URN: 6477799

COM2039: Parallel Programming Coursework

As per request, I confirm that the work stated below is purely my work, and no text stated below is copied, facts and images that are used for illustrative purposes are appropriately referenced. NOTE: All source code is attached alongside this document, under the surrey learn submission.

Username: jj00435

Parallel Computing – CUDA Programming Coursework

Matrix Multiplication

Q1.1) Example inputs and outputs

Lab2:

Matrix A values:

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

Matrix B Values:

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

Matrix A * Matrix B:

```
7.000000 11.000000 9.000000 15.000000 14.000000 10.000000 7.000000 8.000000 11.000000 11.000000 8.000000 12.000000 13.000000 8.000000 5.000000 13.000000
4.000000 4.000000 3.000000 3.000000 6.000000 3.000000 1.000000 5.000000 6.000000 5.000000 3.000000 2.000000 6.000000 3.000000 2.000000 5.000000
5.000000 9.000000 6.000000 12.000000 13.000000 7.000000 7.000000 8.000000 7.000000 5.000000 6.000000 7.000000 7.000000 6.000000 5.000000 8.000000
7.000000 9.000000 7.000000 6.000000 9.000000 4.000000 7.000000 7.000000 7.000000 8.000000 7.000000 7.000000 10.000000 3.000000 2.000000 8.000000
5.000000 6.000000 7.000000 5.000000 9.000000 3.000000 4.000000 6.000000 8.000000 8.000000 8.000000 7.000000 6.000000 5.000000 3.000000 6.000000
5.000000 12.000000 7.000000 10.000000 12.000000 4.000000 6.000000 8.000000 6.000000 9.000000 4.000000 7.000000 9.000000 7.000000 6.000000 11.000000
6.000000 7.000000 7.000000 6.000000 6.000000 5.000000 4.000000 6.000000 7.000000 8.000000 5.000000 6.000000 7.000000 5.000000 3.000000 6.000000
8.000000 13.000000 7.000000 11.000000 12.000000 4.000000 5.000000 10.000000 10.000000 11.000000 6.000000 8.000000 12.000000 6.000000 4.000000 9.000000
3.000000 5.000000 3.000000 9.000000 11.000000 3.000000 4.000000 7.000000 5.000000 5.000000 4.000000 4.000000 4.000000 5.000000 4.000000 4.000000
6.000000 9.000000 9.000000 10.000000 14.000000 7.000000 8.000000 8.000000 11.000000 8.000000 10.000000 11.000000 9.000000 7.000000 5.000000 10.000000
6.000000 11.000000 6.000000 11.000000 12.000000 4.000000 5.000000 7.000000 6.000000 8.000000 7.000000 6.000000 6.000000 5.000000 5.000000 6.000000
6.000000 8.000000 7.000000 7.000000 10.000000 4.000000 5.000000 5.000000 6.000000 6.000000 6.000000 5.000000 3.000000 6.000000 6.000000 6.000000
9.000000 10.000000 7.000000 10.000000 12.000000 5.000000 5.000000 11.000000 12.000000 11.000000 7.000000 7.000000 12.000000 6.000000 4.000000 8.000000
6.000000 11.000000 8.000000 10.000000 15.000000 5.000000 7.000000 9.000000 10.000000 9.000000 9.000000 9.000000 8.000000 6.000000 10.000000
4.000000 6.000000 1.000000 8.000000 10.000000 5.000000 1.000000 7.000000 8.000000 9.000000 3.000000 4.000000 9.000000 5.000000 2.000000 8.000000
8.000000 8.000000 8.000000 5.000000 8.000000 3.000000 5.000000 7.000000 7.000000 7.000000 7.000000 6.000000 7.000000 4.000000 4.000000 6.000000
```

Lab3:

Matrix A values:

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

Matrix B Values:

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

Matrix A * Matrix B:

```
9.000000 11.000000 6.000000 6.000000 9.000000 12.000000 3.000000 11.000000 9.000000 6.000000 13.000000 10.000000 4.000000 6.000000 5.000000 9.000000
7.000000 10.000000 6.000000 10.000000 8.000000 11.000000 2.000000 9.000000 10.000000 4.000000 11.000000 11.000000 7.000000 8.000000 9.000000 7.000000
8.000000 10.000000 5.000000 7.000000 11.000000 11.000000 6.000000 13.000000 8.000000 4.000000 14.000000 9.000000 5.000000 10.000000 9.000000 8.000000
5.000000 10.000000 7.000000 5.000000 6.000000 8.000000 7.000000 11.000000 9.000000 2.000000 11.000000 7.000000 10.000000 7.000000 7.000000 10.000000
4.000000 9.000000 3.000000 9.000000 8.000000 7.000000 2.000000 6.000000 8.000000 4.000000 10.000000 6.000000 8.000000 8.000000 7.000000 10.000000
7.000000 10.000000 6.000000 5.000000 9.000000 8.000000 5.000000 7.000000 8.000000 3.000000 10.000000 8.000000 8.000000 6.000000 5.000000 10.000000
8.000000 14.000000 9.000000 12.000000 10.000000 12.000000 5.000000 9.000000 11.000000 6.000000 11.000000 14.000000 8.000000 7.000000 10.000000 9.000000
6.000000 14.000000 8.000000 11.000000 10.000000 12.000000 4.000000 11.000000 11.000000 5.000000 14.000000 10.000000 6.000000 11.000000 11.000000 10.000000
5.000000 5.000000 2.000000 6.000000 4.000000 7.000000 0.000000 6.000000 7.000000 2.000000 9.000000 5.000000 5.000000 4.000000 6.000000 5.000000
4.000000 12.000000 6.000000 6.000000 7.000000 8.000000 4.000000 11.000000 7.000000 5.000000 11.000000 11.000000 7.000000 9.000000 6.000000 7.000000
5.000000 10.000000 6.000000 7.000000 9.000000 11.000000 6.000000 11.000000 10.000000 4.000000 12.000000 12.000000 8.000000 8.000000 9.000000 9.000000
7.000000 14.000000 8.000000 7.000000 9.000000 9.000000 8.000000 12.000000 8.000000 4.000000 13.000000 11.000000 10.000000 8.000000 9.000000 9.000000
5.000000 11.000000 7.000000 7.000000 8.000000 9.000000 7.000000 12.000000 10.000000 3.000000 12.000000 9.000000 9.000000 10.000000 9.000000 9.000000
7.000000 8.000000 2.000000 9.000000 8.000000 8.000000 1.000000 6.000000 8.000000 3.000000 11.000000 7.000000 8.000000 8.000000 7.000000 8.000000
4.000000 11.000000 6.000000 8.000000 5.000000 7.000000 1.000000 7.000000 9.000000 5.000000 9.000000 9.000000 4.000000 6.000000 7.000000 7.000000
8.000000 9.000000 5.000000 9.000000 8.000000 10.000000 4.000000 8.000000 9.000000 4.000000 10.000000 12.000000 8.000000 5.000000 9.000000 6.000000
```

Both results were verified from an external website to ensure validity of calculation, as well as initialising the input matrix values to equal 1, and ensuring that the values of the output matrix equals the matrix size (I.e here 16.0).

1.2) Appropriate timing methodology:

Lab2 kernel timing:

```
//Cuda timing functionaltiy
//Initializing appropriate objects for timing.
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);//Records time before kernel launch
MatrixMultKern<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);//Kernel call
cudaDeviceSynchronize();
cudaEventRecord(stop);//Recording the time kernel finished
//Calculating the execution time
float executionTime = 0;
cudaEventElapsedTime(&executionTime, start, stop);
printf("Elapsed time was: %f\n ms", executionTime);
```

Lab3 kernel timing:

```
//Cuda timing functionaltiy
//Initializing appropriate objects for timing.
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);//Records time before kernel launch
MultSharedKern<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);//Kernel call
cudaDeviceSynchronize();
cudaEventRecord(stop);//Recording the time kernel finished
//Calculating the execution time
float executionTime = 0;
cudaEventElapsedTime(&executionTime, start, stop);
printf("Elapsed time was: %f\n ms", executionTime);
```

1.3) Appropriate timing for various sizes for dimensions of matrix

The below values represent the appropriate timings of the execution of the multiplication kernels, using both shared memory and without using shared memory; where the dimensions of the matrix increase in multiples of 16 such that 16*(2^x), where x is from 0 to 9. The timings are measures in milliseconds.

	Execution Time (ms)		Difference
Number of Elements	No shared memory	Shared Memory	NoShared - With Shared
16	0.008192	0.014336	-0.006144
32	0.022528	0.009216	0.013312
64	0.021504	0.011264	0.01024
128	0.034816	0.01536	0.019456
256	0.118112	0.083328	0.034784
512	0.778176	0.52064	0.257536
1024	6.479136	4.099296	2.37984
2048	51.553696	32.551361	19.002335
4096	454.816498	247.103546	207.712952
8192	4953.724121	1589.480591	3364.24353

With the above results we can generate the following graphs:

Figure 1 – (Comparing matrix multiplication for number of elements N, with respect to shared memory):

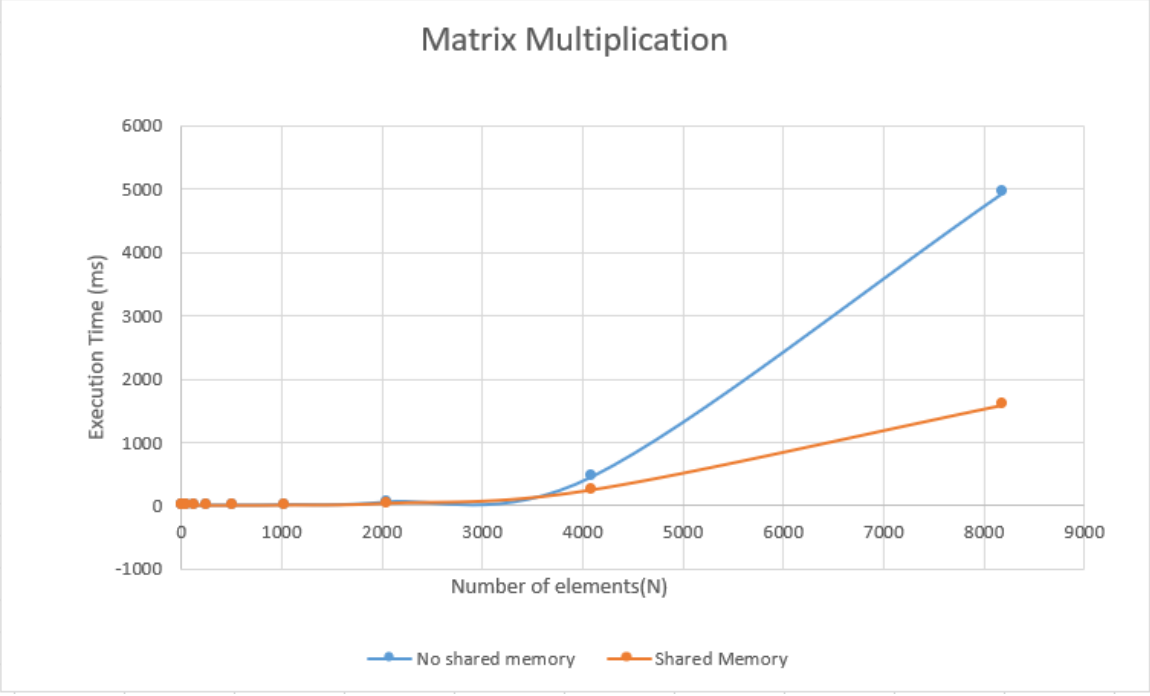


Figure 2- (More granular results):

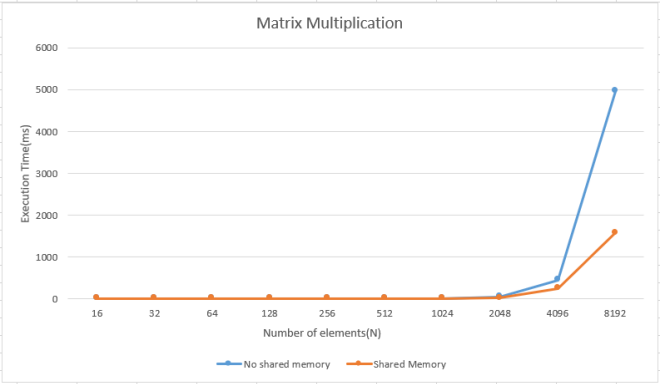
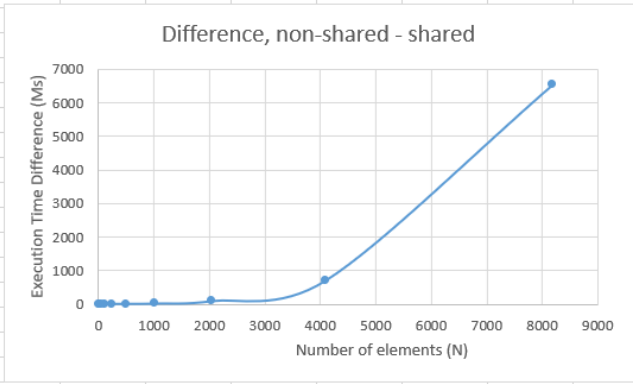


Figure 3 - (Difference: Non-Shared – Shared)



1.4) Analysis of the execution times of shared memory matrixes vs non-shared memory matrixes

Hypothesis: "Execution times of matrix multiplication will be quicker using shared memory vs using just global memory, as the number of elements N increases"

Conclusion:
The results in fact support this hypothesis, where the results suggest that by using shared memory we see a big decrease in execution times than in comparison to just using global memory, for instance as is evident from the figures 1 and 2, the execution times are significantly higher when not using shared memory than when using shared memory. This is further evident form the figure 3, which suggests that as the number of elements increase the difference in execution time similarly increases. We can further say that the mean execution time of not using shared memory is 2.18x that of using shared memory, where we see that as the number of elements increase this factor increases accordingly.

CUDA devices have varying levels of memory, some of which can be controlled by the programmer, and some of which are managed by the CUDA device. For instance, when a global memory call is made the CUDA device first checks L1 cache, then L2 cache for the given data element before accessing global memory. Where values in L1 and L2 cache are managed by the system itself using temporal and special locality. This flow of data lookup is needed as global memory has incredibly slow read time than in comparison with shared, L1 or L2 cache. We can assume that shared memory is 100X faster than global memory access.[1] Shared memory therefore essentially acts as L1 cache, as such by storing elements in shared memory, we can see a significant improvement in calculation time for the kernel than not using shared memory. We can see this concept further in relation to arithmetic intensity, where in by to increase arithmetic intensity we need to either increase the number of computations per thread or decrease the number of global memory access. Due to the nature of matrix multiplication, we cannot decrease/increase the number of needed calculations, therefor the only way to ensure increased execution time is to decrease the global memory reads/writes, which can only be done by substituting the reads from global memory to shared memory. To improve the efficiency, we need to further use the concept of tiled matrix multiplication, where in utilise the principles of matrix multiplications to efficiently use shared memory; which is further explained in section 5.1.

Reduce

2.1) Reduce implemented using GPU, where the final reduction is done using the GPU and CPU

The implementation shown below first calculates a first pass of the reduction by using the GPU, the subsequent final reduction is first done by CPU (without modifying the array of values from previous reduction stage), and then done by purely the GPU. This approach allows us to model the effect of using the GPU and CPU for the final stage of the reduction.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <iostream>
#include <time.h>
using namespace std;

#define BLOCK_SIZE 32

void cpuReduction(unsigned int *arr, int size, float gpuTime);
void printArray(unsigned int *arr, int sizeOfArray);
__global__ void reduceKernel(unsigned int * d_in,unsigned int* d_out,const unsigned int N);
void mainReductionMethod(unsigned int N, bool debug);

int main(void){
    //Allows the looping through of differnt reduction kernel inputs for analysis
    for(int x=1;x<33;x++){
        mainReductionMethod(1<<x,false);
    }
    return 0;
}


void mainReductionMethod(unsigned int N, bool debug){
    // Developed appropraite timing functionality
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float totalExicutionTime=0.0f;

    //Initilaise the value of the array
    unsigned int *input;
    cudaMallocManaged(&input, N * sizeof(unsigned int));
    for(unsigned int x=0;x<N;x++){
        input[x]=1;
    }

    //Initialise the output array values
    unsigned int *output;
    cudaMallocManaged(&output, N * sizeof(unsigned int));
    cudaMemset(output, 0, sizeof(unsigned int) * N); // Initialise the values in the array to 0

    //Below calculation ensures that the correct grid size is calculated
    unsigned int grid_size =  (N+BLOCK_SIZE -1)/BLOCK_SIZE;

    //Calling the reduce kernel once
    cudaEventRecord(start);
    reduceKernel<<<grid_size,BLOCK_SIZE>>>(input,output,N);
    cudaEventRecord(stop);
    cudaDeviceSynchronize();
    cudaEventElapsedTime(&totalExicutionTime, start, stop);

    //Insert here method for CPU final reduce
    cpuReduction(output,grid_size,totalExicutionTime);


    //Method for calling GPU reduction, the set of values reduces every loop, till the values
can fit in
    //a singular block, of size BLOCK_SIZE
    float temp=0.0f; //Temp variable for execution time
    while(grid_size>1){
        //Starts kernel as well as ensuring that the appropraite kernel timing takes place
        cudaEventRecord(start);
        reduceKernel<<<grid_size,BLOCK_SIZE>>>(output,output,grid_size);
        cudaDeviceSynchronize();
        cudaEventRecord(stop);
        cudaEventElapsedTime(&temp, start, stop);
        totalExicutionTime+=temp;

        //Decreases the grid size to reflect the number of elements that need to be further
reduced
        grid_size =  (grid_size+BLOCK_SIZE -1)/BLOCK_SIZE;

    }
    //Appropraite debugging method
    if(debug){
        printArray(output,N);
    }

    //Prints results of reduction to the user
    printf("\nGPU Total: %d Time:%f \n\n",output[0],totalExicutionTime);
    // printf("%f\n",totalExicutionTime);

    //Free approrpaite memory that was used
    cudaFree(input);
    cudaFree(output);

}
```

```
86
87 //Method for calculating the CPU final reduciton
88 void cpuReduction(unsigned int *arr, int size, float gpuTime)
89 {
90     // Appropraite method for timing
91     unsigned int final_reduction = 0;
92     clock_t start = clock();
93     clock_t tStart = clock();
94
95     for (int i = 0; i < size; i++)
96     {
97         final_reduction += arr[i];
98     }
99     // Recordingend time.
100    //Calculating the actual total difference
101    clock_t stop = clock();
102    double time_taken = (double)(stop - start) * 1000.0 / CLOCKS_PER_SEC;
103
104    // Print statments if further debugging is needed
105    // printf("CPU: Summation %d ", final_reduction);
106    // printf("Time %f ms", time_taken + double(gpuTime));
107    // printf("%f\n",time_taken + double(gpuTime));
108
109    printf("CPU: Summation %f , Time: %f ms\n", final_reduction,time_taken + double(gpuTime));
110 }
111
112 //Easy method for printing the contents of the matrix to the screen
113 void printArray(unsigned int *arr, int sizeOfArray)
114 {
115         printf("Printing %d values \n", sizeOfArray);
116         for (int i = 0; i < sizeOfArray; i++)
117         {
118             printf("%d|", arr[i]);
119             if((i+1)%BLOCK_SIZE==0){
120                 printf("\t");
121             }
122         }
123         printf("\n Fin printing");
124     printf("\n");
125 }
126
127
128 //Reduce kernel for device that allows the reduction to take place
129 __global__ void reduceKernel(unsigned int * d_in,unsigned int* d_out,const unsigned int N) {
130     int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
131     int tid = threadIdx.x; // Local ID within the current block
132     __shared__ unsigned int temp[BLOCK_SIZE];
133     temp[tid] = d_in[myId];
134     __syncthreads();
135     // do reduction in shared memory
136     for (unsigned int s = blockDim.x/2; s >= 1; s >>= 1)
137     {
138         if (tid < s && myId<N)
139         {
140         temp[tid] += temp[tid + s];
141         }
142     __syncthreads(); // make sure all adds at one stage are done!
143     }
144     // only thread 0 writes result for this block back to global memory
145     if (tid == 0)
146     {
147         d_out[blockIdx.x] = temp[tid];
148     }
149 }
```

Note: Lines 54 represents the CPU final reduce function, which calls the appropriate method outlined form lines 88 to 110. The final reduction on GPU as outlined by comments, can be found

from lines 59 – 76, which functions by calling the reduce kernel successively, until no further reduction can be made, as shown by commented while loop from lines 59-76.

Further: The same implementation outlined above is done using recursion, and is attached below in the Appendix 1, as well as attached alongside this documentation, which received the same result as a non-recursive solution, shown above.

2.2) Results comparing pure GPU Reduction, vs partial GPU and final reduction on CPU

Reduce Results			
Number of elements(N)	**CPU+GPU(ms)**	**GPU(ms)**	**Difference**
2	0.098264	0.009216	0.089048
4	0.044216	0.00512	0.039096
8	0.038192	0.006144	0.032048
16	0.0402	0.00512	0.03508
32	0.038168	0.004096	0.034072
64	0.042192	0.007168	0.035024
128	0.041192	0.006144	0.035048
256	0.038192	0.00512	0.033072
512	0.041192	0.00512	0.036072
1024	0.038144	0.00512	0.033024
2048	0.036144	0.00512	0.031024
4096	0.039168	0.00512	0.034048
8192	0.040168	0.00512	0.035048
16384	0.15888	0.140288	0.018592
32768	0.177264	0.090112	0.087152
65536	0.20684	0.115712	0.091128
131072	0.251824	0.201728	0.050096
262144	0.347768	0.299008	0.04876
524288	1.023104	0.720864	0.30224
1048576	1.850936	1.303584	0.547352
2097152	3.286824	2.44224	0.844584
4194304	6.820296	4.768768	2.051528
8388608	13.767648	9.659392	4.108256
16777216	27.605489	19.326977	8.278512
33554432	48.021015	38.444031	9.576984
67108864	83.370714	76.001282	7.369432
134217728	160.574613	147.641312	12.933301
268435456	321.667375	290.295807	31.371568
536870912	639.751507	581.081116	58.670391
1073741824	1302.849868	1200.434204	102.41566
2147483648	3768.733172	3548.821533	219.91164

The number of elements increased by 2^x, where x is from 1..31, these values were chosen to clearly see a trend emerge as the number of element increase.

With the above results we can generate the following graphs:

Figure 4 – (Comparing execution time using final reduction of GPU, vs CPU):

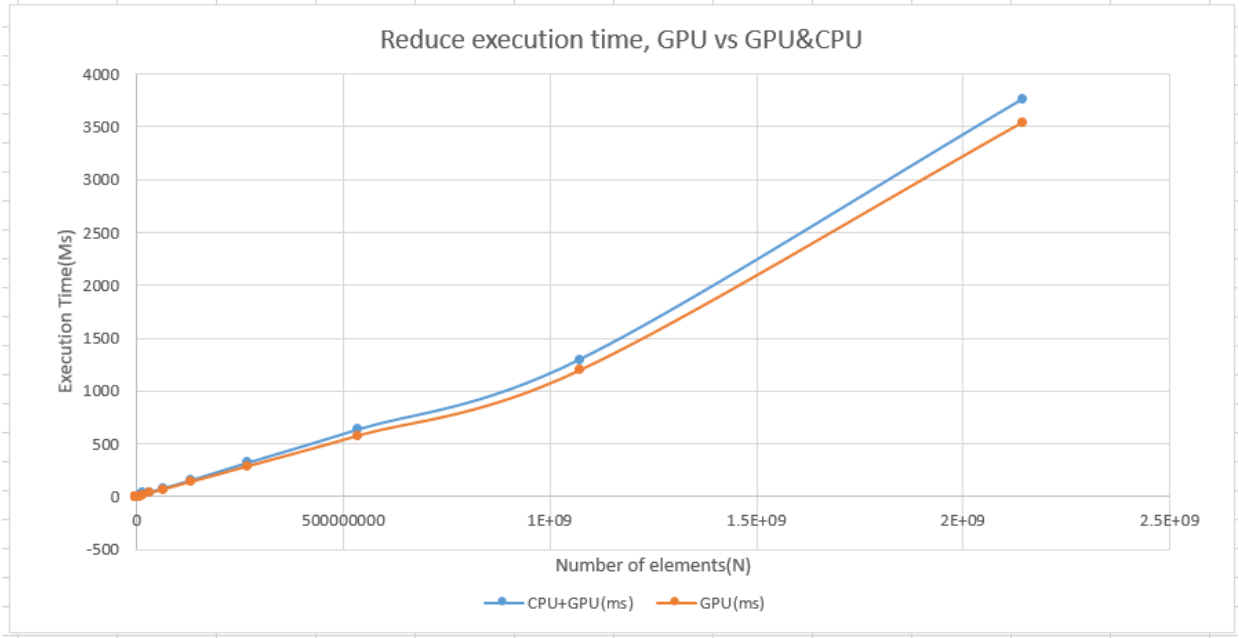


Figure 5- (More granular results):

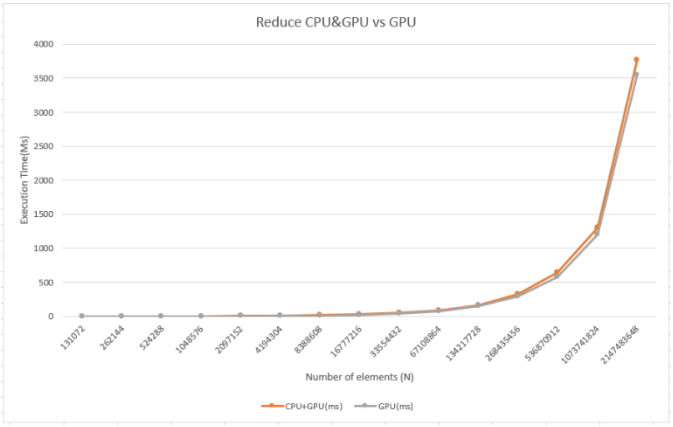
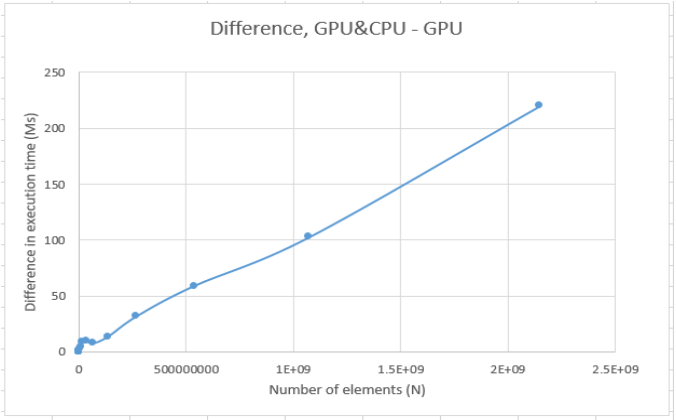


Figure 6 - (Difference: CPU - GPU final reduce)



Comparison:

As is evident from figure 4 and 6, we see a significant improvement by using the GPU for the final reduction, as opposed to using the CPU for the final reduction. We primarily see this however when the number of elements (N) approaches larger numbers, this being since initialising a reduction on the GPU has appropriate overhead in calling the kernel, and as such to offset the overhead we primarily see the most benefit when the number of elements are large.

The main reason in the GPU implementation is faster is that a GPU is optimised for throughput (i.e the ability to large amounts of operations at a given time), which means that lots of simple operations can be completed at a given time, and the reduce algorithm is designed as such to take advantage of this ability. This means we can easily see an increase in execution time as the reduce algorithm works faster on a GPU than serially on the CPU.

Another reason for the increased execution time on the GPU is that the GPU uses shared memory to store the array elements, and since the reduce algorithm needs to access the same values multiples times, we see an increase in execution time. Whereas the CPU version must execute by using main memory for the serial execution, which is inheritably slower than the GPU's shared memory architecture, generally.

2.3) Reduce implementation and thread divergence

The CUDA programming language tries to extract away the underlying physical hardware, as such to make coding easier, however it is important to keep in mind how the physical hardware functions. The CUDA programming model, breaks down elements into grids, blocks and threads, whereas from a hardware perspective it is broken down into SM (Streaming Multiprocessors), CUDA Cores, warps and threads. Where warps are a function of the CUDA devices using a SIMT Model (Single instruction multiple thread), where every 32 threads are divided into warps in which all the threads execute the same sequence of instructions. However, this unified execution of threads can lead to inefficiency's if the code leads to some divergence in the execution pattern between threads in each warp.

The initial approach of not splitting the data into halves is highly thread divergent, as there is significant interleaved addressing, as well as a significant number of threads that are inactive, at each iteration. For instance, if we assuming we had 32 elements, we would have it such that initially only ½ the threads are active, then $1/4^{th}$ then $1/8^{th}$, $1/16^{th}$ and finally only 1/32th of the threads active at a given time. This inactivity in threads are as a result of all instructions in a warp executing the same instruction and only neighbouring values being used, by a thread, at a given step, due the nature of the defined algorithm. More explanation on why this leads to divergence is explained further in section 5.2.

Thus, the only approach to preventing thread divergence and increased efficiency is to use sequential addressing and ensuring that we divined up the number of active threads by half each time, and altering the algorithm to facilitate such. By halving the number of threads that are active at a given time we ensure that all the warps (of 32 threads) are active at a given time, and no idle threads are present in a warp. More explanation of this is provided in section 5.2, as well as an explanation for using sequential addressing.

Scan

3.1) Implementation of Hillis and Steele inclusive scan

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

//Defining the different methods
void gpuScan(int N);
void printArray(float *arr, int sizeOfArray,bool printStament);
__global__ void scanKernel(float *idata,int n,int startIndex=0);
__global__ void findMaxValueArray(float* scanData,float* outputData);
__global__ void finalScanKernel(float* initialData,float* maxValues);


int main(void)
{
    //The below for loops allows the scan to be initialised with range of elements
    printf("Starting the scan kernel");
    for (int i = 1; i < 29; i++)
    {
        gpuScan(1<<i);
    }
    return 0;
}

//By calling the scan in its own method we can pass in multiple starting array sizes
void gpuScan(int N){
    //Methods for debugging
    bool debug = false;
    bool debugStep1=false;
    bool debugStep2=false;
    bool debugStep3=false;
    bool debugStep3Indepth=false;
    bool debugStep4=false;

    //Methods for initialising the timing functionaltiy
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaError_t err;

    // Allocate Unified Memory - accessible from CPU or GPU
    float *idata;
    cudaMallocManaged(&idata, N * sizeof(float));
    // Initialise the input data on the host
    // Making it easy to test the result
    for (int i = 0; i < N; i++)
    {
        idata[i] = 1.0f;
    }


    // STEP 1 : Generate an intial scan
    int grid_size = ((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
    cudaEventRecord(start);
    scanKernel<<<grid_size, BLOCK_SIZE>>>(idata,BLOCK_SIZE);
    // Wait for GPU to finish before accessing on host
    err = cudaDeviceSynchronize();
    cudaEventRecord(stop);
    printf("Kernel Errors: %s\n", cudaGetErrorString(err));
    float totalExecutionTime = 0;
    cudaEventElapsedTime(&totalExecutionTime, start, stop);
    // Now output the resulting array:
    if(debug)
        printf("\nSTEP 1: Generating initial scan values, grid dim: %d, block dim: %d
\n",grid_size,BLOCK_SIZE);
    printArray(idata,N,debugStep1); //Only if debugging is active, then it is printed

```

```
67    // Step 2: Generate scan of the max values of the output, i.e the last element in each block.
68     // Then apple a scan within the kernel call
69     float *scanMax;
70     cudaMallocManaged(&scanMax, grid_size * sizeof(float));
71     cudaEventRecord(start);
72     findMaxValueArray<<<grid_size, BLOCK_SIZE>>>(idata,scanMax);
73     err = cudaDeviceSynchronize();
74     cudaEventRecord(stop);
75     //Record execution time and add to the total
76     float temp=0;
77     cudaEventElapsedTime(&temp, start, stop);
78     totalExecutionTime+=temp;
79
80     //Print output values for debugbing
81     if(debug)
82         printf("\n STEP 2: Finding max scan values, grid dim: %d, block dim: %d
   \n",grid_size,BLOCK_SIZE);
83     printArray(scanMax,grid_size,debugStep2);
84
85     // Step3 : Apply scan again to max values from previous scan, this needs to be done by calling the
   scan kernel
86     //multiple times depending on how many elements fit into a block, since you need cross block
   communication to ensure
87     //overall scan method can functionm, as such a fixed stride is used to select intervals to add up
   the last block element.
88     int tempBlockSize=512; //Stride value must be below 1024, as that is max block size for cuda
89     int scanGridSize=((grid_size + tempBlockSize - 1) / tempBlockSize);
90     int startIndex=0;
91     int endIndex=0;
92     while(scanGridSize>0){
93         //Initialising the scan start values.
94         scanGridSize=scanGridSize-1;
95         endIndex=grid_size-(tempBlockSize*(scanGridSize));
96
97         if(debugStep3Indepth)
98             printf("\nscanGridsize %d,startIndex:%d,endIndex:%d\n",scanGridSize,grid_size,endIndex);
99         //Calling kernel, as well as intialising the timing for the method.
100        cudaEventRecord(start);
101        scanKernel<<<1, tempBlockSize>>>(scanMax,grid_size,startIndex);
102        cudaDeviceSynchronize();
103        cudaEventRecord(stop);
104        //Adding timing to overall timing
105        cudaEventElapsedTime(&temp, start, stop);
106        totalExecutionTime+=temp;
107        //Setting the kernel start point for next itteration.
108        startIndex=endIndex;
109
110        // Only up untill the last block should the previous blocks last value should be added to next
   blocks
111        // starting value!
112        if(scanGridSize>0){
113            scanMax[endIndex]+=scanMax[endIndex-1];
114            if(debugStep3Indepth)
115                printf("\nscanMax[endIndex+1]:%d ,scanMax[endIndex]
   %d\n",scanMax[endIndex+1],scanMax[endIndex]);
116
117        }
118    }
119    if(debug)
120        printf("\n\n STEP 3: Apply scan again to max values from previous scan, grid dim: %d, block
   dim: %d , meaning max %d elements \n",scanGridSize,BLOCK_SIZE, scanGridSize*BLOCK_SIZE);
121    printArray(scanMax,grid_size,debugStep3);
```

```
121    // Step 4: The step now is to ensure that you can apply the summation to the remaining values
122     // Such that values in scanMax get added to the original array
123     cudaEventRecord(start);
124     finalScanKernel<<<grid_size, BLOCK_SIZE>>>(idata,scanMax);
125     err = cudaDeviceSynchronize();
126     cudaEventRecord(stop);
127     cudaEventElapsedTime(&temp, start, stop);
128     totalExecutionTime+=temp;
129 
130     // Appropraite debuggin methods if needed
131     if(debug)
132         printf("\n\n STEP 4: Adding scan max values to original array of values , grid dim: %d, block
    dim: %d \n",grid_size,BLOCK_SIZE);
133     printArray(idata,N,debugStep4);
134 
135     printf("\n ALL GPU: N: %d ,exuction time: %f , Last Elem: %f\n",N,totalExecutionTime,idata[N-1]);
136 
137     //Free used memory
138     cudaFree(idata);
139     cudaFree(scanMax);
140     cudaDeviceReset();
141 }
142 
143 // Method for adding scan values to the output array
144 // Not shared memeory is not used since there will be no tangible benfiit,
145 // since each kerenl is works indpenent of each other in realtion to global memory
146 __global__ void finalScanKernel(float* initialData,float* maxValues){
147     int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
148     // All elements are added to the next value, provided the block id is not block 0
149     if(blockIdx.x!=0){
150         initialData[thIdx]+=maxValues[blockIdx.x-1];
151     }
152 }
153 
154 
155 //Scan kerenel is applied to the initial set of values
156 __global__ void scanKernel(float *idata,int n,int startIndex)
157 {
158     int thIdx = threadIdx.x + blockIdx.x * blockDim.x+startIndex;
159     int tid = threadIdx.x;
160     //Add appropriat data elements to shared memeory of the blocks
161     __shared__ float temp[1024];
162     temp[tid] = idata[thIdx];
163     __syncthreads();
164 
165     // The fllowiing loop creats an initial scan as per the scan algorithm
166     for (int offset = 1; offset < n; offset *= 2)
167     {
168         if (tid >= offset)
169             temp[tid] += temp[tid - offset];
170         __syncthreads();
171     }
172     idata[thIdx] = temp[tid];
173 }
```

```
173
174 // Essentially responsible for finding the last element in a given block and storing the
175 // output in an array for further analysis
176 __global__ void findMaxValueArray(float* input,float* output){
177     int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
178     int tid = threadIdx.x;
179     //This makes the assumption that each max value in a block is the last element in the
180 blockif(tid==BLOCK_SIZE-1){
181         output[blockIdx.x]=input[thIdx];
182     }
183 }
184
185 //Allows easy printing of data balues for for debugging.
186 void printArray(float *arr, int sizeOfArray, bool printStament)
187 {
188     if(printStament){
189         printf("Printing %d values \n", sizeOfArray);
190         for (int i = 0; i < sizeOfArray; i++)
191         {
192             printf("%f|", arr[i]);
193             if((i+1)%BLOCK_SIZE==0){
194                 printf("\t");
195             }
196         }
197         printf("\n Fin printing");
198
199     }
200     // printf("\n");
201 }
```

The main idea behind the code:

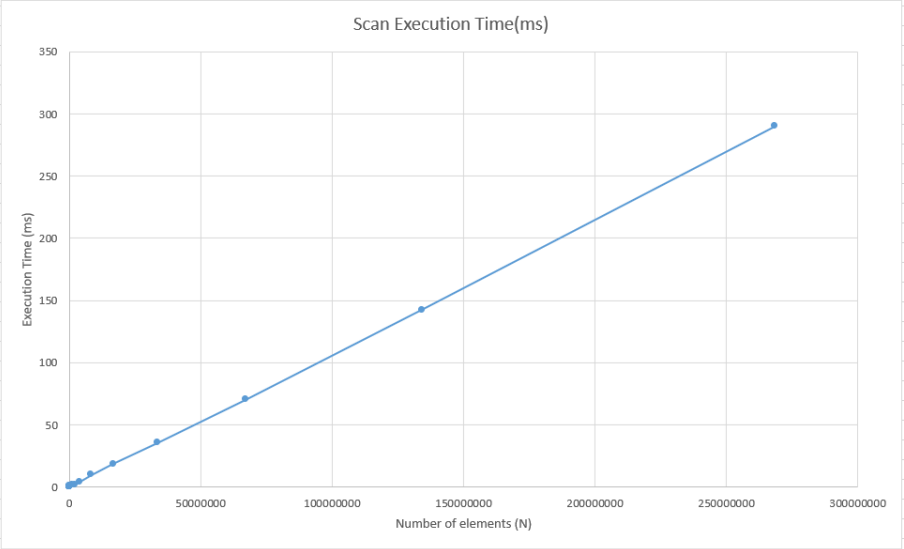
The code follows the 4 main steps, first we apply an initial scan to all the elements, block by block. We then find the last element in each block and store those values into another array of values. We then apply a scan to the values in array of max values, if the number of elements do not fit inside a single scan block, we increment the starting value of the next block with the last value of the previous block, as such to enable the scan to function across blocks. Finally, we add the scan results to the original set of scan values, where we only add the elements form block 1 onwards, since the values in the block 0 of the initial scan values will not change, whereas the subsequent block values will. Whilst doing all this we also ensure that appropriate timing methods are run for all appropriate kernel calls, as such to ensure we can plot the execution times, in accordance with the number of elements in the initial array N.

3.2) Scan results

The below are the results and associated graphs for the scan implementation:

Number of Elements(N)	Execution TIme(ms)
2	0.135168
4	0.145408
8	0.139264
16	0.13312
32	0.14336
64	0.142336
128	0.140288
256	0.101376
512	0.154624
1024	0.13824
2048	0.145408
4096	0.159744
8192	0.191632
16384	0.206848
32768	0.232688
65536	0.263888
131072	0.377856
262144	0.515072
524288	0.772336
1048576	1.393664
2097152	2.367488
4194304	4.150272
8388608	9.537536
16777216	18.391041
33554432	35.251202
67108864	70.226944
134217728	142.306625
268435456	290.044891

Figure 7 – Scan execution time



As is evident from the results, and the associated figure 7, the execution time of scan grows linearly, such that every time the number of elements doubles the execution time of the scan, on average, doubles. This being as the number of elements increase the number of operations increases at the same rate, i.e the algorithm is an O(n), over a parallel implementation.

3.3) Improvements to the Hillis and Steele parallel implementation:

The current implementation stated above, as is does most of the operation on the GPU itself, rather than on the CPU, where a less optimised implementation would do the final stages on the CPU. Note that a clear explanation of the underlying algorithm is defined in section 5.3. In addition to that, we can further improve the current code with the following improvements:

1) Use shared memory – Under the current implementation(below) the final addition, in which we add the max value of each scan to the relevant array values are done without using shared memory, therefore by changing the kernel to add elements to the final output array from shared memory than global memory we see an improvement in execution time; this can be primarily see when we have a large number of elements.

```
// Method for adding scan values to the output array
// By changing the maxValue elements to shared memory we can see a signiciant improvement in execution times.
__global__ void finalScanKernel(float* initialData,float* maxValues){
    int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
    // All elements are added to the next value, provided the block id is not block 0
    if(blockIdx.x!=0){
        initialData[thIdx]+=maxValues[blockIdx.x-1];
    }
}
```

2) Change the scan kernel to a double buffered Hillis and steel kernel. By doing this we reduce the change of conflicts in read and write requests being done at a given time by multiple threads, as such by having 2 shared memory arrays, one for reading and the other for writing, we reduce the chance a conflict may occur; thus improving efficiency of the code.

We can also see a slight improvement in execution speed by using constant memory for the initially array of elements.

Histogram

4.1) Parallel implementation of histogram

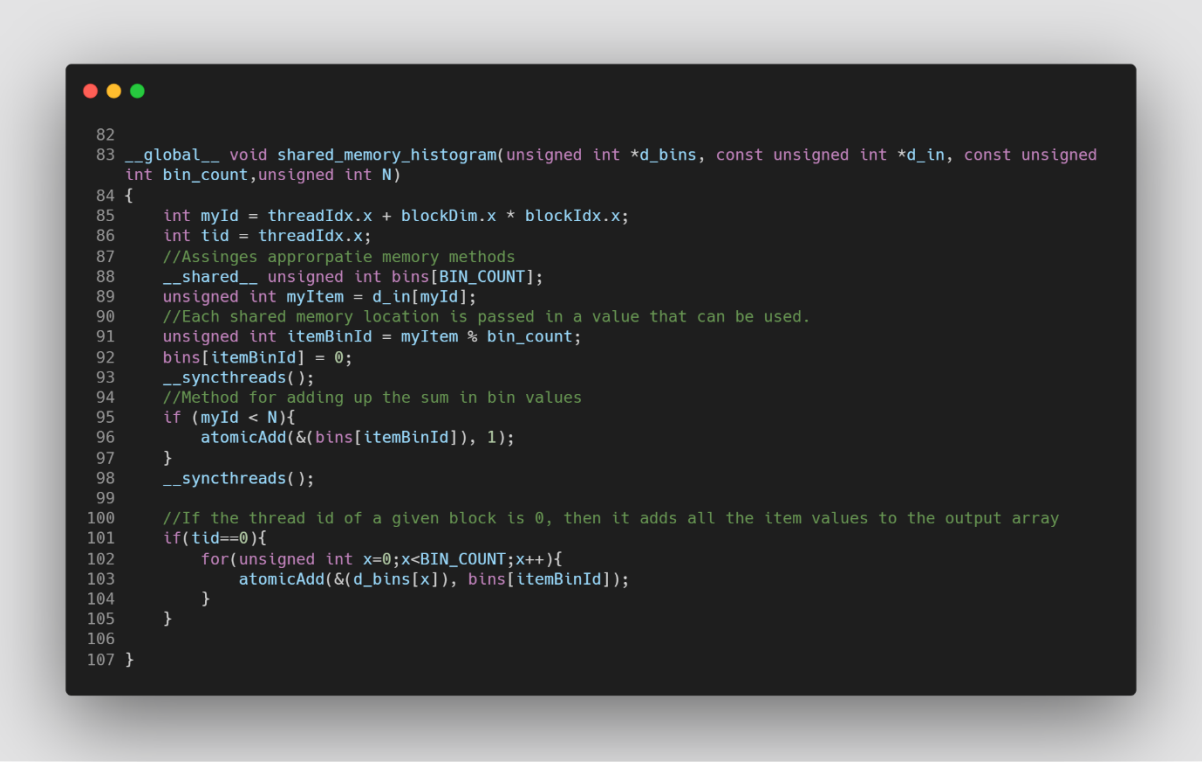
```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define BLOCK_SIZE 64
// This sets the number of bins in the histogram
#define BIN_COUNT 8

//Below states the method needed for histogram
__global__ void simple_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N);
__global__ void shared_memory_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N);
void sharedMemorykernelCallMethod(unsigned int N,bool debug);
void nonSharedkernelCallMethod(unsigned int N,bool debug);

int main(void)
{
    //Method for passing in values of N, that are multiples of 2^x
    for (int k = 1; k < 32; ++k)
    {
        //Initialising the values
        unsigned int N = (1 << k);
        nonSharedkernelCallMethod(N,false);
        // printf("\n");
        // sharedMemorykernelCallMethod(N,false);
    }
    return 0;
}

//Kernel to ensure non shared parallel implementation
void nonSharedkernelCallMethod(unsigned int N,bool debug){
    //Appropriate timing methods for cuda
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    unsigned int *d_bins; // This is the array that will contain the histogram
    unsigned int *d_in;   // This is the array that will contain the input data
    // We will use the CUDA unified memory model to ensure data is transferred between host and device
    cudaMallocManaged(&d_bins, BIN_COUNT * sizeof(unsigned int));
    cudaMallocManaged(&d_in, N * sizeof(unsigned int));

    //We firstly generate input data that can be used
    for (unsigned int i = 0; i < N; i++)
    {
        d_in[i] = i;
    }
    // We initialise the size of each bin to equal 0 initially
    for (unsigned int i = 0; i < BIN_COUNT; i++)
    {
        d_bins[i] = 0;
    }

    //THIS ALLOWS US TO FUNCTION WITH VALUES OF N THAT ARE NOT MULTIPLES OF BLOCK_SIZE
    unsigned int grid_size = ((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
    cudaEventRecord(start);
    simple_histogram<<<grid_size, BLOCK_SIZE>>>(d_bins, d_in, BIN_COUNT,N);
    // wait for Device to finish before accessing data on the host
    cudaDeviceSynchronize();
    // Recording the execution time below
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float totalExecutionTime = 0;
    cudaEventElapsedTime(&totalExecutionTime, start, stop);

    // Now we can print out the resulting histogram
    unsigned int total = 0; //For debugging
    for (unsigned int i = 0; i < BIN_COUNT; i++)
    {
        if(debug)
            printf("Bin no. %d: Count = %d\n", i, d_bins[i]);
        total += d_bins[i];
    }
    //Do appropriate error handling if required.
    if(total!=N){
        printf("Error for value %d\n",N);
    }
    //printf("NonShared: Elements N: %d, histogram total: %d, excution time:%f", N, total, totalExecutionTime);
    printf("%f|",totalExecutionTime);
    if (debug)
        printf("\n Blocks: %d, Threads %d , Histogram total %d\n", grid_size, BLOCK_SIZE, total);
}
```

```
82
83 __global__ void simple_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int
   bin_count,unsigned int N)
84 {
85     int myId = threadIdx.x + blockDim.x * blockIdx.x;
86     int myItem = d_in[myId];
87     //The below method is just one of MANY ways to allocate elements to bins
88     // Below is purley just an illustation
89     int myBin = myItem % bin_count;
90     //The below condition with additonal blocks sizes above allow the method
91     // to function with any number of elements and NOT just multiples of BLOCK_SIZE
92     if (myId < N)
93         atomicAdd(&(d_bins[myBin]), 1);
94 }
```

The above states the histogram implementation, we first initialise an input array of values, we then allocate the memory for an array of bins that will store the results. The current implementation allows any size of input to passed into the kernel, and not just multiples of the block size, this is done through the validation done by the kernel. The kernel is then called to assign the values to the appropriate histogram bins, note that currently we are assigning the element to bin by using the modulus operator, but any other allocation can be defined and used. The values are added to the output matrix by using the atomic add function, that ensures there are no race conditions when adding a value to the array of bins, this however acts as a significant bottleneck.

4.2) Parallel implementation of histogram using shared memory

```
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define BLOCK_SIZE 64
// This sets the number of bins in the histogram
#define BIN_COUNT 8

//Below states the method needed for histogram
__global__ void simple_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N);
__global__ void shared_memory_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N);
void sharedMemorykernelCallMethod(unsigned int N,bool debug);
void nonSharedkernelCallMethod(unsigned int N,bool debug);

int main(void)
{
    //Method for passing in values of N, that are multiples of 2^x
    for (int k = 1; k < 32; ++k)
    {
        //Initialising the values
        unsigned int N = (1 << k);
        // nonSharedkernelCallMethod(N,false);
        // printf("\n");
        sharedMemorykernelCallMethod(N,false);
    }
    return 0;
}

void sharedMemorykernelCallMethod(unsigned int N,bool debug){
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    unsigned int *d_bins; // This is the array that will contain the histogram
    unsigned int *d_in;   // This is the array that will contain the input data
    // We will use the CUDA unified memory model to ensure data is transferred between host and device
    cudaMallocManaged(&d_bins, BIN_COUNT * sizeof(unsigned int));
    cudaMallocManaged(&d_in, N * sizeof(unsigned int));

    //We firstly generate input data that can be used
    for (unsigned int i = 0; i < N; i++)
    {
        d_in[i] = i;
    }
    // We initialise the size of each bin to equal 0 initially
    for (unsigned int i = 0; i < BIN_COUNT; i++)
    {
        d_bins[i] = 0;
    }

    //THIS ALLOWS US TO FUNCTION WITH VALUES OF N THAT ARE NOT MULTIPLES OF BLOCK_SIZE
    unsigned int grid_size = ((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
    cudaEventRecord(start);
    shared_memory_histogram<<<grid_size, BLOCK_SIZE>>>(d_bins, d_in, BIN_COUNT,N);
    // wait for Device to finish before accessing data on the host
    cudaDeviceSynchronize();
    // Recording the execution time below
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float totalExecutionTime = 0;
    cudaEventElapsedTime(&totalExecutionTime, start, stop);

    // Now we can print out the resulting histogram
    unsigned int total = 0; //For debugging
    for (unsigned int i = 0; i < BIN_COUNT; i++)
    {
        if(debug)
            printf("Bin no. %d: Count = %d\n", i, d_bins[i]);
        total += d_bins[i];
    }
    //Do appropriate error handling if required.
    if(total!=N){
        printf("Error for value %d\n",N);
    }
    // printf("SharedMemory: Elements N: %d, histogram total: %d, excution time:%f\n\n", N, total, totalExecutionTime);
    printf("%f|",totalExecutionTime);
    if (debug)
        printf("\n Blocks: %d, Threads %d , Histogram total %d\n", grid_size, BLOCK_SIZE, total);

}

```



```
82
83 __global__ void shared_memory_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned
   int bin_count,unsigned int N)
84 {
85     int myId = threadIdx.x + blockDim.x * blockIdx.x;
86     int tid = threadIdx.x;
87     //Assinges approrpatie memory methods
88     __shared__ unsigned int bins[BIN_COUNT];
89     unsigned int myItem = d_in[myId];
90     //Each shared memory location is passed in a value that can be used.
91     unsigned int itemBinId = myItem % bin_count;
92     bins[itemBinId] = 0;
93     __syncthreads();
94     //Method for adding up the sum in bin values
95     if (myId < N){
96         atomicAdd(&(bins[itemBinId]), 1);
97     }
98     __syncthreads();
99
100    //If the thread id of a given block is 0, then it adds all the item values to the output array
101    if(tid==0){
102        for(unsigned int x=0;x<BIN_COUNT;x++){
103            atomicAdd(&(d_bins[x]), bins[itemBinId]);
104        }
105    }
106
107 }
```

The above implementation builds up on the code defined in section 4.1, however the kernel now uses shared memory, this can be done by change the kernel implementation as sated from lines 83 to 107.

4.3) Histogram Results

Below are the results for using shared memory, for the atomic add as opposed to using global memory for the atomic add.

	No Shared Memory	Shared Memory	
Number of Elements(N)	Execution TIme(ms)	Execution TIme(ms)	Difference(Shared-None)
8	0.07168	0.474112	-0.402432
16	0.065536	0.064512	0.001024
32	0.062464	0.063488	-0.001024
64	0.063488	0.063488	0
128	0.036896	0.03584	0.001056
256	0.032768	0.033792	-0.001024
512	0.063488	0.062464	0.001024
1024	0.069632	0.077824	-0.008192
2048	0.065536	0.067584	-0.002048
4096	0.095232	0.067584	0.027648
8192	0.134144	0.10752	0.026624
16384	0.14336	0.140288	0.003072
32768	0.193536	0.149504	0.044032
65536	0.288768	0.183296	0.105472
131072	0.45056	0.26624	0.18432
262144	0.421888	0.338944	0.082944
524288	0.90624	0.513024	0.393216
1048576	1.6384	0.71168	0.92672
2097152	3.491808	1.1776	2.314208
4194304	6.729728	2.412544	4.317184
8388608	13.782016	4.584448	9.197568
16777216	27.956224	9.61536	18.340864
33554432	54.959103	19.337215	35.621888
67108864	108.534782	37.704704	70.830078
134217728	202.65062	73.443329	129.207291
268435456	390.324219	129.762299	260.56192
536870912	778.866699	258.40332	520.463379
1073741824	2047.531006	551.498779	1496.032227

Figure 8 – Atomic add using shared memory vs on purely global memory

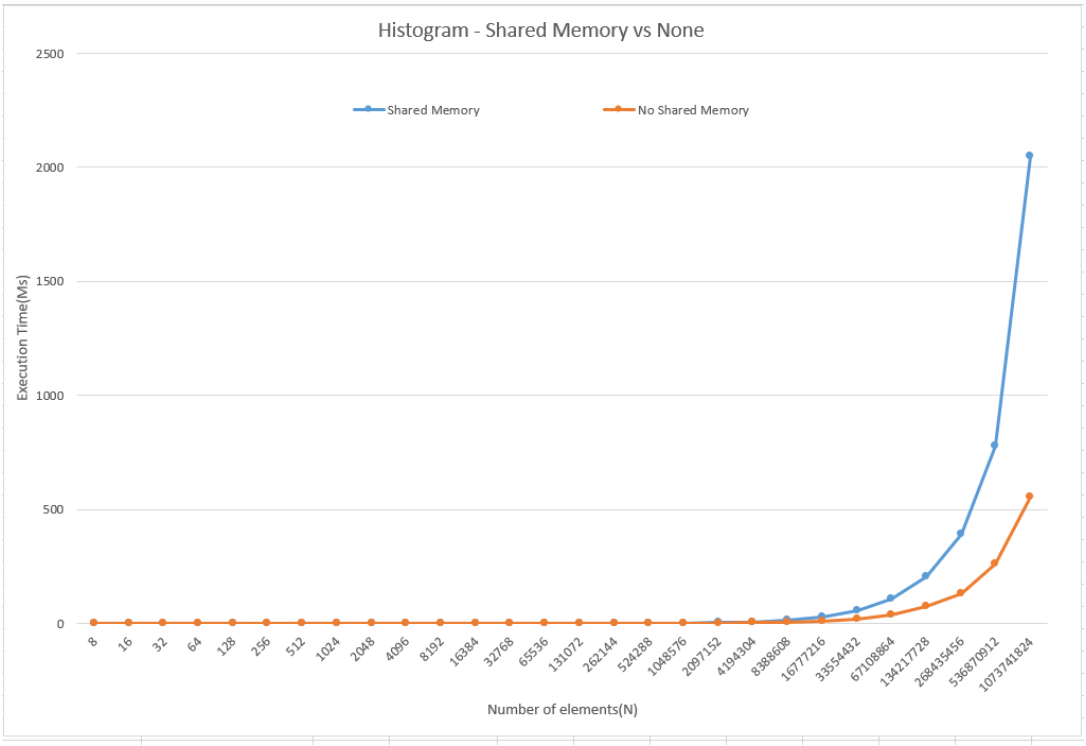
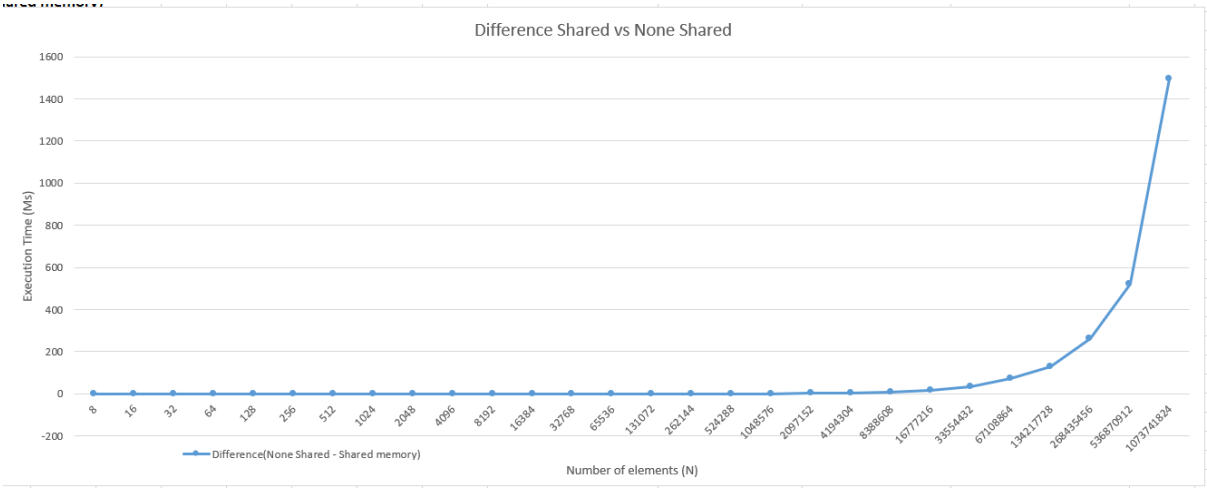


Figure 10 –Difference graph (Global memory atomic add – shared memory atomic add)



As is evident form the results, using shared memory for atomic add is significantly faster than using global memory, this is even more pronounced the larger the number of elements, N, increase.

The main difference in execution times are as a result of the use of the atomic add functionality of CUDA. Atomic add operations ensure that no race conditions exist, where race conditions are as a result of multiple threads trying to update the same given data elements at the same time, leading to incorrect updating of the associated data. Therefore, functions like the atomic add, of CUDA, ensure that no race conditions exist by ensuring operations occur sequentially, which prevent race conditions. However, using atomic add leads to performance issues, since it forces a kernel to run serially instead of in a parallel manor, as such atomic add method calls should in general be limited.

The negative effects of atomic adds are more pronounced whilst the atomic add occurs on global memory as opposed to shared memory, since global memory is significantly slower than shared memory (as outlined in question1). As mentioned before, by using atomic add we are essentially forcing the kernel to execute sequentially(when a thread tries to access the same data another thread tries to access), so if memory access is slower the total execution time (for a set of atomic add functions) decreases, since it takes longer to access and update memory locations. Therefore, if memory access is slower the total time to execute all atomic add functions increase significantly, as the memory access time acts as the limiting factor. Thus, since the memory access time of shared memory is on a magnitude faster than global memory, the total execution time for a kernel that does the majority of the atomic adds on shared memory is significantly faster.

Therefore the kernel outlined in section 4.2 (lines 83 to 107) uses 2 atomic adds, however since the majority of the atomic adds occur on shared memory, and only a minority of the atomic adds occur on global memory, and since shared memory is faster we can see from the results that the shared memory kernel is still faster even with 2 sets of atomic adds. Furthermore if we increased the number of bins in the histogram we see an increase in the execution time, since there is less chance of conflicting atomic adds occurring on a given data element at a given instance, thus it is evident that by having fewer atomic adds on global memory we see significant improvements in execution.

Note: The following section builds upon the concepts explored above, and builds upon concepts in a sequential manor below, as such to minimize the same information from being repeated.

Q5: Further expansion

The general concepts surrounding CUDA devices:

Parallel architectures such as Invidia's CUDA devices are built upon the idea of optimising hardware for throughput rather than latency, by having multiple slower processors do multiple operations at a given time rather than having a singular processor sequentially solve multiple processes, we increase the number of overall operations we can complete in a given time.

The CUDA programming language tries to extract away the underlying physical hardware, as such to make coding easier, however it is important to keep in mind how the physical hardware functions. The CUDA programming model, breaks down elements into grids, blocks and threads, whereas from a hardware perspective it is broken down into SM (Streaming Multiprocessors), CUDA Cores, warps and threads. Where warps are a function of the CUDA devices using a SIMT Model (Single instruction multiple thread), where every 32 threads are divided into warps in which all the threads execute the same sequence of instructions at a given instance; which can lead to additional problems such as thread divergence which are explored in subsequent sections.

The speed of execution of a CUDA device is a function of both the base clock speed of the CUDA cores, the number of CUDA cores found on a given device, as well as the amount of the different types of memory found on a given CUDA device; with other factors such as transfer speed between the CPU and GPU playing limited/constant effect across different CUDA device implementations. As such the ability of a program to take advantage of the physical hardware is a crucial importance, one of the most important being the use of different memory types. Some of the different types of memory available to a CUDA programs/devices can be summarised from fast to slow as:

- Register Memory – Memory allocated to a given thread, which exists only for the lifetime of the given thread, it is the fastest possible form of memory closest to the given chip but is unavailable to share data between threads.
- Shared Memory – Data that is visible to all threads within a given block but lasts only for the lifetime of the block. Where the main benefit is it allows cross communication between threads in each block, at a fast speed. At a max size of the 64k bytes.[1]
- Constant Memory – Constant memory is found off chip but has faster access times than using global memory since it is restricted to only reading data than also being able to write data. The benefits of using constant memory can only been see If threads in a warp try and access the same memory location at a given time, rather than multiple threads accessing different elements.
- Global Memory & Local Memory – Visible to all threads within the given CUDA application, as well as the host, and is available for the entire lifetime of the host application but is extremely slow in comparison to shared memory, often in the magnitude of 100X slower[1]. Local memory is essentially global memory that is specific to a given thread, which is found on device but of a given CUDA Core and is essentially used as an overflow for data that does not fit into registers.

Generally, we find that to optimise the performance of a given application we need to ensure that we minimise the number of global memory access and increase the number of shared memory access, since shared memory is often in the magnitudes of 100X faster than global memory. We also need to ensure that we optimise each thread to do the maximum amount of work that needs to be done at a given instance. As such we can visualise this as:

$$Arithmatic\ Intensity = \frac{The\ number\ of\ computer\ operations}{The\ number\ of\ global\ memory\ acess}$$

Where in our goal is to maximise Arithmetic Intensity by minimising global memory access and increase the number of operations a given thread is responsible for executing.

5.1) Parallel implementation of matrix multiplication

Basic Idea + naive implementation

Matrix Multiplication is a mathematical operation that occurs on two matrixes A and B, where in you multiply the elements in the row of matrix A with the column of Matrix B. Which can be further summarised as:

$$C_{\mathrm{ji}} = \sum_{k=1}^{m} a_{jk}\, \mathrm{b}_{ki}$$ Provided A is a n x m matrix and B is a n x p matrix, for j=1,..n and i=1,…p

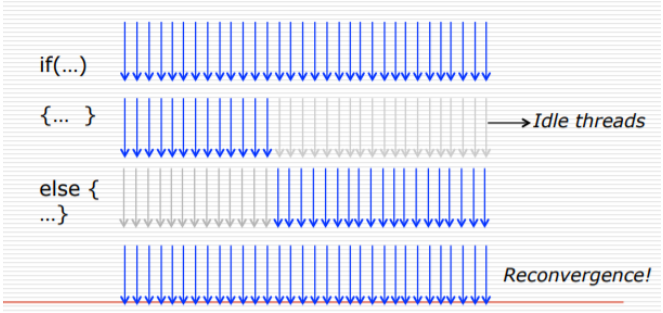
The naive approach to solve this problem is to assign each thread an element in the matrix C, which is then responsible for calculating the relevant element in matrix C, by iterating through the relevant rows and columns in matrix A and B. Therefore, matrix multiplication uses a gather communication method to function, where in each thread uses multiple data to calculate an overall result. This can be done easily using CUDA programming languages concept of grids and blocks.

```
__global__ void MatrixMultKern(const Matrix A, const Matrix B, const Matrix C) {
   // Calculate the column index of C and B
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   // Calculate the row index of C and of A
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   if ((row < A.height) && (col < B.width)) {
       float Cvalue = 0;
       // each thread computes one element of the block sub-matrix
       for (int k = 0; k < A.width; ++k) {
           Cvalue += A.elements[row * A.width + k] * B.elements[k*B.width + col];
       }
       C.elements[row * C.width + col] = Cvalue;
   }
  }
```

Under structure of CUDA programming we can initialise the kernel with a 2Dimensional block and grid size, which we can then use to easily access the elements in the input matrixes A and B. For instance, given a thread we can access an element at a given matrix location by using its threadID's in relation to the x and y dimensions of the block, which essentially allows you to map onto the dimensions of the matrixes. Whilst using multiple dimensions we need to however ensure that we do not surpass the limitations of each dimension, for example in a block each dimension x, y, z must be below 1024, 1024 and 64 respectively where x * y * z ≤ 1024 must also hold. Similarly, we must ensure that blocks within a grid have dimensions x, y, z below 65535.

As mentioned before, CUDA uses a SIMT architecture meaning that, a singular instruction is executed on multiple threads at a given time. This leads to the idea of warps, which is a collection of 32 threads that execute the same instruction at a given time. If, however a given thread in a warp wants to execute a different instruction opposed to the other threads in a warp, you lead to the idea of thread divergence. Thread divergence is as therefore as a result of all threads in a warp being designed to execute the same instruction, thus if threads in a warp tries executing conflicting instructions some of the threads must become inactive, since all the warps must execute the same instruction. Therefore, thread divergence leads to extreme inefficiency in execution times, since maximum efficiency is when all the threads in a warp execute at a given time, instead of being idle. This divergence can often be seen in branch conditions, such as the combination of If and else statements, that lead to divergences in the code. The diagram below visualises the effect of thread divergence:

Figure 11 –Thread Divergence[1]:



Now under the naive implementation of Matrix multiplication we do not have to worry about thread divergence, since all the threads are active within a warp at a given time, and the kernel is not designed to have instances of divergence within the code. Since all the threads are active at a given instance to ensure the summation and multiplication occur at a given time, we therefore cannot optimise the number of operations that a given thread is allowed to do, since it is optimised currently to do the minimum most calculations needed for the matrix multiplication. Note however there is minor thread divergence at the boundaries of the matrix multiplication, when the row and column length exceed that of the matrix, but the effect can be assumed to be minimal, and there is no major source of thread divergence in the naive implementation.

Therefore, we can summarise the arithmetic intensity of the naive matrix multiplication using global memory as:

$$Arithmatic\ Intensity = \frac{The\ number\ of\ computer\ operations}{The\ number\ of\ global\ memory\ access}$$
$$= \frac{Matrix.A.width\ multiplications + Matrix.A.width\ additions}{2 * Matrix.A.width\ Global\ Reads}$$

Therefore, since we cannot decrease the number of operations, due to the nature of the calculations needed for matrix multiplication, the only way to increase arithmetic intensity is to decrease the total number of global memory access.

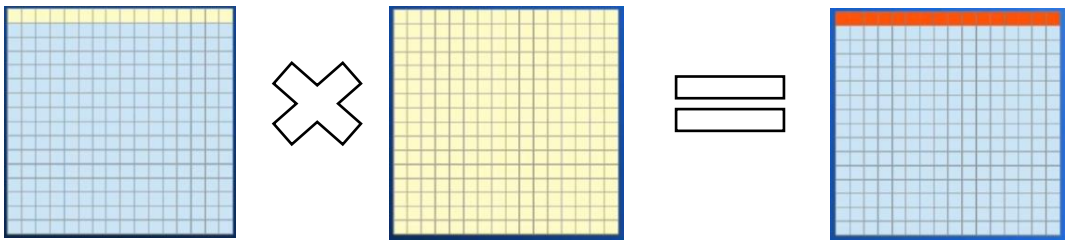
Optimised implementation

We could decrease the number of global memory access by moving the elements to constant memory, however this not be very efficient for the application of matrix multiplication; this being as a result of the nature of constant memory. In CUDA devices we have L1 and L2 cache, where L1 cache is found near a CUDA core, whereas L2 Cache is accessible from all threads that are active, these caches often are extremely quick to read and write to. The L2 and L1 cache use special and temporal locality to increase the efficiency of memory access, where temporal locality refers to a given instruction being used again and again, whereas spatial locality refers to subsequent instructions being ones that are close to each other in memory. L1 and L2 cache are assigned by the system itself using principals of locality, and not manually controlled by the user, as say for shared or global memory which are user defined, however constant memory takes advantage of the different caches by keeping elements on caches. In fact, global and local memory pass through the different cache levels before being used, unlike shared memory which directly interacts with the kernel, thus adding onto the overhead of global [2]; since when global memory call is made, we first check L1 cache, then L2, and then load the data from global memory. However, using constant memory has its disadvantages, as it requires the threads to access data in the same order and with regards to elements nearby each other, otherwise we have the same overhead as global memory. Since matrix multiplication does not fit these parameters, since a thread does not need to access all matrix elements, and the flow of data access is not sequential, we see no improvement in using constant memory to access data.

Therefore, the only way to increase arithmetic intensity is to instead use shared memory, shared memory is memory that is accessible to all threads within a given block, the main advantage being that shared memory has a considerably faster read and write time than global memory. From a hardware perspective shared memory is memory that are found on the associated streaming multiprocessors, where the streaming multiprocessors facilitates the execution of blocks and its associated threads. As mentioned before, the CUDA architecture breaks down the execution model into threads, blocks and grids, as such when a kernel is initialised its associated blocks are executed on the earliest available streaming multiprocessor, which schedules which block should execute at a given time. The streaming multiprocessor is responsible for the execution of a block and as such shared memory is associated to each block, since from a hardware perspective it is found on the streaming multiprocessors and is not accessible by other threads in other blocks, since they execute on different streaming multiprocessors. As such since the shared memory is on the streaming multiprocessor it is close to the associated CUDA cores that execute the kernel instructions, this locality to the CUDA cores is a main contributing factor to the speed of shared memory. [1] Whereas the global memory can be found physically further away from the associated CUDA Cores, and as such the need to check

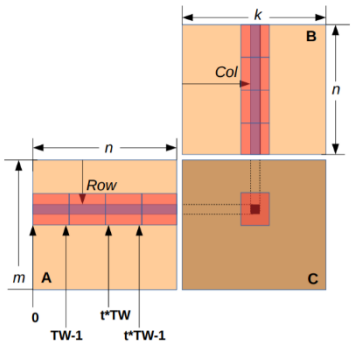
L1 and L2 cache before making read/write requests from Global Memory, as means to try and increase the execution time by using the associated caches to facilitate faster reads. Therefore, the use of shared memory in the multiplication kernel, allows us to increase the arithmetic intensity of the kernel, and thus decrease the execution time of the kernel.

However, to optimise the use of shared memory, we need to take into consideration key features surrounding actual matrix multiplication, in that a shared Tiled matrix multiplication utilises the fact that in a matrix multiplication threads will overlap in their use of row and column values. For instance, a given row of values of one matrix will be reused for all the column elements in a second matrix, as is visualised below:



We can take this concept further and assign a set of the values to shared memory which can then be used by the threads more efficiently since shared memory has a significantly faster read time than global memory; note that each thread is responsible for gathering one element of each of the tiles and storing the elements to shared memory. We start off with a tile of data and move from left to right in the first matrix, and top down in the second matrix, by the dimensions of the tile, whilst constantly adding onto a partial row by column matrix multiplication. This means that eventually we reach a point where we have a full sum of the results, even based of the partial total sums. This can be seen clearly below:

Figure 13 – Tiled matrix multiplication [3]:



Therefore, by using this approach of calculating the matrix multiplication sum by tiling values, we see a significant increase in the execution time of the matrix multiplication, as is evident from figure 1. This can be further seen as now the arithmetic intensity, with shared memory can be thought of as:

$$Arithmatic\ Intensity\ = \frac{Matrix.A.width\ multiplications + Matrix.A.width\ additions}{2 * Matrix.A.width\ Global\ Reads/BLOCK\_SIZE}$$

Therefore, by using the appropriate improvements, we see a significant increase in using a shared memory kernel over one that does not, as is evident in figure 1 and 3, and evident from the analysis of said results stated in question 1.

5.2) Reduce

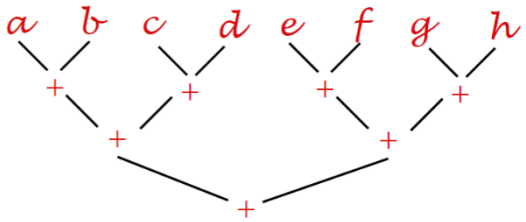
Note: This section builds upon the ideas explored in the introduction and section 5.1, to prevent repeating information

Reduce algorithm

A reduce operation is responsible for calculating a singular final value, from a set of values, using a given binary operation such as addition. The easiest way to implement reduce is to iterate though all elements in the given set of elements, however this is not easily parallelisable since under this basic approach we need the previous sum value to calculate the next sum.

Therefore, we need to implement scan differently for the parallel implantation. The naive way to do this in a parallel fashion is to do:

Figure 14 –Reduce Naive Parallel implementation [4]:



The naive parallel implementation is such that, at each stage we add up the nearest neighbour value to the a given element till we reach a singular value, for example the nearest neighbour at stage 1 of a is b, and we therefore add a and b, at the next stage the nearest neighbour of a and b is the sum of c and d, which we add together, and keep repeating till we reach a singular value. However, in actual implementation we need to consider how the elements are stored in memory and implement it such that we add together the nearest neighbour value.

To implement this, we first copy all the elements to memory, and access the elements with their associated index values, where the initial index value is 0. We then initialise a stride value of 1 at stage 1, where the stride value determines which values we add together. So, as an example in stage 1, to the index value of 0, we add the element at index 1, which is 1 stride away. We then add to the element at index 2 the element at index 3, which is again 1 stride away, and keep doing so till we reach the end of the array. We keep repeating these steps until we reach a singular value, however at each stage we increase the stride value such that the stride value S, at a given stage X, can be calculated by:

$$S = 2^{X-1}$$

However even though this approach is simple, there are many disadvantages to this algorithm, one disadvantage being that this algorithm relies on the number of elements to be at maximum the size of threads in the given block. Since, otherwise we would require a communication between kernels at the end of their lifecycle, which cannot be done directly in CUDA without a separate kernel invocation. However, this does not act as a significant limiting factor, since generally kernel invocations do not take significant overhead. The main performance limitations of the naive implementation are due to thread divergence and interleaved addressing.

The most basic naive implementation can be thought with the kernel stated in figure 15.

```
//Reduce kernel for device that allows the reduction to take place
__global__ void reduceKernel(unsigned int * d_in,unsigned int* d_out) {
    int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
    int tid = threadIdx.x; // Local ID within the current block
    __shared__ unsigned int temp[BLOCK_SIZE];
    temp[tid] = d_in[myId];
    __syncthreads();
    // do reduction in shared memory
    for (unsigned int s = 1; s < blockDim.x; s *= 2)
    {
        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];
    }
}
```

Figure 15 – Basic Reduce

One of the main errors with the kernel defined in figure 16, is that it leads to significant thread divergence. This is as a result of kernel deciding which thread is active at a given time using the modulus operation (line 11), this means that any thread that does not meet the if statement becomes inactive, and only if it does is it active. This division of active and inactive threads area as a result of the hardware implementation of CUDA, as mentioned before CUDA devices use a SIMT architecture, in which we divide up 32 threads into a warp which must all execute the same instruction at a given time, if a given thread in a warp tries to execute a different instruction than the other threads, we have a set of threads become inactive to maintain the fact all the instructions executing must be the same at a given time, in a warp. A more detailed explanation of this can be found in section 5.1.

This thread divergence therefore leads to half the number of threads being active at each step from the subsequent step. For example if we had 32 elements, after the first step (as defined in the algorithm above), we will have ½ the threads active at the initial step, followed by $1/4^{th}$ the threads active at the second step, followed by $1/8^{th}$, $1/16^{th}$ and $1/32^{nd}$ threads active at each subsequent step. This being as mentioned above, at each stage half the elements become inactive due to the modulus operation, that ensures only some of the threads are active at a given iteration of the for loop.

We can remove most of the thread divergence by removing the modulus operation, and changing the most basic kernel to:

Figure 16 – Basic Reduce Minimal divergence

```
1  //Reduce kernel for device that allows the reduction to take place
2  __global__ void reduceKernel(unsigned int * d_in,unsigned int* d_out) {
3      int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
4      int tid = threadIdx.x; // Local ID within the current block
5      __shared__ unsigned int temp[BLOCK_SIZE];
6      temp[tid] = d_in[myId];
7      __syncthreads();
8      // do reduction in shared memory
9      for (unsigned int s = 1; s < blockDim.x; s *= 2)
10     {
11         int index = 2*s*tid;
12         if (index < blockDim.x)
13         {
14             temp[index] += temp[index + s];
15         }
16     __syncthreads(); // make sure all adds at one stage are done!
17     }
18     // only thread 0 writes result for this block back to global memory
19     if (tid == 0)
20     {
21         d_out[blockIdx.x] = temp[tid];
22     }
23 }
24
```

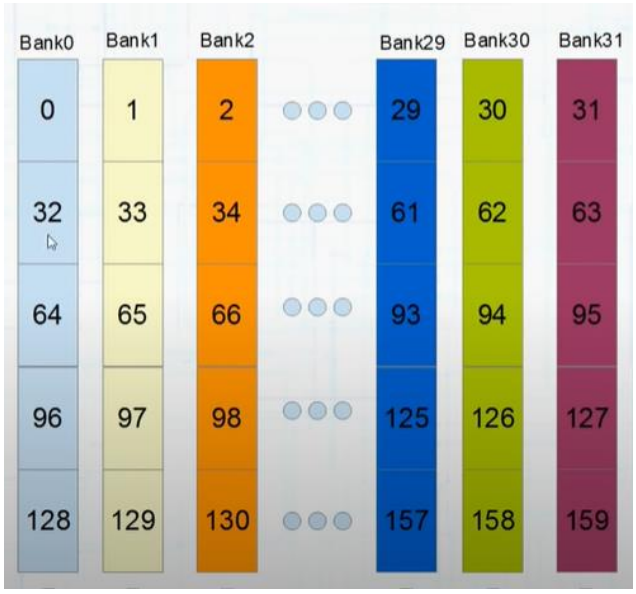
We can remove the modulus operation from the kernel (from figure 15) to the kernel stated above (figure 16) to remove the main cause of thread divergence. By making the above alteration, we are now accessing the elements based on the calculated index value instead of the modulus of the thread id, which means there is no need for threads in a warp to become inactive, as they are all active in accessing their relevant data, based on individual index values. However there is minor thread divergence in that if the index value is greater than the block dimensions, we see a minor thread divergence in the warp that contains the thread that does not meet the condition; however this is the smallest possible number of thread divergence we can have. This approach leads to a significant decrease in thread divergence, and as such a parallel improvement in exaction time. The approach stated in figure 16 can be further improved to prevent interleaved addressing, which can lead to bank conflicts since we are using shared memory, this concept can also be thought of as using gather approach than using a scatter approach.

As we explored in section 1 and 5.1, using shared memory is much more efficient than using global memory, however using shared memory does have its own downfalls; which is primarily due to bank conflicts, and as such we need to optimise our algorithm to prevent this. As with all memory access they are initialised by the threads in a warp, which must all execute the same instruction at a given time, whereas as if they do not a warp divergence occurs. When memory access occurs, we must copy the data from the associated memory location to the respective registers, and all threads make the request from shared memory at the same time, even if the threads try and access different memory addresses, provided no thread divergence.

Under the CUDA architecture, shared memory has 64k of storage, which are broken into 4-byte sections called words, which dependant on the data type used take a different amount of memory in a given word, for instance a word could be a 32 bit int which would then fit into a singular word. Therefore, when accessing a single byte form a word, we first return the entire word, and take the subset of the element we want. The words are then further divided into banks, where each word is assigned to a bank in a successive manor, so say

word 0 would belong to bank 0, word 1 to bank 1, up till bank 31; the total number of banks are thus 32 for all devices past the fermi architecture.[4] After the 31st word, we then wrap around to bank 0, so word 32 will be the second element in bank 0. This can be visualised below:

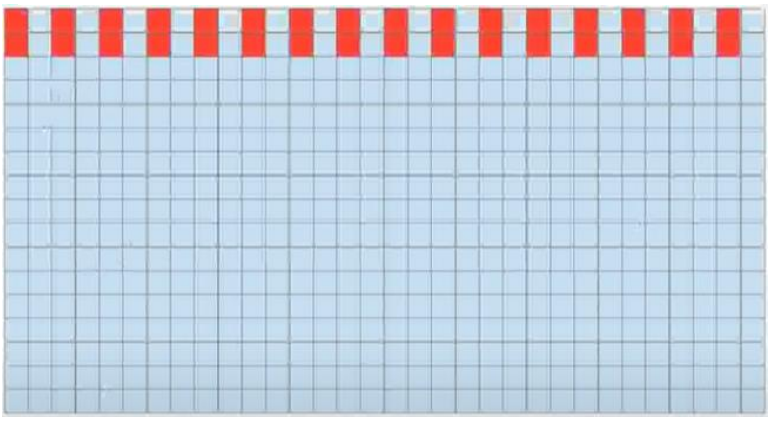
Figure 17 – Words and Banks division in shared memory [5][6]



A warp of threads can access any set of the 32 banks, and its constituent words. Although, shared memory performs fastest when there is one request for each bank per given warp, as such inefficiency occurs when threads of a warp request different values from the same bank in a single request; this requesting of different values from the same bank is referred to as a bank conflict. We can also see an increase in efficiency if we implemented a broadcast, in which all threads of warp request the same element you get an efficient read; similarly if you get a set of threads request the same value we have a multicast which is also efficient, however if two or more threads read from a bank per request we get a bank conflict.[5][6]

Therefore, under the implementation of CUDA banks we get the least number of bank conflicts when we either request elements sequentially (for example by purely using thread id) or accessing the same element through a broadcast. However, since in the kernel, outlined in figure 16, we calculate the index of the element using by 2 * threadID.x * Stride, we always get an even value for index (due to *2), therefore at each request we access 2 words from a given bank, and as such we get bank conflicts, as shown below (figure 17), which greatly decreases the performance.

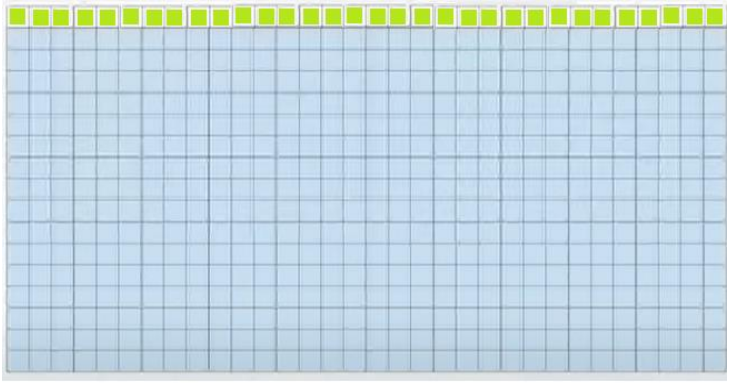
Figure 17 – Bank conflicts in figure 16[5][6]



Rows represent Banks (32 in length from 0 to 31), columns represent the words in banks

As such we need to alter the kernel outlined in figure 16, such that to access element sequentially, and as such remove any bank conflicts. So that the access pattern for the kernels look like below:

Figure 18 – No bank conflicts[6]



Rows represent Banks (32 in length from 0 to 31), columns represent the words in banks

As such we need to modify the kernel to ensure that we sequentially retrieve the elements, say using thread id. We can do this under the most optimised kernel, that does not have any thread divergence or bank conflicted:

Figure 19 – No bank or thread divergence [6]

```
//Reduce kernel for device that allows the reduction to take place
__global__ void reduceKernel(unsigned int * d_in,unsigned int* d_out) {
    int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
    int tid = threadIdx.x; // Local ID within the current block
    __shared__ unsigned int temp[BLOCK_SIZE];
    temp[tid] = d_in[myId];
    __syncthreads();
    // do reduction in shared memory
    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];
    }
}
```

The updated kernel now has no bank conflicts or thread divergence, this is because as mentioned before thread divergence occurs when different blocks of code operated based on some under lying condition such as say using the modulus operating on thread id in an if statement, which leads to divergence in execution of instructions of a thread in a warp. As such in the updated kernel we do not have alternating code for different threads, but only have threads that need to be active at a time access the correct element, using their given ids; essentially meaning there is no point at which we get threads separating from a common function, excluding extremes at where the thread id is greater than stride value. Under this approach only the threads in a warp that need to be active at a given time are active, and since there are no idle threads available.

In the updated kernel we also do not have bank conflicts since we are accessing elements based on their thread id and a given stride, and thus accessing them in a sequential manor, preventing any bank conflicts that may arise, as mentioned above. Under the new kernel we use reversed looping in which we start with elements further back at the array and move towards the first element in the array, whilst doing so ensuring that only all threads within the warps execute the same function. We can clearly see this approach in figure 20.

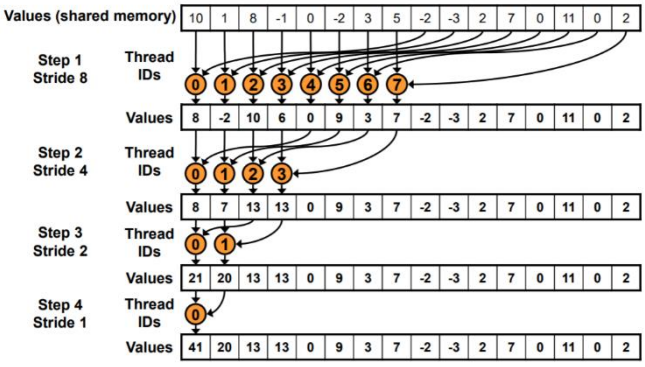


Figure 20 – Optimised kernel visualised [5]

Therefore, by altering the kernel to remove thread divergence and bank conflicts by switching to a gather approach over a scatter, we see significant improvements in the execution times of the given kernel.

5.3) Scan

Note: This section builds upon the ideas explored in the introduction and section 5.1,5.2, to prevent repeating information

Basics of scan

An inclusive scan operation takes a set/array of elements, along with a binary and associative operator and its associated identity element, to create essentially a forward sum of a given set of values, where a value at a given index is the sum of all the elements before it. We can have 2 different types of scan, an inclusive and exclusive scan, the main difference being that in an exclusive scan we the initial element of the output array is the identity element of the chosen operator, whereas in an inclusive scan the first element in the output array is the first element in the input array. We can further see that in an inclusive scan the final value of the output will be the total sum of the elements, whereas in an exclusive scan this is not the case; however we can easily convert between an inclusive and exclusive scans. As an example, an input array of [1,2,3,4], under the operation +, would be [1,3,6,10] in an inclusive scan.

We can easily implement this in a serial, non-parallel manor such by, using the below code, note by changing the order of line 4 and 5, we can change the below inclusive scan kernel to an exclusive scan kernel.

```
1  //Assuming operation "+", with identity "0"
2  int currentSumValue = 0; //Equal 0 since identity of "+" is 0
3  for (int i=0;i<sizeof(elements);i++){
4      currentSumValue = currentSumValue + elements[i];
5      out[i] = currentSumValue;
6  }
```

Figure 21 – Serial implementation of inclusive scan

However, transfer the serial version of scan to a CUDA device will not lead to any significant improvement in execution time, this being the implementation stated in figure 21, required the previous value of an array to calculate the next, and as such is not parallelizable. Therefore, to make the scan operation parallelizable, we use the Hillis and Steel algorithm, however minor additions are needed to the Hillis and Steel algorithm to account for the limitations/structures of CUDA GPUs.

Hillis and Steele Scan algorithm

The Hillis and Steele algorithm works by essentially starting off with a stride value of 1, where the stride value determines which values you add together, at each stage of the calculation, much like in reduce the stride value S, at a given stage X(starting at stage 1), can be calculated by:

$$S = 2^{X-1}$$

Based on a stride value, we decide which values to the left of an index we add onto a given index value, if no values are present to left of the index, we ignore that value, and make that the final result for that given array value. We successively increase the stride value based on the formula stated above for each stage, and constantly update indexed values, until we get to a point where the stride value is greater than or equal to the number of elements. This can be easily visualised with the diagram below:

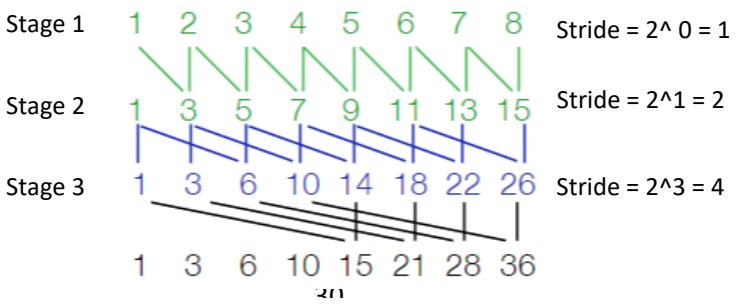


Figure 21 – Hillis and Steel inclusive scan[7]

For instance, at stage 1, we every element adds to itself a neighbouring value of stride 1 away to its left, if it does not have a neighbour that value is assumed to be the result. In stage 2, we increase the stride value to 2, and every element adds to itself the value 2 elements away to the left of that given value. Finally, on stage 3, the stride is 4, and every element adds to itself 4 elements to the left provided the element is present. At the end of stage 3, we cannot go further as the stride value is greater than or equal to the number of elements in the array, and as such there will be no change after at the next stage.

Hillis and Steele Scan algorithm in CUDA

Although the Hillis and Steel algorithm is simple to implement in CUDA, we need to modify it to match some of the limitations of CUDA devices. For example, as mention in the introduction of section 5 and 5.1, CUDA devices make coding easier by simplifying code into threads -> blocks -> grids, where each initial level makes up subsequent levels. When a CUDA program is initialized onto the CUDA device it causes the blocks to be executed on the Streaming Multiprocessors. Where a Streaming Multiprocessors, has the associated shared memory that is specific to a given block, along with CUDA cores that allow the execution of threads. However, depending on the specifications of the CUDA GPU, we have differing number of streaming multiprocessors available per device; for example, the GTX 970 has 13 SM's with 128 CUDA Cores [8]. Now since each Streaming Multiprocessor is responsible for executing a block, and there are a limited number of Streaming Multiprocessors, we cannot guarantee that all blocks will execute at a given time, since the systems scheduling diagram determines which block executes at a given time on the Streaming Multiprocessors. This therefore means that we can only ensure that all the blocks have finished executing at the end of the lifecycle of a kernel and cannot be synchronised from within the kernel itself. This however is not the case for threads, since a block contains all its associated threads, as such which can have appropriate break conditions for a given block.

Another limitation of the CUDA architecture is that we must ensure that the dimension of a given block x, y, z must be below 1024, 1024 and 64 respectively where $x * y * z \leq 1024$ must also hold. Similarly, we must ensure that blocks within a grid have dimensions x, y, z below 65535. This along with no block synchronisation within a kernel, means that if the number of elements in our initial scan array is greater than 1024, we will not be able to perform an inclusive scan. The reason for this is, that under Hillis and Steel, we need the previous set of elements to calculate the next set of elements, therefore we need all the elements to be accessible in memory to complete a scan operation. Therefore, as a result of the restriction on the size of a thread, we can only have 1024 elements, since any number of elements greater cannot be directly addressed from thread id, and by not doing so we lose the benefits of parallelism. We could in theory fix this issue if we had some mechanism of block synchronisation within a given kernel however this is not possible under current CUDA architecture.

Although we cannot have block synchronisation within a kernel, we can have block synchronisation outside of a kernel invocation, we can therefore perform multiple scans across blocks and combine them together, as such we can add additional steps to ensure the scan works across blocks; this however does not lead to a large decrease in performance since kernel invocations are generally efficient.

We can therefore perform cross block Hillis and steel using the following steps (which are clearly commented in the code outlined in section 3), note "|" indicates a separate block, since block size is 2:

Stage	Operation	Example
0	Initialise all elements to an array.	Elements = [1,2,0,3,3,2,4,5] Block Size = 2
1	Apply a scan across all the elements in a block, thus performing an incomplete scan, where there is no dependency to the overall block values.	Elements = [1,3|0,3|3,2|4,9]
2	Take the maximum values of each block value and store in a temporary array.	Elements = [1,3|0,3|3,2|4,9] Max-Elements = [3,3|2,9]

3	Apply a scan operation performed in stage 1, across the array of max elements from the previous stage; however, ensure that if we perform the scan sequentially, in that to ensure that the last element of one (left hand side stage 3) is added onto the first element of the second block, before performing the scan again (5).	Elements = [1,3|0,3|3,2|4,9] 1)Max-Elements = [3,3|2,9] 2) Max-Elements = [3,6|2,9] 3) Max-Elements = [3,6|6+2,9] 4) Max-Elements = [3,6|8,9] 5) Max-Elements = [3,6|8,17]
4	If the block is not the initial block, we add the max value at the index position of 1 before the current block's id value to all associated elements in the associated array of initial elements.	Elements = [1,3|0,3|3,2|4,9] Max-Elements = [3,6|8,17] New elements= [1,3|0+3,3+3|3+6,2+6|4+8, 9+17| =[1,3,3,6,9,8,12,26]

Even though in the steps outlined above, in step 3 we added the max elements at intervals of block dimensions it's often more efficient to add based on 1024 since that is theoretical max a scan kernel can do, this removes the overhead of the adding the last element in a block to the start element in another block, to absolute minimum. In the code outlined in section 3, this has been implanted, and a detailed commented explanation is provided above. We cab further use the efficiencies of parallelism to ensure that the entirety of the steps outlined above occur on the GPU, as such implemented in section 3.

Efficiencies of Hillis and Steele Kernel

The kernel for Hillis and Steele implements the methodology outlined above and shown in figure 21, and as such is shown below:

```
//Scan kerenel is applied to the initial set of values
__global__ void scanKernel(float *idata,int n,int startIndex)
{
    int thIdx = threadIdx.x + blockIdx.x * blockDim.x+startIndex;
    int tid = threadIdx.x;
    //Add appropriate data elements to shared memeory of the blocks
    __shared__ float temp[1024];
    temp[tid] = idata[thIdx];
    __syncthreads();

    // The following loop creats an initial scan as per the scan algorithm
    for (int offset = 1; offset < n; offset *= 2)
    {
        if (tid >= offset)
            temp[tid] += temp[tid - offset];
        __syncthreads();
    }
    idata[thIdx] = temp[tid];
}
```

Figure 22 – Hillis and Steel inclusive scan Kernel

Under this implementation of the kernel, we see no major thread divergence, because as stated before, thread divergence occurs when there are conditional statements based on some underlying value, such as thread id, determining which section of the code should be executed. So, under the

current implementation there is only minor thread divergence when the thread id Is larger than the offset value.

Another implementation of an inclusive scan is to use the Blelloch scan, which is another implementation of parallel scan. The Blelloch scan can be considered more work-efficient but is considerably less step efficient, meaning it is ideal for large amounts of elements that require a low computationally expensive operation. Whereas Hillis and Steel algorithm is much more step efficient and as such is more useful when the operation in a scan are not computationally efficient. As such we can alter an overall scan kernel to switch between the algorithms to efficiently compute a scan depending on its input values.

5.4) Histogram

Note: This section builds upon the ideas explored in the introduction and section 5.1,5.2,5.3 to prevent repeating information

Basics of the Histogram operation

The histogram program outlined in section 4, allows the generation of a histogram form an input set of elements, the algorithm will have a set number of bins in which if a given element belongs to the range of a bin we increase the counter of that bin. In the implementation outlined above we decided if a given element is part of a bin using the modulus operator, where in the modulus of a given element with respect to the max number of bins determined which bin value we increment. We can do this easily under the serial implementation stated below:

```
int BIN_SIZE = 8;
int elements[] = {1,2,3,4,5,6};//Initialising values
int histogram[BIN_SIZE];
// Initialising Histogram values to 0
for(int x=0;x<BIN_SIZE;x++){
    histogram[x]=0;
}

//Incrementing histogram values
for (int x = 0;x<sizeof(elements);x++){
    //Mod operator assingins values to bins
    int binPosition = elements[x] % BIN_SIZE;
    histogram[binPosition]++;
}
```

Figure 23 – Histogram Serial Kernel

As is evident form the code, this implementation requires the previous set of bin values to calculate the next set of values, and as such requires sequential updating of the array of histogram values. This principle cannot be overturned, and as such when altering the code for a parallel implementation, we must ensure that we have no race conditions that prevent this from occurring.

Due to the architecture of CUDA race conditions are often an issue when writing values to arrays. Race conditions occur when threads try and update the same element at a given time, and as such we get a result when a thread updates a given data to an incorrect value. For example if thread 1 tries to update a variable x, which initially has a value 10, to one greater than the current value of x, and thread 2 tries to update the value of the variable to 2 greater than the current value, thread 1 will read the value of x as 10, and thread 2 will read the value as 10, and both will update the value to their respective results of 11 and 12 and the same time, resulting in an incomplete answer, that will purely depend on which thread last updated the value. This kind of error is even more prevalent in CUDA, since the core principle of CUDA requires multiple threads working at a given time across multiple blocks and streaming multiprocessors.

Therefore, the only way to prevent race conditions is to ensure that we serialise the updating of elements that are constantly accessed and updated by multiple threads. In CUDA we can achieve this by using an atomic add operation, where an atomic add operation ensures that if 2 or more threads try and update a given element, they are only able to update the elements one after the other. However aggressively using atomic add leads to significant performance degradation, as it forces kernels to run serially, and as such we lose the benefits of parallelism. A basic parallel kernel that does implements this can be seen below:

```
__global__ void simple_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    //The below method is just one of MANY ways to allocate elements to bins
    // Below is purley just an illustation
    int myBin = myItem % bin_count;
    //The below condition with additonal blocks sizes above allow the method
    // to function with any number of elements and NOT just multiples of BLOCK_SIZE
    if (myId < N)
        atomicAdd(&(d_bins[myBin]), 1);
}
```

Figure 24 – Basic histogram kernel

However, the kernel in figure 24, is extremely inefficient as is evident from figure 8, the main reason for this is the above kernel performs the atomic add on global memory than on shared memory. As explored in section 5 and 5.1, global memory has significantly lower read and write speed than shared memory and therefore by performing any operation on global memory is significantly slower than on shared memory, this particularly holds true for atomic add. This is particularly true for atomic add since, we are forcing the kernel to sequentially update the elements, and if memory access is slower it takes a longer time to update the elements under atomic add. We can therefore update the kernel to utilise shared memory to be:

```
84  __global__ void shared_memory_histogram(unsigned int *d_bins, const unsigned int *d_in, const unsigned int bin_count,unsigned int N)
85  {
86      int myId = threadIdx.x + blockDim.x * blockIdx.x;
87      int tid = threadIdx.x;
88      //Assinges approrpatie memory methods
89      __shared__ unsigned int bins[BIN_COUNT];
90      unsigned int myItem = d_in[myId];
91      //Each shared memory location is passed in a value that can be used.
92      unsigned int itemBinId = myItem % bin_count;
93      bins[itemBinId] = 0;
94      __syncthreads();
95      //Method for adding up the sum in bin values
96      if (myId < N){
97          atomicAdd(&(bins[itemBinId]), 1);
98      }
99      __syncthreads();
100
101     //If the thread id of a given block is 0, then it adds all the item values to the output array
102     if(tid==0){
103         for(unsigned int x=0;x<BIN_COUNT;x++){
104             atomicAdd(&(d_bins[x]), bins[itemBinId]);
105         }
106     }
107
108 }
```

Figure 25 – Shared memory histogram kernel

The kernel outlined in figure 25, first stores the individual elements to registers associated to each thread, and then creates a common shared memory array that is accessible to all appropriate threads in a block. We then decide which bin a given element belongs to and perform an atomic add operation on the shared memory element. After all the threads have finished updated the shared memory data set, the 1st thread in a block performs an atomic add operation to the global memory array that stores the bin values. We can see from figure 8, that even with 2 sets of atomic add operations, the write speed of shared memory is significantly fast such that we outperform a purely global memory implementation that has less atomic add operations (as shown in figure 24). By increasing the number of bins available we can see the negative effect of atomic add operations, since by increasing the number of bins we see less race conditions, and as such we would see an increased execution time.

The kernels outlined in figure 25 does not have any block conflicts since all the threads access the associated global memory elements sequentially rather than cause blocking as described in section 5.2. We further see that there is very little thread divergence that occurs in the kernel, since there are no main branching conditions. The 2 minor thread divergence that do occur however are as a result of line 96, where at the extremes of the number of elements in the array we see divergence. We also see a minor thread divergence in line 102, since every thread apart from first in a warp are inactive under the current implementation of the kernel.

References

[1]"Georgia Tech - Thread Divergence", *Frc.gatech.edu*, 2020. [Online]. Available: http://frc.gatech.edu/wp-content/uploads/sites/550/2016/12/ControlFlow-I.pdf. [Accessed: 08- May- 2020]

[2]"Memory Statistics - Caches", *Docs.nvidia.com*, 2020. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticscaches.htm. [Accessed: 08- May- 2020]

[3]*Bt.nitk.ac.in*, 2020. [Online]. Available: https://bt.nitk.ac.in/c/17b/co332/notes/6-Tiled-MM.pdf. [Accessed: 09- May- 2020]

[4]*Surreylearn.surrey.ac.uk*, 2020. [Online]. Available: https://surreylearn.surrey.ac.uk/d2l/le/content/188692/viewContent/1766840/View. [Accessed: 09- May- 2020]

[5]A. Heinecke, M. Klemm and H. Bungartz, "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture", *Computing in Science & Engineering*, vol. 14, no. 2, pp. 78-83, 2012.

[6]"Bank Conflicts", 2020. [Online]. Available: https://www.youtube.com/watch?v=CZgM3DEBplE&t=946s. [Accessed: 10- May- 2020]

[7]*Surreylearn.surrey.ac.uk*, 2020. [Online]. Available: https://surreylearn.surrey.ac.uk/d2l/le/content/188692/viewContent/1772404/View. [Accessed: 11- May- 2020]

[8]"GEFORCE GTX 900 SERIES GRAPHICS CARDS", *Nvidia.com*, 2020. [Online]. Available: https://www.nvidia.com/en-gb/geforce/900-series/. [Accessed: 11- May- 2020]

Appendix

Appendix 1

```cpp
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#include <ctime>
#include <iostream>

#define BLOCK_SIZE 32
float gpuRecursiveReduce(float * d_in, int N);
__global__ void reduceKernel(float *d_out, float *d_in);
__global__ void summationKernel(float *d_out, float *d_in,const int N);
void cpuReduce(float * h_in,int N);
void gpuReduce(float * h_in,int N);
void cpuReduction(float *arr, int size, float gpuTime);
void printArray(float *arr, int sizeOfArray);


int main(){
    for (int k = 1; k < 35; ++k)
    {
        //Initialising the values
        int N = (1 << k);

        float * h_in = new float[N];
        for (int x=0; x<N;x++){
            h_in[x]=1.0f;
        }
        //Method for doing entier function on GPU
        gpuReduce(h_in,N);

        //Method for doing entier function of CPU
        cpuReduce(h_in,N);
    }
    return 0;
}

//Method for ensuring cpu reduce
void cpuReduce(float * h_in,int N){

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

    bool debug =false;
    size_t size = N * sizeof(float);
    //How many blocks should be created, provided that each block has 1024 threads
    int GRID_SIZE = ((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
    // The above calculation makes it such that the correct amount of blocks are
    //Generated when we assume each block has 1024 threads in them.
    size_t size_o = GRID_SIZE * sizeof(float);
    // Define the array for host device
    float h_out[GRID_SIZE];
    // Define the arrays for the cuda device(GPU)
    float *d_in;
    float *d_out;


    cudaMalloc((void **)&d_in, size);
    //The values are copied from the host array, to the device array.
    cudaMemcpy(d_in, h_in, size, cudaMemcpyHostToDevice);
    //We allocate glbal memory to out array on the cuda device
    cudaMalloc((void **)&d_out, size_o);

    // thread size is a calculation of how many threads there will be in a given
block
    if(debug){
        printf("INITITAL: Block dimensions(Threads in a block): %d\n", BLOCK_SIZE);
        printf("INITIAL: Grid dimensions(Blocks in a grid): %d\n", GRID_SIZE);
    }

    //Defining the size of the block
    dim3 blockDim(BLOCK_SIZE);
    //Defiing the size of the grid
    dim3 gridDim(GRID_SIZE);

    //Starting timing of cuda kernel
    cudaEventRecord(start);
    //Starting the kernel
    summationKernel<<<gridDim, blockDim>>>(d_out, d_in,N);
    // Wait for GPU to finish before accessing on host
    err = cudaDeviceSynchronize(); //Ensure all the blocks are finished executing
    //Stop the event timing recording as well as present appropraite errors
    cudaEventRecord(stop);
    if(debug)
        printf("Run kernel: %s\n", cudaGetErrorString(err)); // FOr debugging

    //Moving the results form the GPU to cPU
    err = cudaMemcpy(h_out, d_out, size_o, cudaMemcpyDeviceToHost);
    if(debug)
        printf("Copy h_C off device: %s\n",cudaGetErrorString(err)); //For debugging

    //Print the elapsed time of the kernel
    cudaEventSynchronize(stop);
    float totalExecutionTime = 0;
    cudaEventElapsedTime(&totalExecutionTime, start, stop);
    // printf("\n\nElapsed GPU time was: %f milliseconds\n", milliseconds);
    if(debug)
        printArray(h_out,GRID_SIZE);

    cpuReduction(h_out, GRID_SIZE, totalExecutionTime);

}
```

```
102
103 //Method for esur
104 void gpuReduce(float * h_in,int N){
105     //Allocating appropaite space on GPU
106     float * d_in;
107     cudaMalloc(&d_in,sizeof(float)*N);
108     cudaMemcpy(d_in,h_in,sizeof(float)*N,cudaMemcpyHostToDevice);
109     float total = gpuRecursiveReduce(d_in,N);
110     printf("\n\nGPU: Total For %d is %f \n",N,total);
111     // std::cout << "CPU time: " << duration << " ms" << std::endl;
112
113 }
114
115 float gpuRecursiveReduce(float * d_in, int N){
116     //Approrpaite method for timing
117     cudaEvent_t start, stop;
118     cudaEventCreate(&start);
119     cudaEventCreate(&stop);
120     float totalSum=0.0f;
121     //Calculating number of blocks in a given grid
122     int grid_size=((N + BLOCK_SIZE - 1) / BLOCK_SIZE);
123
124     //Appropriatly allocate memory for the recursive call
125     float * d_blocks;
126     cudaMalloc(&d_blocks,sizeof(int)*grid_size);
127     cudaMemset(d_blocks, 0, sizeof(float) * grid_size);
128
129     //Call the kernel to further reduce the data by using the reduction kernel
130     float temp=0;
131     cudaEventRecord(start);
132     reduceKernel<<<grid_size,BLOCK_SIZE>>>(d_blocks,d_in);
133     cudaEventRecord(stop);
134     cudaDeviceSynchronize();
135     cudaEventElapsedTime(&temp, start, stop);
136     printf("T:%f",temp);
137
138     //The recursion occurs untill the number of elements that can be executed is the size of a
139 block
    if(grid_size<=BLOCK_SIZE){
140         float* d_total;
141         cudaMalloc(&d_total,sizeof(float));
142         cudaMemset(&d_total,0,sizeof(float));
143         //Method for measuring execution time as well as calling the kernel
144         cudaEventRecord(start);
145         reduceKernel<<<1,BLOCK_SIZE>>>(d_total,d_blocks);
146         cudaEventRecord(stop);
147         cudaDeviceSynchronize();
148         cudaEventElapsedTime(&temp, start, stop);
149
150         //Coppying results to approrpaite memory location
151         cudaMemcpy(&totalSum, d_total, sizeof(float), cudaMemcpyDeviceToHost);
152         cudaFree(d_total);
153     }
154     else{
155         //Recursivly calls the reduce function if the elements dont fit in a reduction block
156         float * d_in_block_sums;
157         cudaMalloc(&d_in_block_sums, sizeof(float) * grid_size);
158         cudaMemcpy(d_in_block_sums, d_blocks, sizeof(float) * grid_size, cudaMemcpyDeviceToDevice);
159         totalSum = gpuRecursiveReduce(d_in_block_sums, grid_size);
160         cudaFree(d_in_block_sums);
161     }
162     cudaFree(d_blocks);
163     return totalSum;
164 }
165
166
167 __global__ void reduceKernel(float *d_out, float *d_in)
168 {
169     int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
170     int tid = threadIdx.x;              // Local ID within the current block
171     __shared__ float temp[BLOCK_SIZE];
172     temp[tid] = d_in[myId];
173     __syncthreads();
174     // do reduction in shared memory
175     for (unsigned int s = blockDim.x / 2; s >= 1; s >>= 1)
176     {
177         if (tid < s)
178         {
179             temp[tid] += temp[tid + s];
180         }
181         __syncthreads(); // make sure all adds at one stage are done !
182     }
183     // only thread 0 writes result for this block back to global memory
184     if (tid == 0)
185     {
186         d_out[blockIdx.x] = temp[tid];
187     }
188 }
189
```

```
189
190 //Minor change to work for CPU reduction
191 __global__ void summationKernel(float *d_out, float *d_in,const int N)
192 {
193     int myId = threadIdx.x + blockDim.x * blockIdx.x; // ID relative to whole array
194     int tid = threadIdx.x;             // Local ID within the current block
195     __shared__ float temp[BLOCK_SIZE];
196
197         temp[tid] = d_in[myId];
198
199         __syncthreads();
200         // do reduction in shared memory
201         for (unsigned int s = blockDim.x / 2; s >= 1; s >>= 1)
202         {
203             if (tid < s && myId<=N)
204             {
205                 temp[tid] += temp[tid + s];
206             }
207             __syncthreads(); // make sure all adds at one stage are done !
208         }
209
210     // only thread 0 writes result for this block back to global memory
211     if (tid == 0)
212     {
213         d_out[blockIdx.x] = temp[tid];
214     }
215 }
216
217 //Method for recusivly calling the final reduction
218 void cpuReduction(float *arr, int size, float gpuTime)
219 {
220     float final_reduction = 0.0f;
221     clock_t start = clock();
222     clock_t tStart = clock();
223     /* Do your stuff here */
224
225     for (int i = 0; i < size; i++)
226     {
227         final_reduction += arr[i];
228     }
229     // Recordingend time.
230     //Calculating the actual total difference
231     clock_t stop = clock();
232     double time_taken = (double)(stop - start) * 1000.0 / CLOCKS_PER_SEC;
233
234     printf("CPU: sumation %fin a time %f milliseconds\n\n", final_reduction,time_taken +
    double(gpuTime));
235 }
236
237 //Method for printing the data in array
238 void printArray(float *arr, int sizeOfArray)
239 {
240         printf("Printing %d values \n", sizeOfArray);
241         for (int i = 0; i < sizeOfArray; i++)
242         {
243             printf("%f|", arr[i]);
244             if((i+1)%BLOCK_SIZE==0){
245                 printf("\t");
246             }
247         }
248         printf("\n Fin printing");
249
250     printf("\n");
251 }
252
253
```