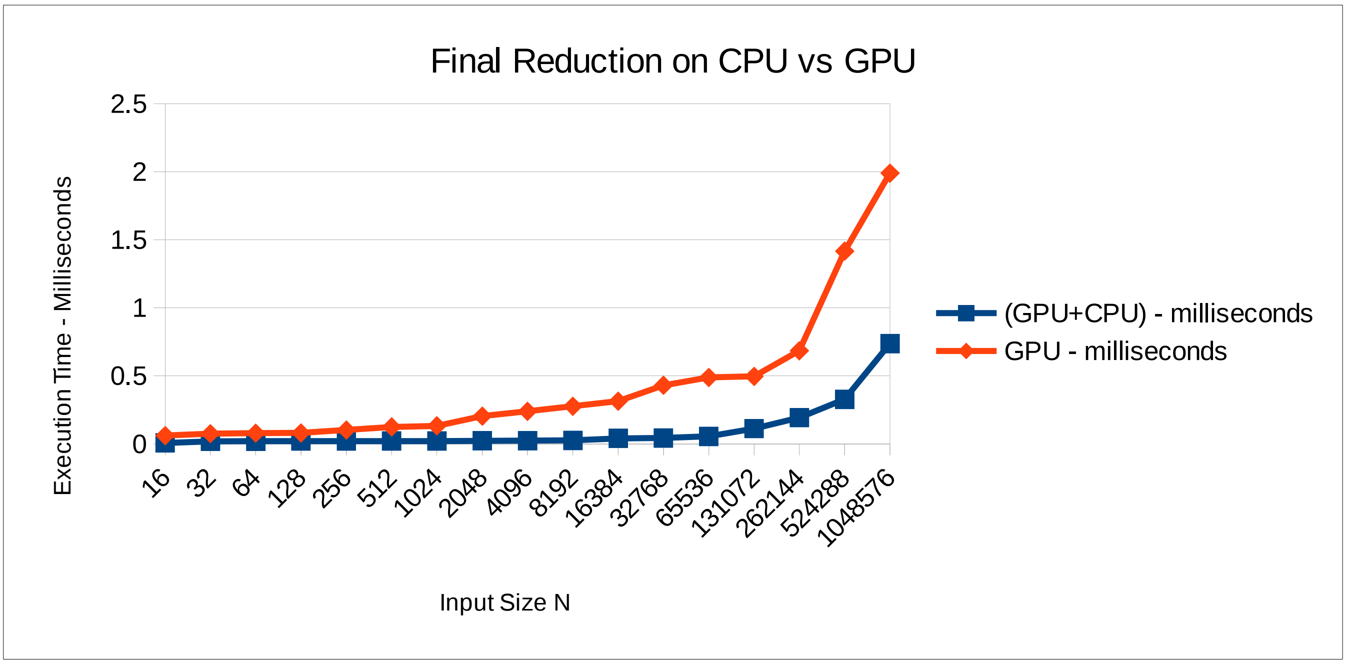
```
            }
            else {
            reduceKernel<<<grid_size, BLOCK_SIZE>>>(d_out, d_out);
            cudaDeviceSynchronize();
            grid_size = grid_size / BLOCK_SIZE;
            }
      }
      // Stop timing on gpu
      cudaEventRecord(stop);
      // Synchronise all timing events and total them
      cudaEventSynchronize(stop);
      // Initialise float to store time
      float milliseconds = 0;
      cudaEventElapsedTime(&milliseconds, start, stop);
      printf("Elapsed time on gpu was: %f milliseconds", milliseconds);
      printf("\n");
      printf("Final GPU reduction: %f\n", d_out[0]);
      // Free the memory allocated on the gpu
      cudaFree(d_in);
      cudaFree(d_out);
}

/**
 * Reduce kernel taking two parameters d_out and d_in
 * d_in takes the input of the kernel, on the first run this is N
 * d_out takes the output of the kernel, on the first run this is N/BLOCK_SIZE
 */
__global__ void reduceKernel(float* d_out, float* d_in) {

      // ID relative to whole array
      int myId = threadIdx.x + blockDim.x * blockIdx.x;
      // Local ID within the current block
      int tid = threadIdx.x;
      //initialisation of shared memory temporary array whose size is equal to
the BLOCK_SIZE
      __shared__ float temp[BLOCK_SIZE];
      //assign the index of tid to the index of d_in at the value my_Id
      temp[tid] = d_in[myId];
      __syncthreads();

      /*
       * Do reduction in shared memory
       * It uses logical shifts to total up values
       * Makes sure that values are greater than or equal to 1
       * Must sync all threads to prevent a race condition
       */
      for (unsigned int s = blockDim.x/2; s >= 1; s >>= 1){
            if (tid < s){
                  temp[tid] += temp[tid + s];
            }
            __syncthreads();
      // make sure all adds at one stage are done!
      }

      // only thread 0 writes result for this block back to global memory
      if (tid == 0){
            d_out[blockIdx.x] = temp[tid];
      }
}
```

2. CPU vs GPU execution times

Block_Size = 32

Input-Size(N)	(GPU+CPU)-milliseconds	GPU - milliseconds
16	0.007625	0.061431
32	0.019075	0.075360
64	0.019885	0.079360
128	0.020298	0.080768
256	0.020471	0.102208
512	0.020786	0.124864
1024	0.021040	0.133168
2048	0.023217	0.203936
4096	0.023788	0.238816
8192	0.025605	0.276608
16384	0.040660	0.313600
32768	0.043028	0.430560
65536	0.055505	0.488416
131072	0.112396	0.496192
262144	0.192947	0.684544
524288	0.327918	1.416992
1048576	0.736908	1.989536

Plot



Explanation of results

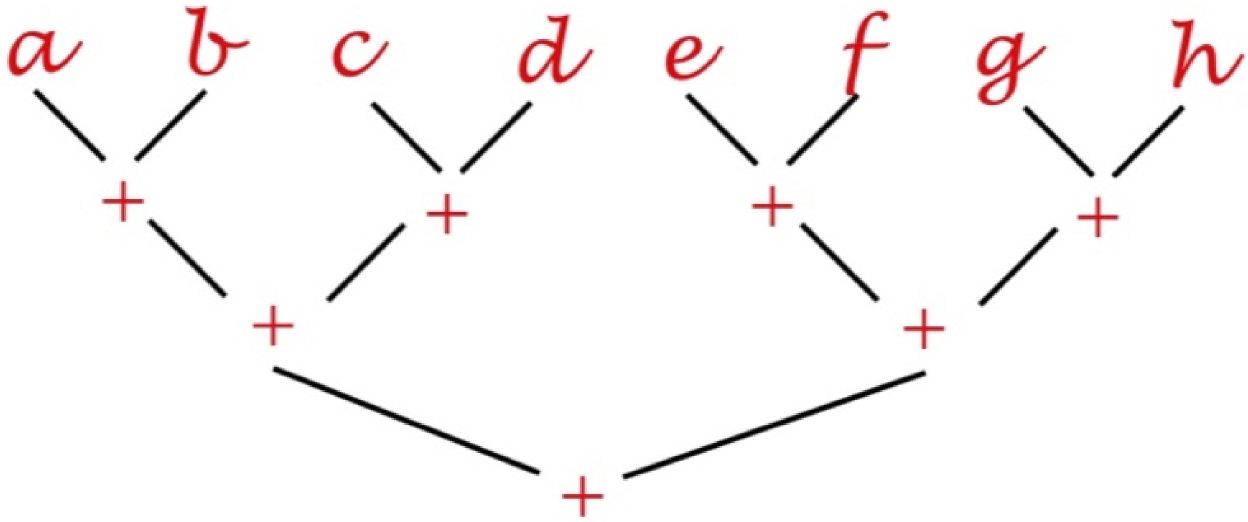
The results show that running the final reduction on the CPU is faster than running it on the GPU.This is an incorrect conclusion as GPU should always be faster a larger array inputs for the reduce algorithm when threads are working in parallel with correct synchronization. After trying to fix my code for hours, I couldn't work out what was wrong. My original solution on the GPU was faster than the CPU, but it did not work for all multiples of a block_size. There was a rounding issue with my grid_sizes when calling the kernel multiple times, and I couldn't find a solution to this issues. Instead, I decided to start a new implementation and make use of cudaMallocManaged. I understand that using cudaMalloManaged will not be faster than cudaMalloc, but it removed the rounding error that I was previously encountering. I Couldn't work out why it was slower than my CPU implementation, but at least it works for all multiples of block_size which in my eyes is a better implementation.

Conclusion

In theory, the GPU version should be faster than the CPU version, my results may not prove this but some extensive research can. The reason for this is due to the ability for the GPU to reuse data. If we synchronize all the threads so that they are executing in parallel, this is going to be a lot more useful than adding up numbers one at a time. Instead of threads waiting on other threads to finish we can synchronize them to pull data at the same time.This means that if we had a more significant input array, multiple blocks with multiple threads would be executing at the same time. Once these blocks have finished, we can merge them to form the final result, and this process is much more efficient than iterating through a for loop one element at a time. Hence the GPU is more efficient than the CPU when it comes to the final reduction.

3. Warp and Thread Divergence

A Warp consists of 32 threads and executes one common instruction at a time, luckily in our case this instruction is just adding up a series of numbers. With the “data collection algorithm” threads within a warp are data-dependent on other threads which in turn causes thread divergence. Looking at the figure below we can see that (a+b) has to wait for (c+d) to finish before it can move onto the next step. Another example of thread divergence is that when the left branch is finished, it needs to wait for the right branch to finish before it can move onto the next step. As the array sizes increase, this is going to become more and more of a problem. We can overcome this problem by using the new “partition” algorithm which splits the input array up into a series of warps. Each warp works on a particular set of inputs, and once the warp is complete, it is merged with other warps. This reduces the total number of steps need and minimises the thread divergence within a warp.



To summarise I would conclude that we used the “repeatedly partitions algorithm” to reduce the time spent dealing with thread divergence and increase the overall efficiency of the program.To support this claim we can look at a quick example using specific input and block sizes. If I had an array of size 512 and a block size of 32, it would take the partition algorithm 2 steps. However, if we look at the data collection algorithm it would take log2(512) which is a total of 9 steps. From just looking at these two numbers we can see a clear performance gain from using the new algorithm.

Part 3: Scan

1. Hillis and Steel implementation

```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#include <time.h>
#include <inttypes.h>
#define BLOCK_SIZE 32 // Block size depicts the number of threads within a block
#define N 1048576  // Size of array

/**
 * Defining the kernel as a global function
 */
__global__ void scanKernel(int n, float *idata);

/**
 * Defining the kernel as a global function
 */
__global__ void mapKernel(float *idata, float *firstScan);

int main(void) {

      // Initialising idata which stores the the array N
      float *idata;
      // Initialising mapScan which stores and manages the
      // intermediate step after the first scan on idata
      float *mapScan;

      /**
      * Initialise start and stop events for timing gpu
       */
      cudaEvent_t start, stop;
      struct timespec begin, end;
      cudaEventCreate(&start);
      cudaEventCreate(&stop);

      // Initialise cudaError variable to check if there our illegal memory
      access issues.
      cudaError_t err;

      //Allocate idata and mapScan into memory that is managed by unified memory
      cudaMallocManaged(&idata, N * sizeof(float));
      cudaMallocManaged(&mapScan, N * sizeof(float));

      //Iterate through N assigning values within idata[i] array 1,1,1,1.....
      for (int i = 0; i < N; i++) {
            idata[i] = 1;
      }

      //grid_size stores the number of blocks needed to execute the kernel on a
      given array size!
      int grid_size = (N / BLOCK_SIZE);
      // We time the first scan on the gpu and total it
      cudaEventRecord(start);

      /**
```

```
 * Call Kernel on given grid_size and number of threadsPerBlock
 * blocks = grid_size = N/BLOCK_SIZE
 * threadPerBlock = BLOCK_SIZE
 */
scanKernel<<<grid_size, BLOCK_SIZE>>>(BLOCK_SIZE, idata);
// Here we total the time it took for the first scan.
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Elapsed time on gpu first scan: %f milliseconds", milliseconds);
printf("\n");

err = cudaDeviceSynchronize();
printf("Run kernel: %s\n", cudaGetErrorString(err));
// Start timing cpu operations
clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &begin);
/*
 * Firstly this process gets the intermediate results for the scan kernel
 * which is the (block_size-1)th element of the array
 * It then rewrites the array with the second intermediate step which
 * involves totalling these values
 * For example if we had an array size N = 8 and a block size of 4
 * map scan would store 4 and 8 as long as the values in the orignal array
 * are all 1
 * We then call the mapping kernel on this new array.
 */
for(int i = 0; i < grid_size; i++){
       mapScan[i] = idata[((i+1)*BLOCK_SIZE)-1];
       if(i != 0){
            mapScan[i] += mapScan[i-1];
       }
}
// Finish timing cpu operations.
clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &end);
uint64_t time = 1e9 * (end.tv_sec - begin.tv_sec) + (end.tv_nsec -
begin.tv_nsec);
/*
 * Converting the time to milliseconds
 * 1 million nano seconds equals 1 millisecond
 * hence time/1million converts it to milliseconds.
 */
float clocktime = (time/1.0e6);
printf("Elapsed time on cpu was: %f milliseconds", clocktime);
printf("\n");
// Start timing on gpu for mapping kernel
cudaEventRecord(start);
/**
 * Call Kernel on given grid_size and number of threadsPerBlock
 * blocks = grid_size = N/BLOCK_SIZE
 * threadPerBlock = BLOCK_SIZE
 */
mapKernel<<<grid_size, BLOCK_SIZE>>>(idata, mapScan);
err = cudaDeviceSynchronize();
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds2 = 0;
cudaEventElapsedTime(&milliseconds2, start, stop);
printf("Elapsed time on gpu map scan: %f milliseconds", milliseconds2);
printf("\n");
```

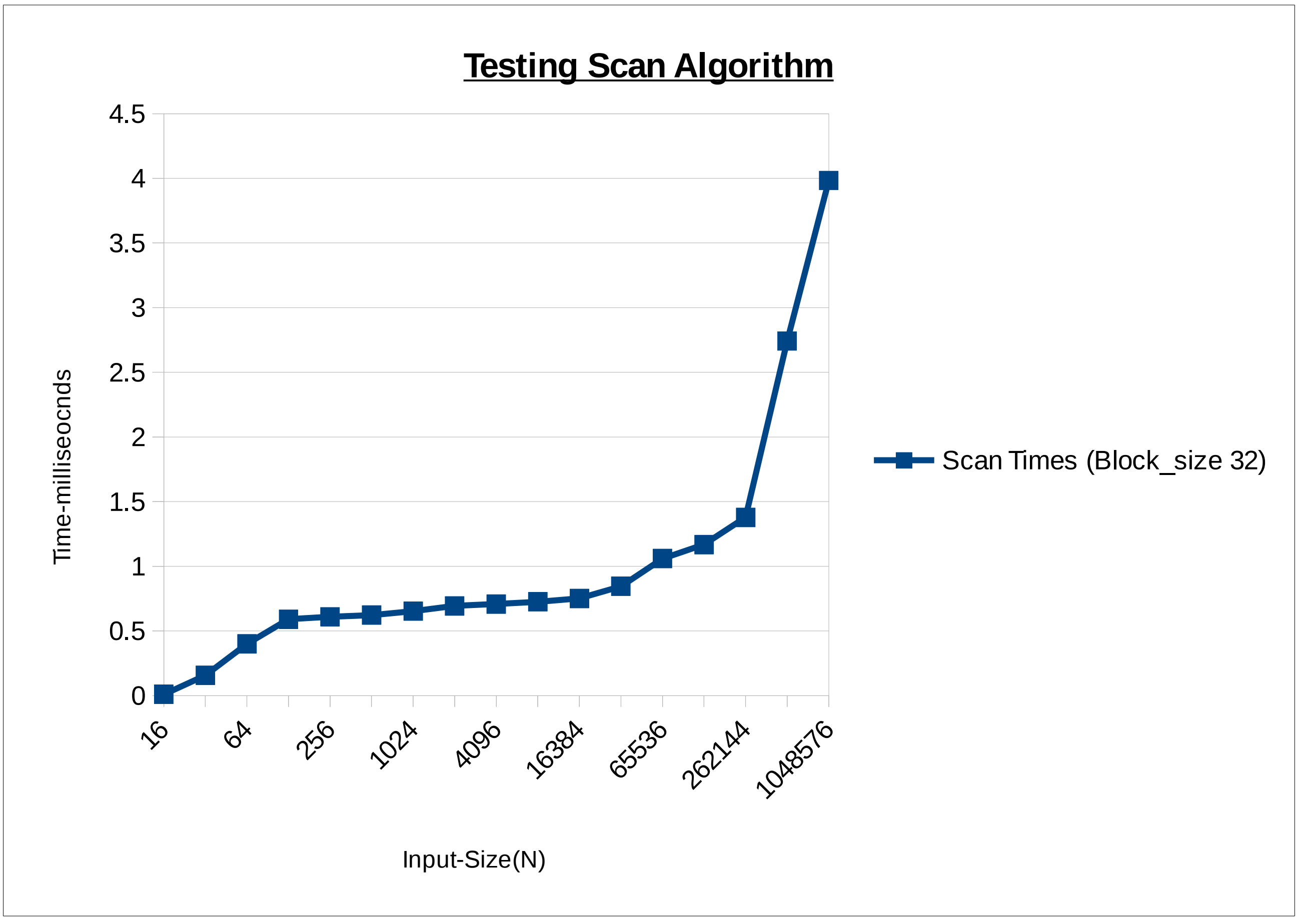
```
        printf("\n");
        //Add to Gpu times and cpu time together and output it
        float overalltime =  clocktime + milliseconds + milliseconds2;
        printf("CPU+GPU-Total Time:%f ", overalltime);
        printf("\n");
        //Output the final result(the last element in the array to test it)
        printf("%f ", idata[N-1]);
        printf("\n");
        //Free the memory
        cudaFree(idata);
        cudaFree(mapScan);
}
/**
 * Scan kernel taking two parameters n and idata
 * It loops through the array taking the (BLOCK_SIZE-1)th Element
 * We have defined two temporary shared memory arrays to prevent a race
condition
 * With the previous kernel you provided there were a few cases where a thread
 * used a number before the previous thread had finished its calculation
 */
__global__ void scanKernel(int n, float *idata) {

        // ID relative to whole array
        int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
        // Local ID within the current block
        int tid = threadIdx.x;

        //initialisation of shared memory temporary array whose size is equal to
        the BLOCK_SIZE
        __shared__ float temp[BLOCK_SIZE];
        //initialisation of shared memory temporary array whose size is equal to
        the BLOCK_SIZE
        __shared__ float temp2[BLOCK_SIZE];

        // declare a boolean to identify which of the two buffers we are currently
        reading from
        bool tempSelector = true;

        // each thread reads one data item into the first buffer in shared memory
        temp[tid] = idata[thIdx];
        __syncthreads();

        /**
         * Do the scan in shared memory
         */
        for (int offset = 1; offset < n; offset *= 2) {
                // Let's also make sure that threads are only working on array
                elements that have data
                if (tid >= offset && thIdx < N) {
                        // for odd loop numbers, read from first buffer into second
                        if (tempSelector)
                        {
                                temp2[tid] = temp[tid] + temp[tid - offset];
                        }
                        // for even loop numbers, read from second buffer into first
                        else
                        {
                                temp[tid] = temp2[tid] + temp2[tid - offset];
                        }
                }
```

```
            // We also need to make sure all the unmodified values are copied
            between the two buffers
            if (tid < offset) {
                  if (tempSelector)
                        temp2[tid] = temp[tid];
                  else
                        temp[tid] = temp2[tid];
            }

            // and update the condition
            tempSelector = !tempSelector;
            // then make sure all threads have finished before going round the
            loop again
            __syncthreads();
      }

      /**
       * If tempSelector is true write temp[tid] to idata
       * If not write temp2[tid] to idata
       * This prevents race conditions
       * We need to make sure we output the value from the correct buffer
       */
      if (tempSelector) {
            idata[thIdx] = temp[tid];
      }
      else{
            idata[thIdx] = temp2[tid];
      }
}

__global__ void mapKernel(float *idata, float *firstScan){
      // ID relative to whole array
      int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
      // Local ID within the current block
      int tid = threadIdx.x;
      //initialisation of shared memory temporary array whose size is equal to
      the BLOCK_SIZE
      __shared__ float temp[BLOCK_SIZE];
      // each thread reads one data item into the first buffer in shared memory
      temp[tid] = idata[thIdx];
      // then make sure all threads have finished
      __syncthreads();

      /*
       * Mapping the mapscan array to the idata array
       * If the block is not the first block we set temps value to be equal
       * to the value of the firstscan block plus the original value of idata
       */
      if (blockIdx.x > 0)
            temp[tid] = firstScan[blockIdx.x-1] + idata[thIdx];
      __syncthreads();
      /*
       * Rewrite idata[thIdx] to be equal to the temp value we just worked out
       * For example if we had an array size N = 8 and a block size of 4
       * map scan would store 4 and 8 assuming the values within idata are 1
       * The mapping kernel would then add 4 to the second block
       * The final output would be 1234,5678.
       */
      idata[thIdx] = temp[tid]; }
```

2. How execution time scales with input size

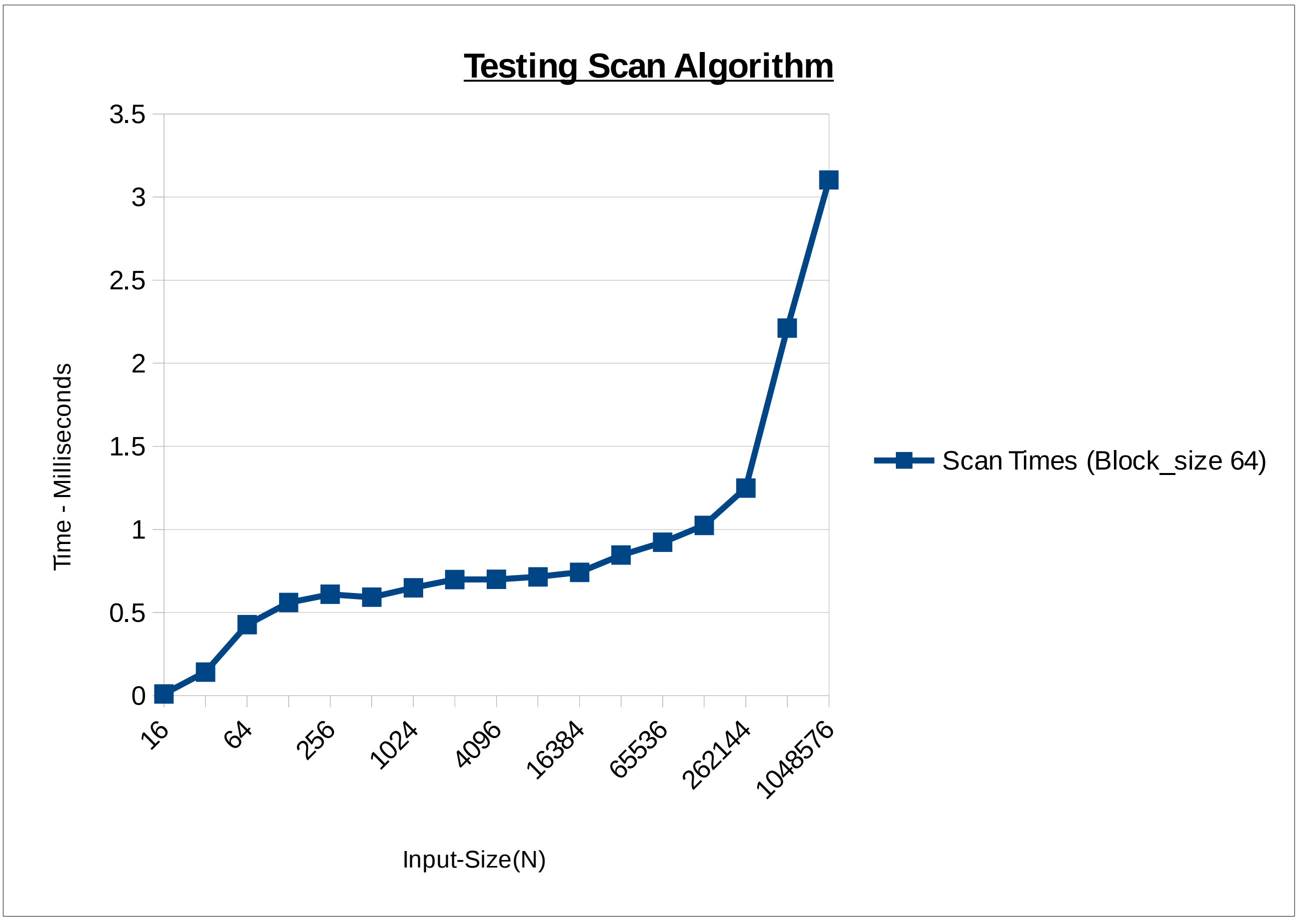
Tested at Block_Size = 32

Input-Size	Time-milliseconds
16	0.010679
32	0.156735
64	0.399944
128	0.590638
256	0.609826
512	0.623423
1024	0.653416
2048	0.693611
4096	0.708639
8192	0.725434
16384	0.751573
32768	0.846241
65536	1.060041
131072	1.168232
262144	1.378488
524288	2.742247
1048576	3.983506

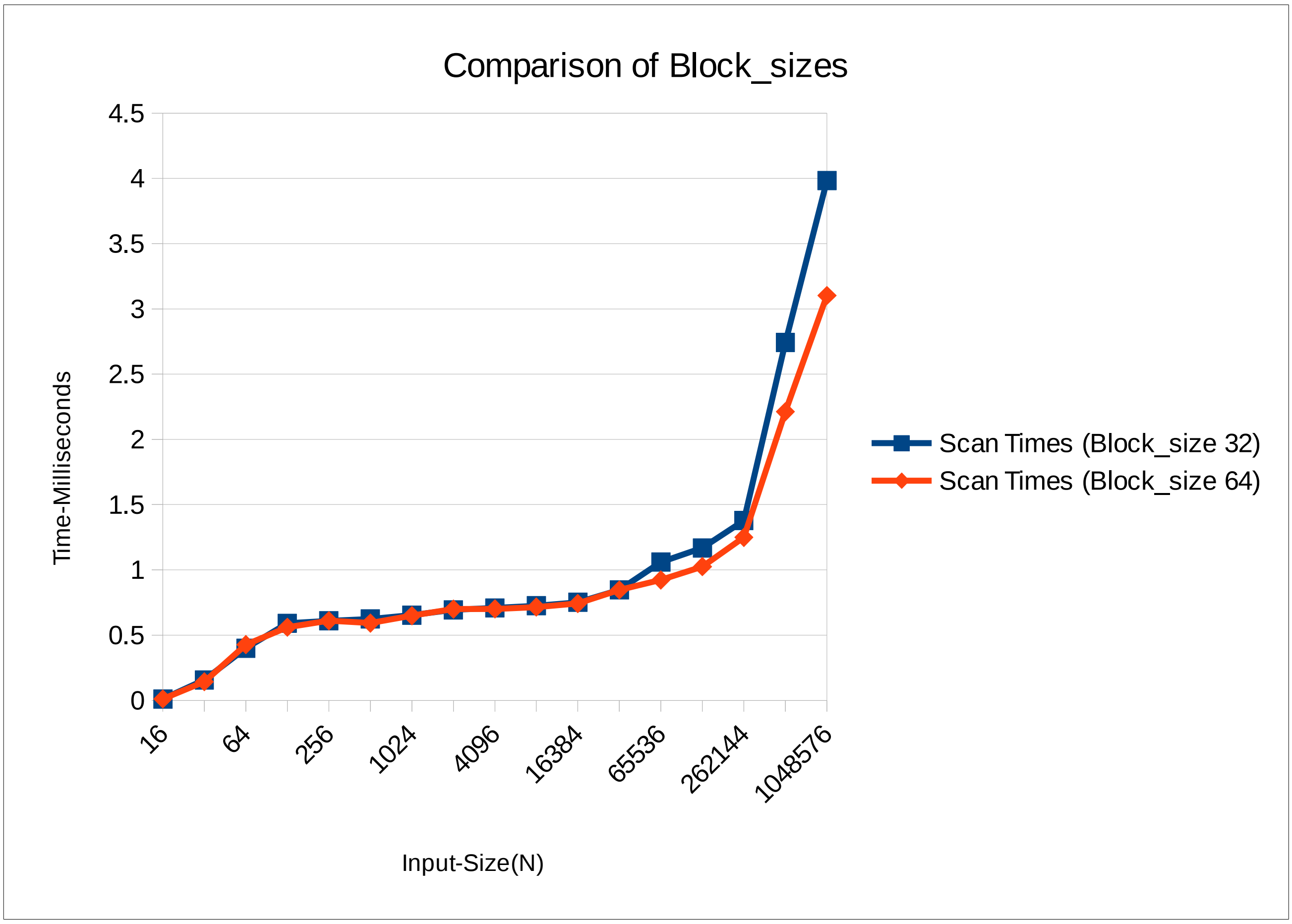


Tested at Block_Size = 64

Input-Size	Time-milliseconds
16	0.011324
32	0.142613
64	0.426912
128	0.560122
256	0.610231
512	0.592502
1024	0.649203
2048	0.699201
4096	0.700421
8192	0.715432
16384	0.742592
32768	0.846241
65536	0.923012
131072	1.025120
262144	1.249312
524288	2.212312
1048576	3.102891



Comparison



I find this graph rather interesting as it indicates that as we increase block_size, there is a performance gain on the GPU which in turn results it in taking a shorter amount of time to process the input array. However, I ran a few tests on larger block sizes and found this was not the case for these specific input sizes. At an input size of 2^20(1048576) and a block_size of 1024, it took “3.24120” milliseconds which is slightly longer than at a block_Size of 64. If we test any block_size above 64 for this range of input values, 64 is still the optimal block_size.This indicates that there is some ratio at play which determines the optimal block_size. If we were to test input sizes greater than 2^20 I'm sure 128 would become the optimal block_size. Unfortunately, the GPU that was provided by the uni will not allow me to test very large values so I cannot confirm this hypothesis. However, if we take a look at how scan works its pretty evident that some sort of ratio exists. For example, if we had an input size of 512 and a block_size of 256, we would only have two blocks working on the input array at any time. However, if we used a block_size of 16, we would have 32 blocks working in parallel which would reduce the time it takes to iterate over all the values as threads wouldn’t be waiting for other threads to finish. I think it would be interesting to work out the exact cut off point where the block_size should increase to match the input size increase!

3. Suggestions for improving code

Improvement 1:

In my scan implementation, I do the intermediate step on the CPU instead of on the GPU. This for loop collects the last element from each block and stores it in a new array (mapScan). As we are storing these new values, we make the next amount equal to the previous value plus the new one.For example, if we had (4,4,4,4) map scan would store (4,8,12,16). We can see from the code below how this works

```
for(int i = 0; i < grid_size; i++){
        mapScan[i] = idata[((i+1)*BLOCK_SIZE)-1];
        if(i != 0){
                mapScan[i] += mapScan[i-1];
        }
}
```

We could improve this implementation by calling the scan kernel again straight after the first call on map scan instead of looping through and adding up all the values like this:

```
scanKernel<<<grid_size, BLOCK_SIZE>>>(grid_size, mapScan);
```

This is similar to how reduce works on the GPU, and it would mean we don't need to loop through all the values adding them up one by one. Instead, we rely on the scan kernel to do this for us which means we're working on the GPU and not the CPU. This allows the threads to reuse the data when executing in parallel. Instead of waiting for the previous calculation to finish the threads can all finish at the same time, this will be a lot more effective than the current implementation.

Improvement 2:

My second improvement is to make use of “CudaMalloc and CudaMemcpy” instead of using “CudaMallocManaged.” “Cudamalloc” is a lot more challenging to implement as we have to explicitly specify host and device memory allocation, however it has its benefits. For example “cudaMallocManaged” fails once you allocate more memory than what is available on the device. Most importantly when running multiple scan calls, we work out the amount of memory that needs to be assigned to the device instead of wasting allocated unwanted memory. By mapping specific memory sizes, we will see a clear performance gain hence improving my scan implementation

Part 4: Histogram

1. Parallel Implementation using kernel outlined in lecture 11

```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define BLOCK_SIZE 64 // Block size depicts the number of threads within a block

/**
 * Defining the kernel as a global function
 */
__global__ void simple_histogram(int *d_bins, const int *d_in, const int
BIN_COUNT);

/**
 * Main method which invokes calls on the kernel
 * Outputs the histogram
 */
int main(void) {
      // Size of array
      int N = 1048576;
      // The number of bins used to output the histogram
      int *d_bins;
      // Initialise a variable which stores the values of the array
      int *d_in;

      /**
       * Initialise start and stop events for timing gpu
       */
      cudaEvent_t start, stop;
      cudaEventCreate(&start);
      cudaEventCreate(&stop);

      // This sets the number of bins in the histogram
      int BIN_COUNT = 8;
      // We will use the CUDA unified memory model to ensure data is transferred
      between host and device
      // We times both values by (*sizeof(int)) as integers require between 1
      byte and as much as 8 bytes depending on the value of the integer
      cudaMallocManaged(&d_bins, BIN_COUNT*sizeof(int));
      cudaMallocManaged(&d_in, N*sizeof(int));

      // Now we need to generate some input data, just 1's in this case for
      simplicity and checking it works
      for (int i=0; i < N; i++)
      {
            d_in[i] = i;
      }

      // We also need to initialise the bins in the histogram
      for (int i=0; i < BIN_COUNT; i++)
      {
            d_bins[i] = 0;
      }

      // Now we need to set up the grid size
```

```
	int grid_size = N/BLOCK_SIZE;
	// Start timing the gpu
	cudaEventRecord(start);

	/**
	 * Call the simple histogram kernel
	 * Takes in the parameters d_bins, d_in, BIN_COUNT, see above for
	 * decleration meaning
	 */
	simple_histogram<<<grid_size, BLOCK_SIZE>>>(d_bins, d_in, BIN_COUNT);
	// wait for Device to finish before accessing data on the host
	// Finalising and outputing timing using simple histogram
	cudaDeviceSynchronize();
	cudaEventRecord(stop);
	cudaEventSynchronize(stop);
	// initialise variable to store time..
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, start, stop);
	printf("Elapsed time on gpu was: %f milliseconds", milliseconds);
	printf("\n");
	// Now we can print out the resulting histogram
	/**
	 * This method checks that the kernel call was a success
	 * Depending on the BIN count it should output (N/BINCOUNT) for each bin
	 * For example if we had an array size of 1048576 and 8 bins,
	 * Each bin should output a value of 131072 (131072*8 == 1048576)
	 */
	for (int i = 0; i < BIN_COUNT; i++)
	{
		printf("Bin no. %d: Count = %d\n", i, d_bins[i]);
	}
}
/*
 * Simple_histogram takes in three parameters
 * The first one d_bins is number of bins used(in this case 8) *sizeOf(int)
 * d_in is the array containing the values up to 1048576(all 1's)
 * BIN_COUNT is the number of bins used
 */
__global__ void simple_histogram(int *d_bins, const int *d_in, const int
BIN_COUNT)
{
	// ID relative to whole array
	int myId = threadIdx.x +blockDim.x * blockIdx.x;
	//assign the myitem to the index of d_in at the value my_Id
	int myItem = d_in[myId];
	// myBin i equal to the index of d_in at the value my_Id remainder
	BIN_COUNT
	int myBin = myItem % BIN_COUNT;
	/**
	 * Calling atomicAdd() on global memory
	 * This is slower than shared memory as only one thread can
	 * update a specific bin at a specific time so there is very little
	concurrency
	 * To improve this we can call atomic add on a local histogram in shared
	memory(see next implementation)
	*/
	atomicAdd(&(d_bins[myBin]), 1);
}
```

2. Implementation using local memory per block

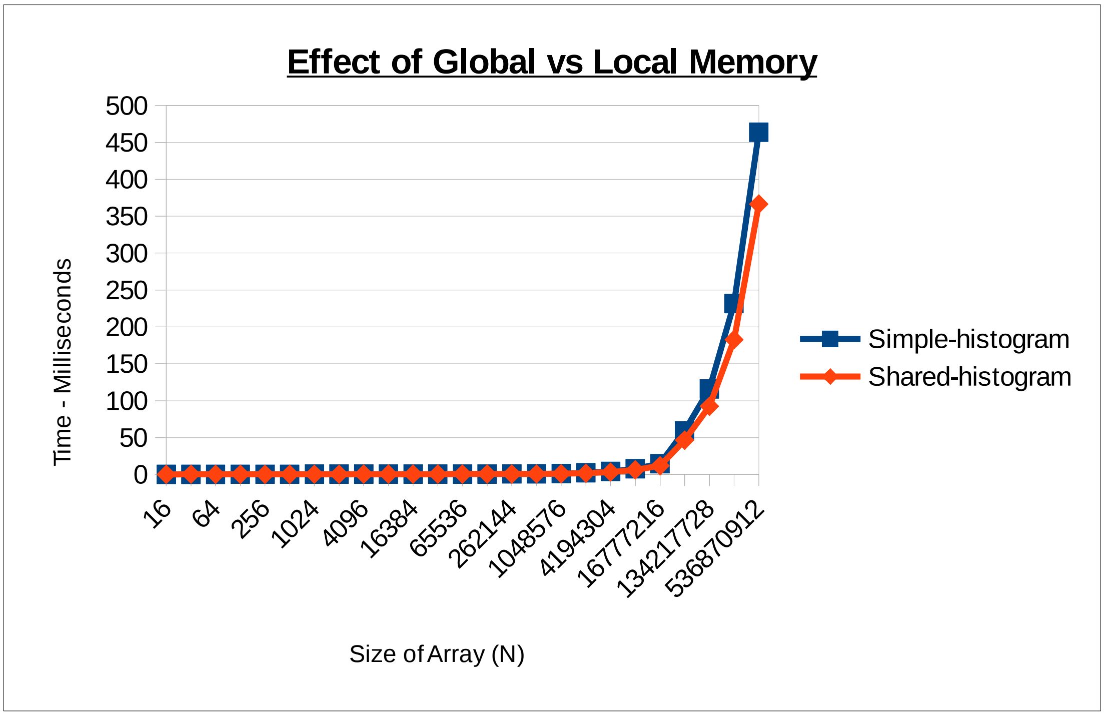
```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define BLOCK_SIZE 32 // Block size depicts the number of threads within a block
#define B_COUNT 8 // Define the number of bins used

/**
 * Defining the kernel as a global function
 */
__global__ void shared_memory_histogram(int *d_bins, const int *d_in, const int
BIN_COUNT);
int main(void)
{
// This is the size of the input array
int N = 1048576;
// The number of bins used to output the histogram
int *d_bins;
// Initialise a variable which stores the values of the array
int *d_in;

/**
* Initialise start and stop events for timing gpu
*/
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// This sets the number of bins in the histogram
int BIN_COUNT = B_COUNT;
// We will use the CUDA unified memory model to ensure data is transferred
between host and device
// We times both values by (*sizeof(int)) as integers require between 1 byte and
as much as 8 bytes depending on the value of the integer
cudaMallocManaged(&d_bins, BIN_COUNT*sizeof(int));
cudaMallocManaged(&d_in, N*sizeof(int));

// Now we need to generate some input data, just 1's in this case for simplicity
and checking it works
for (int i=0; i < N; i++)
{
d_in[i] = i;
}
// We also need to initialise the bins in the histogram
for (int i=0; i < BIN_COUNT; i++)
{
d_bins[i] = 0;
}
// Now we need to set up the grid size
int grid_size = N/BLOCK_SIZE;

// Start timing the gpu
cudaEventRecord(start);

/**
 * Call the simple histogram kernel
 * Takes in the parameters d_bins, d_in, BIN_COUNT, see above for decleration
meaning
```

```
 */
shared_memory_histogram<<<grid_size, BLOCK_SIZE>>>(d_bins, d_in, BIN_COUNT);
// wait for Device to finish before accessing data on the host
// Finalising and outputing timing using simple histogram
cudaDeviceSynchronize();
cudaEventRecord(stop);
cudaEventSynchronize(stop);
// initialise variable to store time..
float milliseconds2 = 0;
cudaEventElapsedTime(&milliseconds2, start, stop);
printf("Elapsed time on shared was: %f milliseconds", milliseconds2);
      printf("\n");
// Now we can print out the resulting histogram

/**
 * This method checks that the kernel call was a success
 * Depending on the BIN count it should output (N/BINCOUNT) for each bin
 * For example if we had an array size of 1048576 and 8 bins,
 * Each bin should output a value of 131072 (131072*8 == 1048576)
 */
for (int i = 0; i < BIN_COUNT; i++)
{
printf("Bin no. %d: Count = %d\n", i, d_bins[i]);
}
}
/*
 * Simple_histogram takes in three parameters
 * The first one d_bins is number of bins used(in this case 8) *sizeOf(int)
 * d_in is the array containing the values up to 1048576(all 1's)
 * BIN_COUNT is the number of bins used
 */
__global__ void shared_memory_histogram(int *d_bins, const int *d_in, const int
BIN_COUNT){
      // ID relative to whole array
      int myId = threadIdx.x +blockDim.x * blockIdx.x;
      //assign the myitem to the index of d_in at the value my_Id
      int myItem = d_in[myId];
      // myBin i equal to the index of d_in at the value my_Id remainder
      BIN_COUNT
      int myBin = myItem % BIN_COUNT;
      __shared__ int sharedHisto[B_COUNT];
      sharedHisto[threadIdx.x]=0;
      // now change this line to call atomicAdd on shared memory
      atomicAdd(&sharedHisto[myBin], 1);
      __syncthreads();
      /**
       * Calling atomicAdd() on shared memory
       * Checks that the thread count is always less than bin count
       * This prevents wasting time in a loop going over values that do not
       * matter
       * Reduces resoure ontention on attomic adds if there are a larger number
       * of blocks
       * Latency on shared memory is much lower than on global memory
       */
      if(threadIdx.x < B_COUNT){
                  atomicAdd(&(d_bins[threadIdx.x]), sharedHisto[threadIdx.x]);
    }

      __syncthreads(); }
```

3. Showing how execution time scales with input size

Input-Size(N) Block_Size = 32	Global Histogram - Milliseconds	Shared Histogram -milliseconds
16	0.009376	0.008416
32	0.242944	0.352448
64	0.256640	0.127680
128	0.341280	0.187320
256	0.355008	0.214320
512	0.372313	0.268864
1024	0.411104	0.335808
2048	0.419360	0.395808
4096	0.424912	0.424173
8192	0.446722	0.434121
16384	0.489201	0.445386
32768	0.510235	0.499935
65536	0.561231	0.535328
131072	0.598213	0.567072
262144	0.623111	0.593491
524288	0.818112	0.628064
1048576	1.226400	0.985856
2097152	2.161696	1.713536
4194304	4.062880	3.177120
8388608	7.631104	6.221280
16777216	14.686272	11.707232
67108864	59.127838	46.544830
134217728	115.893539	92.379425
268435456	231.637222	182.782211
536870912	463.763733	366.320831

Plot



From the graph above we can see that as input sizes increase past 1 million, we can start to see that the shared memory implementation has a performance gain over the global histogram. However, at smaller input sizes there is a very little difference between the local and global memory implementation. By creating a local version of the histogram in shared memory in each block and then calling atomic add on it, we can increase the performance rate. By using shared memory, we have a more significant number of blocks available which reduces the length of time wasted with resource contention on atomic adds. The latency of atomic add is significantly lower on shared memory than on global memory. Latency is a fixed value which depends on which memory you're accessing (Quicker in shared memory).With having local histograms stored in shared memory, each block constructs a local histogram and adds up the values using atomics, when all blocks are finished they are merged to form the final result. The diagram below shows an example of how this would work with a given range of numbers on three blocks. What I have concluded from this is that until you reach huge inputs, there isn't much point implementing the histogram on shared memory as the performance gain is so small!
I did a little research and found out that on most architectures global atomics perform better than shared memory atomics. However, with Maxwell architecture shared memory atomics have a better performance rate.