COM2039 Parallel Programming using NVIDIAs CUDA

Content

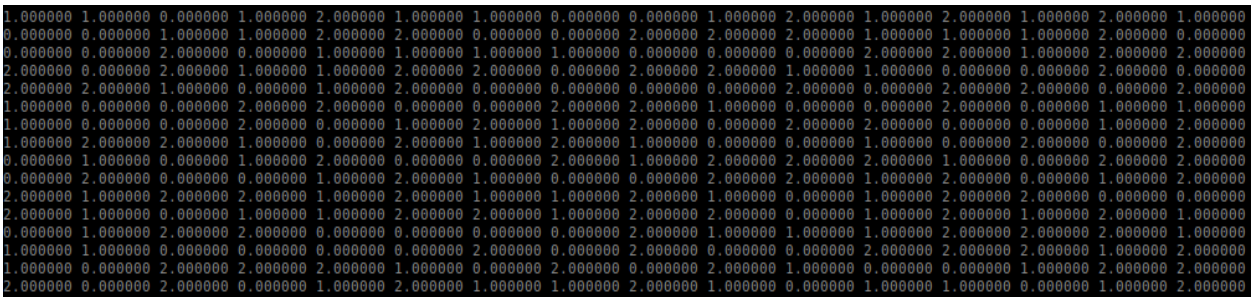
I confirm that this report is my own work and any work that has been used have been referenced appropriately.

Signed by: Nithesh Koneswaran

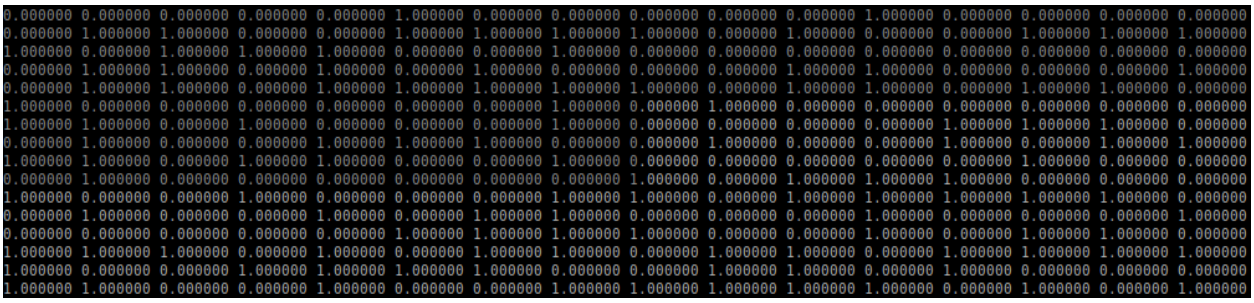
Part 1: Matrix Multiplication

Step 1 [5 marks]: Examples of matrices from matrix multiplication program that uses global memory

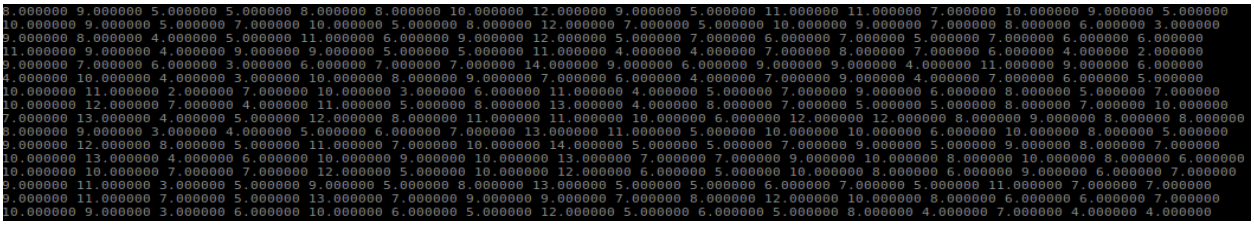
Input A



Input B

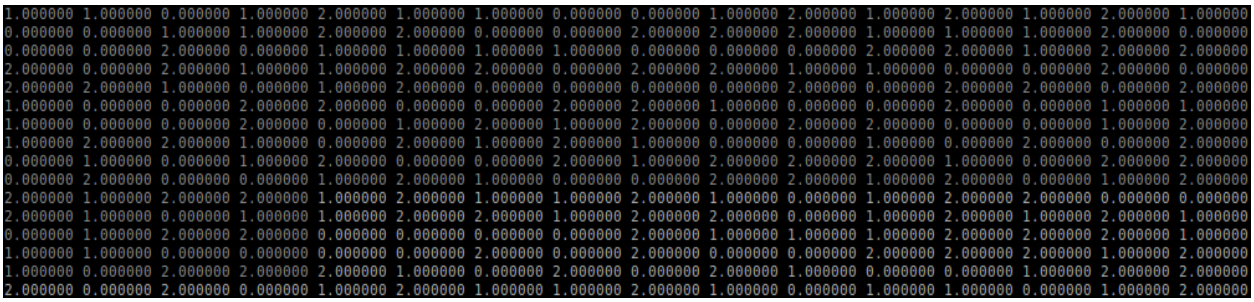


Input C

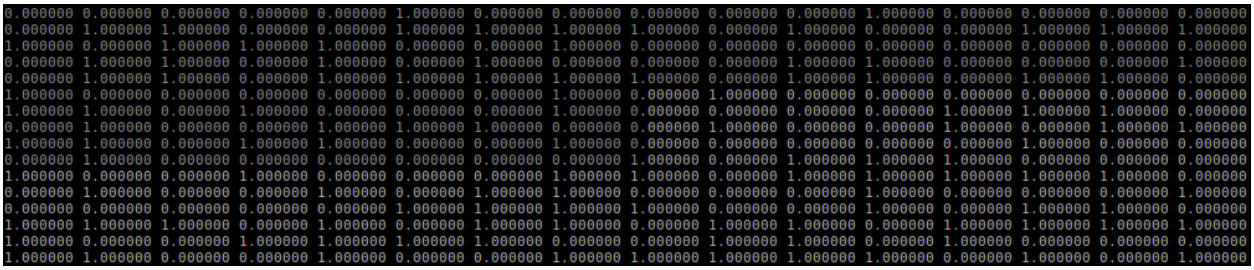


Examples of matrices from matrix multiplication program that uses shared memory.

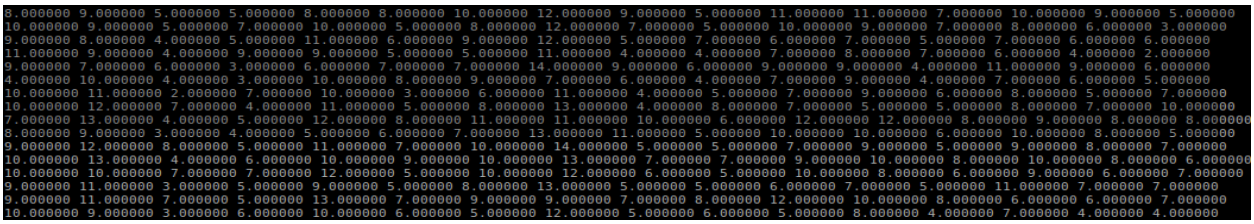
Input A



Input B



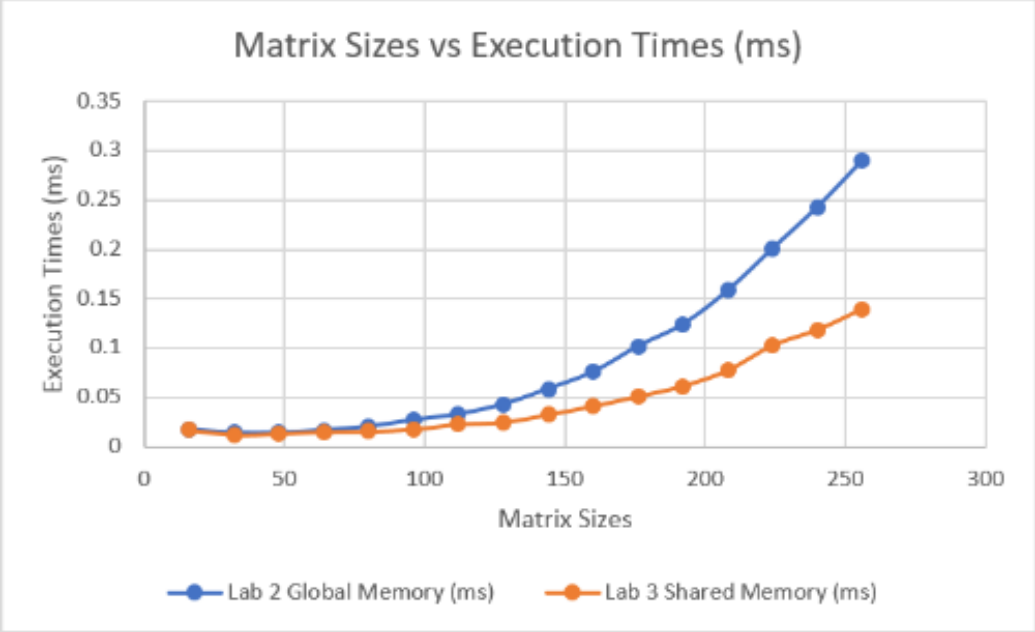
Input C



Step 2 [3 marks]: Added timing event for lab class 3 and 2

Step 3 [8 marks]: Progressively scaling the matrix size

Matrix Size	Lab 2 Global Memory (ms)	Lab 3 Shared Memory (ms)
16	0.017376	0.016992
32	0.014144	0.012288
48	0.014336	0.01312
64	0.016704	0.01536
80	0.02048	0.015616
96	0.027104	0.017728
112	0.032992	0.022944
128	0.043008	0.02496
144	0.058784	0.032512
160	0.075872	0.04112
176	0.101888	0.0512
192	0.124032	0.061568
208	0.158368	0.077568
224	0.200896	0.102784
240	0.24288	0.118432
256	0.290176	0.139616
512	2.349824	1.03744
1024	19.199137	8.708096
2048	153.854813	68.687965
4096	1249.886719	419.366058
8192	13592.4707	3572.080566



Step 4 [9 marks]: Results of the experiment
Note: I tested the execution time for matrices of size 16 to 256 with intervals of 16. In the graph above I ignored matrices with size greater than 256 since the trend was quite clear, the trend being that the program that utilized shared memory had a faster execution time than the program that only used global memory.

Hypothesis: I theorize that as the sizes of the matrices increases the program that uses shared memory would have a faster execution time than the program that uses global memory. This is expected since shared memory (comparable to L1 Cache in a CPU) sits on the streaming multiprocessors providing short access times compared to global memory which sits on the device, off the chip, providing longer access times. Upon further research [1] I found that an ‘SRAM cell’ (Shared memory) can be clocked much faster than ‘DRAM cell’ (Global memory) which further backs up my hypothesis that shared memory would perform faster than global memory. A limitation of this experiment is that memory is not infinite, in the Otter labs the computers running GTX 1050ti have 4GB of global memory and 49152 bytes of shared memory, because of this we might find that either shared memory or global memory will run out if the matrix set size exceeds a large number.

The output data, shown in step 3, clearly supports my hypothesis, if we look at execution times for matrices with size 8192 we find that shared memory (3672.08056ms) has a 73.72% decrease in execution time compared to Global memory (13592.4707ms). We can clearly see that Global Memory has a slower execution time compared to Shared Memory, this is because of the physical architecture of the GPU as explained in the hypothesis.

I also noticed that for both programs the execution times seem to get quicker between the range of matrix sizes of 16 to 64. Upon research I ultimately concluded that it may have gone faster due to coalescing memory access which can boost access speeds, an alternative conclusion is that it could simply be an anomaly on the results since the execution times in the labs can be quite different for each execution of the program, there should be a period to wait before executing again. Nevertheless, upon further testing the pattern remained the same.

Part 2: Reduce

Step 1 [15 marks]: Implementation of reduce

```cpp
#include <iostream>
#include <numeric>
#include <stdio.h>
#include <stdlib.h>
#define ARRAY_SIZE 1024
#define BLOCK_SIZE 32
/*
 *   Efficient code optimisation taken from NvideaL:
 *   https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
 */



/*
 * Warp: is a set of threads that all share the same code
 *       they follow the same execution path with minimal divergences
 *       stalls at the same places
 *
 *       instructions are SIMD (single instructions multiple data) synchronous within a Warp0.081248

 *
 *       That means that when thread index is less than 32, we dont need to sync threads
 *       and we do not need "if (local_index < halfOfBlockSize)" since it doesn't save any work
 *       because of this we can simply reduce as shown below.
 *
 *       Quick Note: _Syncthreads is a block wide barrier, used to avoid shared memory race conditions
 *
 */

__device__ void warpReduce(volatile int* mem, int local_index) {
  mem[local_index] += mem[local_index + 32];
  mem[local_index] += mem[local_index + 16];
  mem[local_index] += mem[local_index + 8];
  mem[local_index] += mem[local_index + 4];
  mem[local_index] += mem[local_index + 2];
  mem[local_index] += mem[local_index + 1];
}
```

```
/*
 * The Lab code had a slight inefficiency;  during the first iteration of reduction in the for loop
 * half of the threads are left idle. In order to solve this we can simply perform the first
 * stage of the reduction during the loading process
 *
 * This is a kernal which will reduce values
 */
__global__ void shared_reduce_kernal(float* d_out, float* d_in) {
  //Gets the position of the thread
  int global_index = threadIdx.x + (blockDim.x*2)* blockIdx.x; // Global ID
  //Gets the index of the thread
  int local_index = threadIdx.x; // Local ID
  //defined amount of shared memory, specified in the Host Code
  //Since we are dealing with half the block size
  __shared__ int mem[BLOCK_SIZE/2];

  /** Copies all values from global memory to shared memory
   *  Remember each block has their own shared memory
   *  that is why we assign position index with shared memory's thread index
   *  example:
   *  thread with index 0 will have a another value compared to
   *  thread with index 0 ins another block
   *
   *  This code will perform the first iteration of the reduction,
   */
  mem[local_index] = d_in[global_index]+d_in[global_index+blockDim.x];
//  mem[local_index] = d_in[global_index];
  /** This makes sure each thread has loaded their values copied from global memory*/
  __syncthreads();
  /*
   * Each iteration of the loop reads n items from shared memory
   * and will write to n/2 items to shared memory
   *
   * We essentially divide the block in 2 and add each elements to its neighbour (block_size/2) elements away.
   * The results are stored in the first half of the block size.0.081248

   * We now take the first block size and repeat the process.
   * Until finally the left most element contains the full reduction of all elements in the thread block.
   *
   * halfOfBlockSize>>=1 will reduce the block size by half, since we are dealing with values ....512, 256, 12
   * We can simply use the left shift operation to effectively divide the given block in 2
   *
   * In this first iteration of the loop we see that only half of the threads are idle. This is quite wasteful
   */
  for (unsigned int halfOfBlockSize = blockDim.x/2; halfOfBlockSize > 32; halfOfBlockSize >>= 1)
    if (local_index < halfOfBlockSize)
      mem[local_index] += mem[local_index + halfOfBlockSize];
      /* makes sure that any threads that had completed the step waits for all the other threads to complete
       * the same step before proceeding to the next step */
      __syncthreads();
```

```
89
90    if (local_index<32)
91      warpReduce(mem, local_index);
92
93      /* The thread with the index 0 will contain the full reduction of all elements in the block
94       * Writes the value from thread 0 back into global memory depending on what block the thread is in and its
95        thread index */
96    if (local_index == 0)
97      d_out[blockIdx.x] = mem[local_index];
98
99  }
100
101 /* Final reduction step is used to serially reduce the given aarray*/
102 float finalReduction(float *h_out, int grid_size) {
103   /* Final reduce step done serially */
104   float final_reduction = 0.0f;
105   /* Iteratively adds each item in the array together */
106   for (int i = 0; i < grid_size; i++) {
107     final_reduction += h_out[i];
108   }
109   /* Final reduced value */
110   printf("Final reduction: %f\n", final_reduction);
111   return final_reduction;
112 }
113
114
115 int main(void) {
116   /* Size defined by multiplying the N by the size of the float */
117   const size_t INPUT_SIZE = ARRAY_SIZE * sizeof(float);
118   const size_t OUTPUT_SIZE =INPUT_SIZE/ BLOCK_SIZE;
119
120   //Creating Array size of 512
121   float h_in[ARRAY_SIZE];
122   float h_out[ARRAY_SIZE / BLOCK_SIZE];
123
124   //Output Array of size of 64 (512/32)
125   float *d_in, *d_out;
126
127   /** Start time event */
128   cudaEvent_t start, stop;
129   cudaEventCreate(&start);
130   cudaEventCreate(&stop);
131
132   /** Initialising our input array with float value 1.0f*/
133   for (int i = 0; i < ARRAY_SIZE; i++) {
134     h_in[i] = 1.0f;
135   }
```

```
136
137     /* Allocates memory for d_in in the GPU */
138     cudaMalloc((void**) &d_in, INPUT_SIZE);
139     /* Copies the array from HOST (cpu) to DEVICE (gpu) */
140     cudaMemcpy(d_in, h_in, INPUT_SIZE, cudaMemcpyHostToDevice);
141     /* Allocates memory for d_out in the GPU */
142     cudaMalloc((void**) &d_out, OUTPUT_SIZE);
143
144     /* Number of blocks per grid */
145     const int GRID_SIZE = ARRAY_SIZE / BLOCK_SIZE;
146     /* Number of threads per block, we divided it by 2 such that  */
147     dim3 threadsPerBlock(BLOCK_SIZE/2);
148     /* Number of blocks*/
149     dim3 blocks(GRID_SIZE);
150
151     /* Start the timer */
152     cudaEventRecord(start);
153
154     /* Launching kernal with 64 block and 32 threads per block */
155     shared_reduce_kernal<<<blocks, threadsPerBlock>>>(d_out, d_in);
156
157     /* cudaDeviceSynchronize(): halts execution in the CPU thread
158      * until the GPU has finished processing all previously requested
159      * cuda tasks.
160      *
161      * Wait for GPU to finish before accessing on host
162      */
163     cudaDeviceSynchronize();
164
165     /* Prints the input array
166     printf("Input Array: \n");
167     for (int n = 0; n < ARRAY_SIZE; n++) {
168       printf("%f ", h_in[n]);
169     }
170     printf("\n");
171
172     */
173
174     /* Copies the output array from device on to the Host's output array */
175     cudaMemcpy(h_out, d_out, OUTPUT_SIZE, cudaMemcpyDeviceToHost);
176
177     /* Do final reduce operation */
178     finalReduction(h_out, GRID_SIZE);
179
180     /* Outputs the results
181     for (int n = 0; n < GRID_SIZE; n++) {
182       printf("%f ", h_out[n]);
183     }
184     printf("\n");
185     */
186
187     /* Stops the timer*/
188     cudaEventRecord(stop);
189     cudaEventSynchronize(stop);
190     float milliseconds = 0;
191     cudaEventElapsedTime(&milliseconds, start, stop);
192     printf("Elapsed time was: %f milliseconds \n", milliseconds);
193
194     /* Frees the variables */
195     cudaFree(d_in);
196     cudaFree(d_out);
197 }
```

Program Reduce done on GPU:

```cpp
#include <iostream>
#include <numeric>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define ARRAY_SIZE 64
#define BLOCK_SIZE 32

/*
 * Warp: is a set of threads that all share the same code
         they follow the same execution path with minimal divergences
         stalls at the same places
Excel
         instructions are SIMD (single instructions multiple data) synchronous within a
          Warp

         That means that when thread index is less than 32, we dont need to sync threads
         and we do not need "if (local_index < halfOfBlockSize)" since it doesn't save
         any work


 *
 */
__device__ void warpReduce(volatile int* mem, int local_index) {
  mem[local_index] += mem[local_index + 32];
  mem[local_index] += mem[local_index + 16];
  mem[local_index] += mem[local_index + 8];
  mem[local_index] += mem[local_index + 4];
  mem[local_index] += mem[local_index + 2];
  mem[local_index] += mem[local_index + 1];

}


/*
 * This code has a slight inefficiency;  during the first iteration of reduction in
 * the for loop half of the threads are left idle. In order to solve this we can simply
 * perform the first stage of the reduction during the loading process
 */
__global__ void shared_reduce_kernal(float* d_out, float* d_in) {
  //Gets the position of the thread
  int global_index = threadIdx.x + (blockDim.x*2)* blockIdx.x; // Global ID
  //Gets the index of the thread
  int local_index = threadIdx.x; // Local ID
  //defined amount of shared memory, specified in the Host Code
  //Since we are dealing with half the block size
  __shared__ int mem[BLOCK_SIZE/2];

```

```
    /** Copies all values from global memory to shared memory
     *  Remember each block has their own shared memory
     *  that is why we assign position index with shared memory's thread index
     *  example:
     *  thread with index 0 will have a another value compared to
     *  thread with index 0 ins another block
     *
     *  This code will perform the first iteration of the reduction,
     */


    mem[local_index] = d_in[global_index]+d_in[global_index+blockDim.x];
  //  mem[local_index] = d_in[global_index];
    /** This makes sure each thread has loaded their values copied from global memory*/
    __syncthreads();
    /*
     * Each iteration of the loop reads n items from shared memory
     * and will write to n/2 items to shared memory
     *
     * We essentially divide the block in 2 and add each elements to its neighbour (block_size/2) elements away.
     * The results are stored in the first half of the block size.
     * We now take the first block size and repeat8 the process.
     * Until finally the left most element contains the full reduction of all elements in the thread block.
     *
     * halfOfBlockSize>>=1 will reduce the block size by half, since we are dealing with values ....512, 256, 128, 64,
     * We can simply use the left shift operation to effectively divide the given block in 2
     *
     * In this first iteration of the loop we see that only half of the threads are idle. This is quite wasteful
     */
    for (unsigned int halfOfBlockSize = blockDim.x/2; halfOfBlockSize > 32; halfOfBlockSize >>= 1)
      if (local_index < halfOfBlockSize)
        mem[local_index] += mem[local_index + halfOfBlockSize];
        /* makes sure that any threads that had completed the step waits for all the other threads to complete
         * the same step before proceeding to the next step */
      __syncthreads();

    if (local_index<32)
      warpReduce(mem, local_index);

      /* The thread with the index 0 will contain the full reduction of all elements in the block
       * Writes the value from thread 0 back into global memory depending on what block the thread is in and its thread index */
    if (local_index == 0)
      d_out[blockIdx.x] = mem[local_index];

}


/**
 *  This does the same as shared_reduce_kernal, however it does atomic add
 *  function at the end of the reduction.
 *
 *  I believe this is slightly more optimised than adding atomic add function
 * during the first kernal  because we are applying atomicAdd to much more less
 * array elements since we essentially reduce the values again.
 *
 */
__global__ void final_reduce_kernal(float* d_out, float* d_in) {
  //Gets the position of the thread
  int global_index = threadIdx.x + (blockDim.x*2)* blockIdx.x; // Global ID
  //Gets the index of the thread
  int local_index = threadIdx.x; // Local ID
```

```
109
110     __shared__ int mem[BLOCK_SIZE/2];
111
112     mem[local_index] = d_in[global_index]+d_in[global_index+blockDim.x];
113
114     __syncthreads();
115
116     for (unsigned int halfOfBlockSize = blockDim.x/2; halfOfBlockSize > 32; halfOfBlockSize >>= 1)
117       if (local_index < halfOfBlockSize)
118         mem[local_index] += mem[local_index + halfOfBlockSize];
119       __syncthreads();
120
121     if (local_index<32)
122       warpReduce(mem, local_index);
123
124
125     if (local_index == 0)
126       atomicAdd(&d_out[0], mem[0]);
127
128   }
129
130
131
132   int main(void) {
133     /* Size defined by multiplying the N by the size of the float */
134     const size_t INPUT_SIZE = ARRAY_SIZE * sizeof(float);
135     const size_t OUTPUT_SIZE =INPUT_SIZE/ BLOCK_SIZE;
136
137
138     const int SECOND_ARRAY_SIZE = ARRAY_SIZE/BLOCK_SIZE;
139
140     //Creating Array size of 512
141     float h_in[ARRAY_SIZE], final_h_in[SECOND_ARRAY_SIZE];
142     float h_out[ARRAY_SIZE / BLOCK_SIZE], final_h_out[SECOND_ARRAY_SIZE];
143
144     //Output Array of size of 64 (512/32)
145     float *d_in, *d_out, *final_d_in, *final_d_out;
146
147     /** Start time event */
148     cudaEvent_t start, stop;
149     cudaEventCreate(&start);
150     cudaEventCreate(&stop);
151
152     /** Initialising our input array with float value 1.0f*/
153     for (int i = 0; i < ARRAY_SIZE; i++) {
154       h_in[i] = 1.0f;
155     }
```

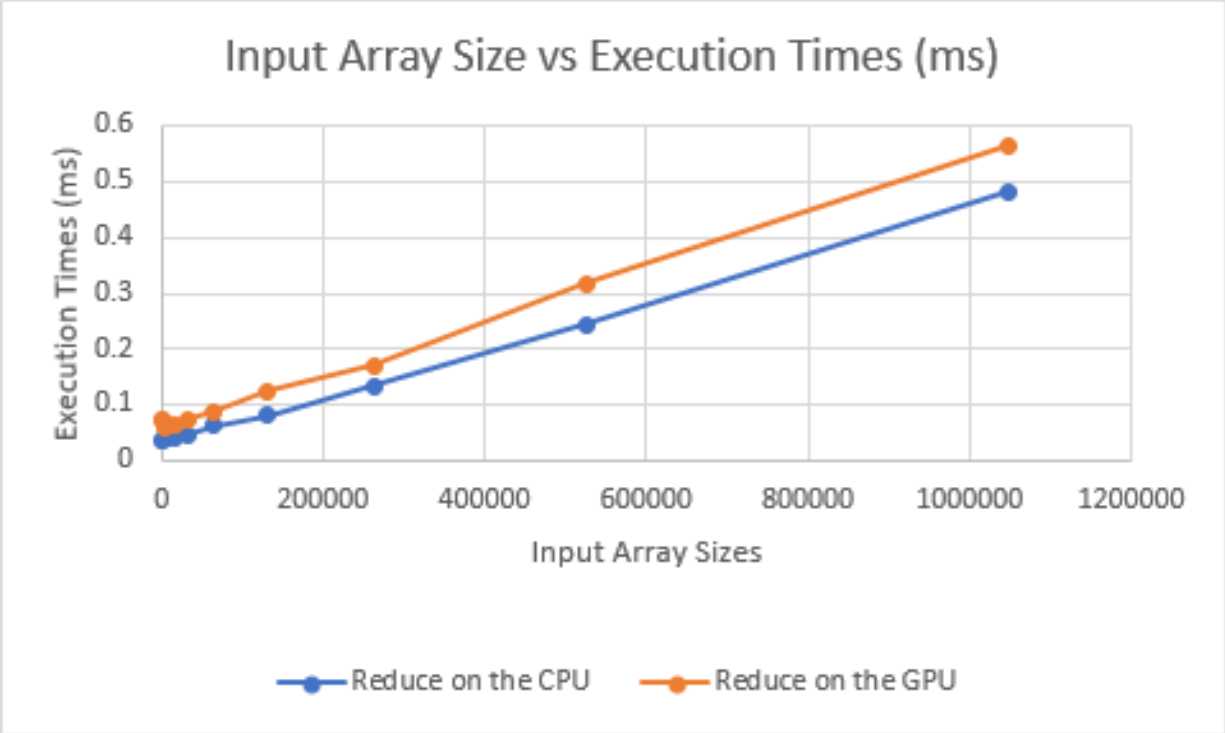
```
156
157     /* Allocates memory for d_in in the GPU */
158     cudaMalloc((void**) &d_in, INPUT_SIZE);
159     /* Copies the array from HOST (cpu) to DEVICE (gpu) */
160     cudaMemcpy(d_in, h_in, INPUT_SIZE, cudaMemcpyHostToDevice);
161     /* Allocates memory for d_out in the GPU */
162     cudaMalloc((void**) &d_out, OUTPUT_SIZE);
163
164
165     /* Number of threads per block, we divided it by 2 such that...  */
166     dim3 threadsPerBlock(BLOCK_SIZE/2);
167     /* Number of blocks*/
168     dim3 blocks(SECOND_ARRAY_SIZE);
169
170     /* Start the timer */
171     cudaEventRecord(start);
172
173     /* Launching kernal with 16 block and 16 threads per block */
174     shared_reduce_kernal<<<blocks, threadsPerBlock>>>(d_out, d_in);
```

```
175
176     /* cudaDeviceSynchronize(): halts execution in the CPU thread
177      * until the GPU has finished processing all previously requested
178      * cuda tasks.
179      *
180      * Wait for GPU to finish before accessing on host
181      */
182     cudaDeviceSynchronize();
183
184     /* Prints the input array
185     printf("Input Array: \n");
186     for (int n = 0; n < ARRAY_SIZE; n++) {
187       printf("%f ", h_in[n]);
188     }
189     printf("\n"); */
190
191     /* Copies the output array from device on to the Host's output array */
192     cudaMemcpy(h_out, d_out, OUTPUT_SIZE, cudaMemcpyDeviceToHost);
193
194
195
196     for (int n = 0; n < SECOND_ARRAY_SIZE; n++) {
197       final_h_in[n] = h_out[n];
198     //  printf("%f ", final_h_in[n]);
199     }
200
201     /* Allocates memory for final_d_in in the GPU */
202     cudaMalloc((void**) &final_d_in, OUTPUT_SIZE);
203     /* Copies the array from HOST (cpu) to DEVICE (gpu) */
204     cudaMemcpy(final_d_in, final_h_in, OUTPUT_SIZE, cudaMemcpyHostToDevice);
205     /* Allocates memory for d_out in the GPU */
206     cudaMalloc((void**) &final_d_out, OUTPUT_SIZE);
207
208
209     double allocatedBlockSize = 0;
210
211
212     if (((float) SECOND_ARRAY_SIZE / BLOCK_SIZE) <= 1) {
213       allocatedBlockSize = 1;
214     }else {
215       allocatedBlockSize = std::ceil(SECOND_ARRAY_SIZE/(double)BLOCK_SIZE);
216     }
```

```
218     final_reduce_kernal<<<allocatedBlockSize, threadsPerBlock>>>(final_d_out, final_d_in);
219
220     cudaDeviceSynchronize();
221     cudaMemcpy(final_h_out, final_d_out, OUTPUT_SIZE, cudaMemcpyDeviceToHost);
222     /* Final reduced value */
223     printf("Final reduction: %f\n", final_h_out[0]);
224
225     /* Outputs the results
226     for (int n = 0; n < SECOND_ARRAY_SIZE; n++) {
227       printf("%f ", final_h_out[n]);
228     } */
229
230     /* Stops the timer*/
231     cudaEventRecord(stop);
232     cudaEventSynchronize(stop);
233     float milliseconds = 0;
234     cudaEventElapsedTime(&milliseconds, start, stop);
235     printf("Elapsed time was: %f milliseconds \n", milliseconds);
236
237     /* Frees the variables */
238     cudaFree(d_in);
239     cudaFree(d_out);
240     cudaFree(final_d_in);
241     cudaFree(final_d_out);
242   }
243
```

Step 2 [5 marks]: record and plot set of timings: Execution Times (ms) vs Input Array Size

Input Array Sizes	Reduce on the CPU	Reduce on the GPU
1024	0.03904	0.07248
2048	0.0392	0.07264
4096	0.040704	0.062272
8192	0.042176	0.065312
16384	0.043776	0.065696
32768	0.04832	0.074976
65536	0.06416	0.089664
131072	0.082592	0.125632
262144	0.134048	0.171744
524288	0.244864	0.318368
1048576	0.48192	0.56544
2097152	Segmentation Fault	Segmentation Fault

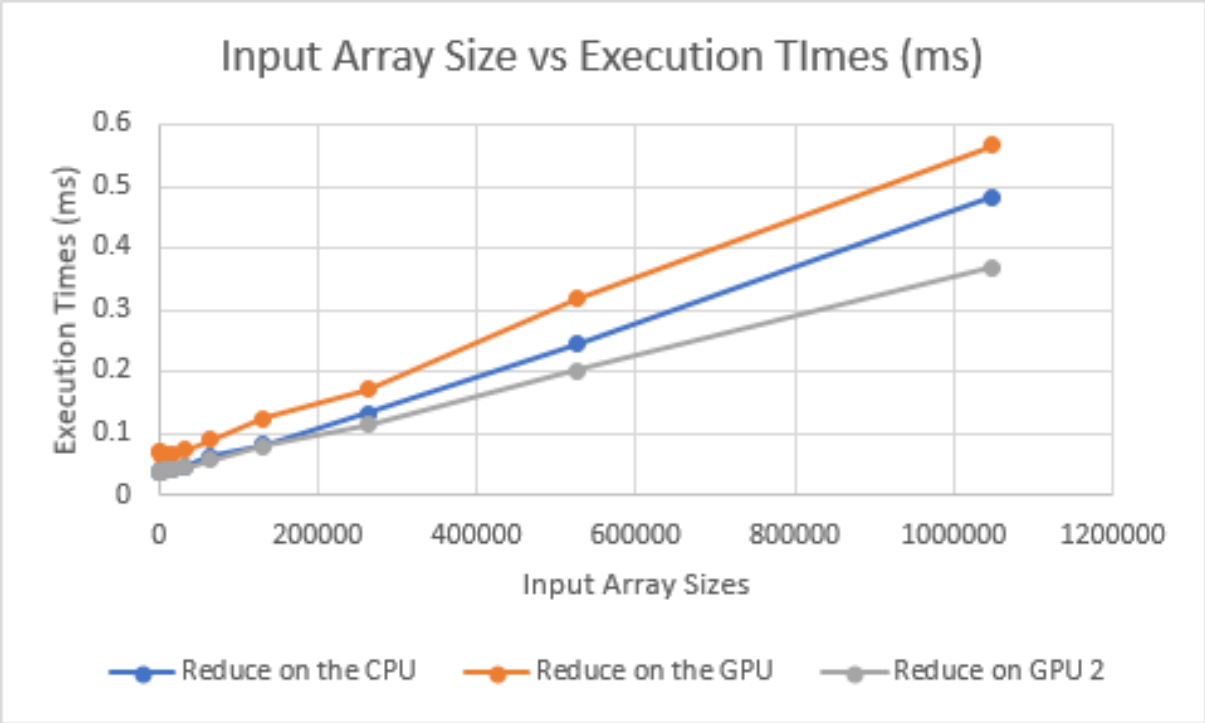


Note: I tested the execution time for Input Array Sizes from 1024 to 1048576 with intervals double the previous values. At array size 2097152 a segmentation Fault error occurred which signified that the GPU most likely ran out of memory. Therefore, I stopped at that point for recording results.

The data gathered from the experiment strongly highlights that the final reduction done on the CPU is faster than the final reduction done on the GPU. This rejects my hypothesis as I had initially thought that the GPU would be faster. This is because, in my implementation my program does a further reduction and then atomically adds the final values together. I thought that by repeating the reduction process we effectively managed to reduce the final number of elements in the array to add together. The results clearly disprove my theory, I believe the reason is because there is bottleneck along the PCI express bus which carries data between the GPU and CPU. The bottleneck being that there is an increase in overhead from sending data back and forth to the GPU, this bottleneck most likely explains the reason

why we are seeing an increase in execution time with the GPU. If this was the case, then perhaps just having the atomic add in the first kernel and not having to call the second kernel will optimize the GPU code further. See below for the results:

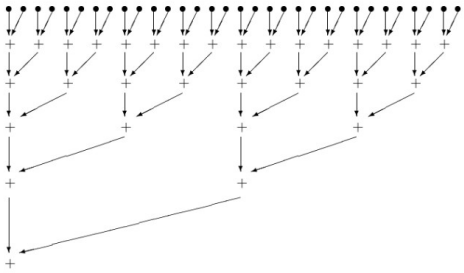
Input Array Sizes	Reduce on the CPU	Reduce on the GPU	Reduce on GPU 2
1024	0.03904	0.07248	0.038272
2048	0.0392	0.07264	0.038912
4096	0.040704	0.062272	0.040192
8192	0.042176	0.065312	0.04208
16384	0.043776	0.065696	0.042752
32768	0.04832	0.074976	0.045728
65536	0.06416	0.089664	0.57824
131072	0.082592	0.125632	0.079424
262144	0.134048	0.171744	0.11472
524288	0.244864	0.318368	0.202016
1048576	0.48192	0.56544	0.3688



Here I have updated the data with my third program “Reduce on GPU 2”. This program will only call one kernel. The kernel will then lastly execute the atomicAdd function to calculate the final reduced value. Atomic operations are used to prevent race conditions to perform a certain operation, to prevent these race conditions, an operation is executed one thread at a time, essentially making the operation serial and not parallel. Nvidia’s atomic operations is optimized and quite fast for variables stored in shared memory but slow in device global memory. Since we are dealing with shared memory atomic operations in shared memory may have contributed to a faster overall execution time compared to the final reduction done on the CPU.

Step 3 [5 marks]: Why we repeatedly partition the array into two halves
If we implement reduce with the pattern mentioned in lecture 10 we introduce thread divergence. Thread divergence is essentially where threads perform different operations causing in-synchronization between threads which could effectively lead to unprecedented errors in the results or increase the execution time. We get thread divergence since implementing the pattern requires the use of if statements/for loop which creates conditional branches causing divergence.

```
__shared__ float partialSum[];

int t = threadIdx.x;
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
   if(t % (2*stride) == 0)
      partialSum[t] += partialSum[t+stride];
}
```



The code above taken from [7] demonstrates the implementation described in the question. We can clearly identify thread divergence here at lines 7 to 8. This is because thread indexes that meet the condition (condition which allows each consecutive pairs to be added together) diverges to execute the code, however for every other thread in a pair would remain in an idle state.

With our current implementation we get less thread divergence, since instead of every other thread in a pair being idle we have every half a block size remaining idle. [7] Upon research I found that all threads in a warp take the same path meaning that the threads are synchronous and does not require and sync barriers. Taking advantage of warps means that there are no thread divergence. Half of the threads within a warp will do the addition operation whilst the other half will skip the statement.

Part 3: Scan

Step 1 [15 marks]: Implementation of Hillis and Steele Scan which uses multiple block size in order to handle large input arrays.

scan.cu

```
/*
============================================================================
Name : Scan.cu
Author : Nithesh Koneswaran
Version :
Copyright : Your copyright notice
Description : CUDA compute Scan (Hillis and Steele)
============================================================================
*/
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>

#define ARRAY_SIZE 64
#define BLOCK_SIZE 32
  /* Performs the first level of scan. Does not handle multiple block size */
__global__ void scan_kernel(int n, float *idata) {
   //Gets the position of the thread
   int global_index = threadIdx.x + blockIdx.x * blockDim.x;
   //Gets the index of the thread
   int local_index = threadIdx.x;
   /** Used to identify which buffer we are currently reading from */
   bool selector = true;
   /* Defined amount of shared memopry, has the same size as block size
      We need two buffers in order to avoid a race condition
      one is used to read and the other is used to write
    */
   __shared__ float mem[BLOCK_SIZE];
   __shared__ float mem2[BLOCK_SIZE];
   /** Copies all values from global memory to shared memory
    *  Remember each block has their own shared memory
    *  that is why we assign position index with shared memory's thread index
    *  example:
    *  thread with index 0 will have a another value compared to
    *  thread with index 0 ins another block   */
   mem[local_index] = idata[global_index];
   /** This makes sure each thread has loaded their values copied from global memory*/
   __syncthreads();
   /*
    * for each loop offset values are 1,2,4,8,16
    * This code will scan an array of elements, each held in a thread, in one block
    */
   for (int offset = 1; offset < n; offset *=2) {
     /* if the thread index is greater than or equal to the offset
        and of the thread index is less than the size of the array then scan
        the elements. */
     if (local_index >= offset && global_index < ARRAY_SIZE) {
        if (selector) {
            /* get the current thread and add it with the thread with the position index (current thread index - offset) */
          mem2[local_index] = mem[local_index] + mem[local_index - offset];
```

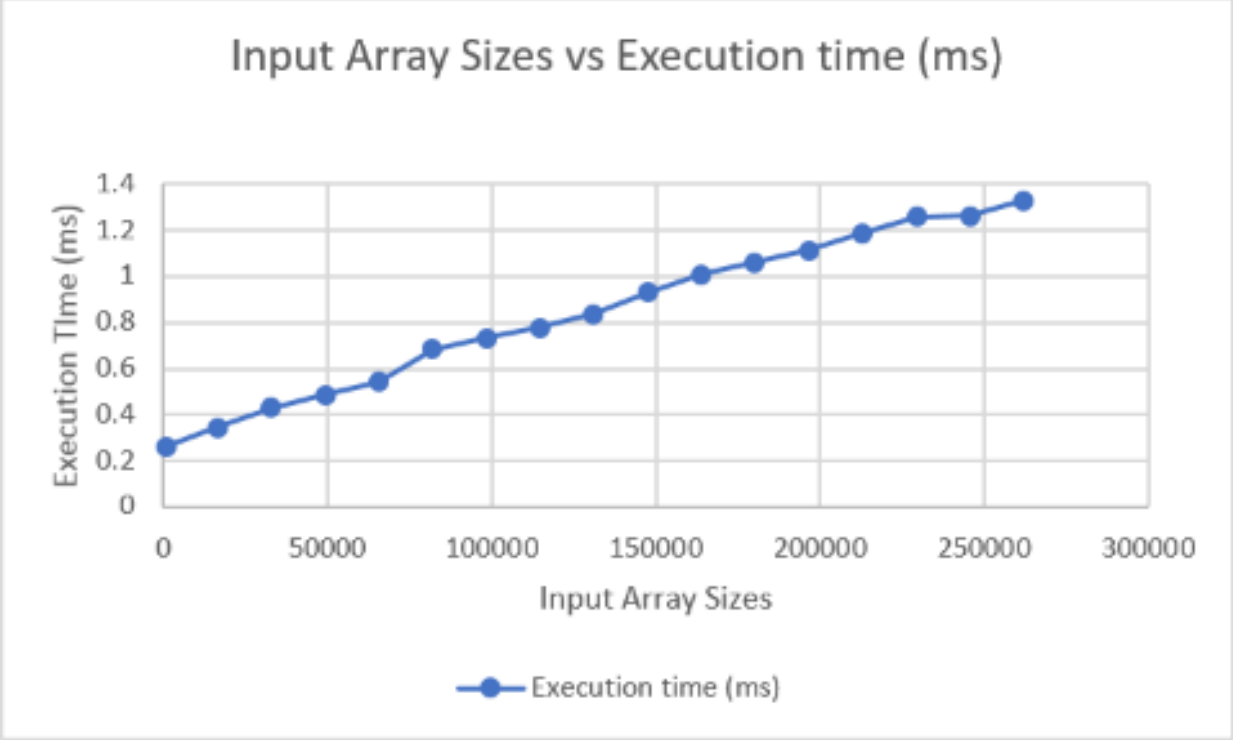
```
        } else {
          /* get the current thread and add it with the thread with the position index (current thread index - offset) */
           mem[local_index] = mem2[local_index] + mem2[local_index - offset];;
        }
      }
      /*  if the thread index is less than the offset
          we will then copy the values between the two buffers
       */
      if (local_index < offset) {
        if (selector) {
          mem2[local_index] = mem[local_index];
        } else {
          mem[local_index] = mem2[local_index];
        }
      }
      /* Updates the condition */
      selector = !selector;
      /** This makes sure each thread has loaded their values copied from global memory*/
      __syncthreads();
    }

    /* We need to make sure we output the value from the correct buffer */
    if (selector) {
       /** Writes results from shared memory to global */
       idata[global_index] = mem[local_index];
    } else {
       idata[global_index] =  mem2[local_index];
    }
}

/* Performs the final level of scan effectively completing the scan */
__global__ void final_scan_kernal(float *idata, float *idata2) {
   //Gets the position of the thread
   int global_index = threadIdx.x + blockIdx.x * blockDim.x;
   //Gets the index of the thread
   int local_index = threadIdx.x;
   /* Defined amount of shared memopry, specified in the host code */
   __shared__ float mem[BLOCK_SIZE];
   __shared__ float out;
   /** Copies all values from global memory to shared memory*/
   mem[local_index] = idata[global_index];
   /** This makes sure each thread has loaded their values copied from global memory*/
   __syncthreads();
   /* Map operation, this will add the elements in idata2 onto the elements found in each block except the first block. */
   if (blockIdx.x!=0) {
     out = idata2[blockIdx.x-1];
     __syncthreads();
     mem[local_index] = mem[local_index] + out;
     /* We sync the threads to prevent any race conditions */
     __syncthreads();
   }
```

```
   /* We finally write the value from shared memory back to global */
   idata[global_index] = mem[local_index];
}

int main(void) {
  /** Declaring pointers,
    * idata will contain the results of the first level scan after the first kernal processing
    * idata at the end of the execution will contain the results of the final scan
    * idata2 will contain the highest most element from each stride equal to block width */
   float *idata, *idata2;
   /* Begin recording */
   cudaEvent_t start, stop;
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   cudaError_t err;
```

```
    // Allocate Unified Memory - accessible from CPU or GPU
    cudaMallocManaged(&idata, ARRAY_SIZE*sizeof(float));
    cudaMallocManaged(&idata2, (ARRAY_SIZE/BLOCK_SIZE)*sizeof(float));
    // Initialise the input data on the host
    // Making it easy to test the results
    for (int i = 0; i < ARRAY_SIZE; i++) {
     idata[i] = 1.0f;
    }

    /* Calculates the the number of blocks */
    int grid_size = (ARRAY_SIZE/BLOCK_SIZE);
    /* Begin recording */
    cudaEventRecord(start);
    /* Launching kernal with the following parameters, performs the first level of scan */
    scan_kernel<<<grid_size, BLOCK_SIZE>>>(BLOCK_SIZE, idata);
    /*  Performs the second level of scan on the CPU */

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

    /* Reads the item at the final block and writes it to another array*/
    for (int x=BLOCK_SIZE-1, y=0; x < ARRAY_SIZE && y<grid_size; x+=BLOCK_SIZE, y++) {
       idata2[y] = idata[x];
    }
   /*  Performs the second level of scan on the CPU */
    for (int i=0; i<grid_size-1; i++) {
      idata2[i+1] = idata2[i]+idata2[i+1];
    }

    /* Launching kernal with the following parameters, performs the final level of scan */
    final_scan_kernal<<<grid_size, BLOCK_SIZE>>>(idata, idata2);
    // Wait for GPU to finish before accessing on host
    err = cudaDeviceSynchronize();

    /* Stops the timer*/
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Elapsed time was: %f milliseconds \n", milliseconds);

    /* Gets any errors */
    printf("Run kernel: %s\n", cudaGetErrorString(err));
    // Now output the resulting array:
    printf("result: %f\n", idata[ARRAY_SIZE-1]);

    printf("\n");
    /* Frees the variables */
    cudaFree(idata);
    cudaFree(idata2);
    return 0;
}
```

Step 2 [5 marks]: Execution times recorded as input array size increases.

Input Array Sizes	Execution time (ms)
1024	0.263040
16384	0.347072
32768	0.432128
49152	0.486400
65536	0.542624
81920	0.685440
98304	0.734912
114688	0.779008
131072	0.837632
147456	0.930336
163840	1.007776
180224	1.059840
196608	1.113184
212992	1.188672
229376	1.259872
245760	1.264320
262144	1.328128



Step 3 [5 marks]: Two propositions in improve performance of the current implementation.

One proposition in improving performance involves using CUDA's warp-level primitives. [4] Warps are a group of threads that perform a single instruction, in this case [3] we could perform a scan across a single warp of threads then scan across the blocks of threads and finally combine to get the final output. To implement this, we would remove the synchronous barrier since threads within a warp are all synchronous. We can 'unroll' the for loop so that only threads with index between 0 and 31 perform the scan function

Another proposition to improve the performance is to perform scan on different segments of the array. [5] By performing the scan on different parts of the array and then combining the solutions we increase the parallelism of the algorithm. This would be effective for a scan operation involving a large array input, instead of using one large scan we could simply perform many independent scan operations on different segments of the array. To mark where a segment begins we require another array (flag) with the same length as the input array and we add the value 1 to specify where a segment begins.

Two propositions to improve performance of **Lab 5 Scan implementation**

Using the material from the labs I learnt that the standard scan lab solution had a race condition which was solved using two buffers, one for reading and the other for writing. The race condition arises when a thread adds two elements together and updates its position however, just before it reads an element, another thread may have already updated it to a different value which will affect the scan result. The proposed solution requires two shared memory variables, one will be used to be read by a thread and the other will store the results of the operation after the read. After each iteration it's important that the two pairs switch functions so that we now read from the 'new' buffer and write to the 'old' buffer. This ensures the thread will never do an operation on an updated value. It's important to note that we must sync threads so that all the threads are working at the same pace, this also helps prevent any race conditions. (Note I have already implemented this in my solution for scan).

It's important that we use shared memory when implementing scan. As explained in part A, (quick summary See part A for the differences between shared and global memory) shared memory sits relatively close to threads in blocks compared to global memory. This physical advantage means that you get faster access times. However, as a price shared memory has much less storage and requires a change in thought process when coding since shared memory is only accessible by threads in a block. Essentially to implement shared memory, we assign each block's SRAM cell with a part of the input array such that each block will have access to different parts of an array. We also use another shared mem variable which will be used to load the second level scan output array (used for the map operation at the end) from global memory. After that we can do our map operation on the two arrays and finally moving the results from shared memory to global memory.

Part 4: Histogram

Step 1 [10 marks]: Implementation of Histogram using kernel from Lecture 11

```
/*
============================================================================
Name : histogramA.cu
Author : Nithesh Koneswaran
Version :
Copyright : Your copyright notice
Description : CUDA compute histogram
============================================================================
*/
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define ARRAY_SIZE 64
#define BLOCK_SIZE 32
/* Performs the first level of scan. Does not handle multiple block size */
__global__ void simple_histogram(int *d_bins, int *d_in, const int BIN_COUNT) {
  /* Gets the global index position of the thread */
  int global_index = threadIdx.x + blockDim.x * blockIdx.x;
  int item = d_in[global_index];
  int bin = item % BIN_COUNT;
  atomicAdd(&(d_bins[bin]), 1);
}
int main(void) {
  const int BIN_COUNT = 8;
  /** The number of bytes required to store the array containing values */
  const size_t INPUT_SIZE = ARRAY_SIZE * sizeof(int);
  /** The number of bytes required to store the number of bins*/
  const size_t BIN_SIZE = BIN_COUNT * sizeof(int);
  /* Initialising pointers to be used in the kernal */
  int *d_bins , *d_in;
  /* Initialising Array of dynamic size 'ARRAY_SIZE', this is the input
     that will be used to increment the bins */
  int h_in[ARRAY_SIZE];
  /* Initialising Array with size equal to the number of bins. Each item in the
     array corresponds to a bin. Each bin will contain a statistical integer
     calculated by simply incrementing the bin whenever an item in the input Array
     falls in the specified bin (this is done by a mod function in the kernal)*/
  int h_bins[BIN_COUNT];
  /* Begin recording */
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaError_t err;
```

```
45    /* Setting values for our input array */
46    for (int i = 0; i < ARRAY_SIZE; i++) {
47        h_in[i] = i;
48    }
49    /* Setting values for the bins, default 0 */
50    for (int i = 0; i < BIN_COUNT; i++) {
51        h_bins[i] = 0;
52    }
53    /* Allocates memory for d_in in the GPU */
54    cudaMalloc((void**) &d_in, INPUT_SIZE);
55    /* Copies the array from HOST (cpu) to DEVICE (gpu) */
56    cudaMemcpy(d_in, h_in, INPUT_SIZE, cudaMemcpyHostToDevice);
57    /* Allocates memory for d_out in the GPU */
58    cudaMalloc((void**) &d_bins, BIN_SIZE);
59    /* Calculates the the number of blocks */
60    const int GRID_SIZE = ARRAY_SIZE / BLOCK_SIZE;
61     /* Begin recording */
62    cudaEventRecord(start);
63    /* Launching kernal with the following parameters, performs the first level of scan */
64    simple_histogram<<<GRID_SIZE, BLOCK_SIZE>>>(d_bins, d_in, BIN_COUNT);
65    // Wait for GPU to finish before accessing on host
66    err = cudaDeviceSynchronize();
67    /* Copies the output array from device on to the Host's output array */
68    cudaMemcpy(h_bins, d_bins, BIN_SIZE, cudaMemcpyDeviceToHost);
69    /* Stops the timer*/
70    cudaEventRecord(stop);
71    cudaEventSynchronize(stop);
72    float milliseconds = 0;
73    cudaEventElapsedTime(&milliseconds, start, stop);
74    printf("Elapsed time was: %f milliseconds \n", milliseconds);
75    /* Gets any errors */
76    printf("Run kernel: %s\n", cudaGetErrorString(err));
77    printf("\n");
78    /* Outputs the results */
79    for (int i = 0; i < BIN_COUNT; i++) {
80      // Now output the resulting array:
81      printf("Bin number: %d: Count: %d\n", i, h_bins[i]);
82    }
83    /* Frees the variables */
84    cudaFree(d_in);
85    cudaFree(d_bins);
86    return 0;
87  }
88
```

Step 2 [10 marks]: Modified Implementation

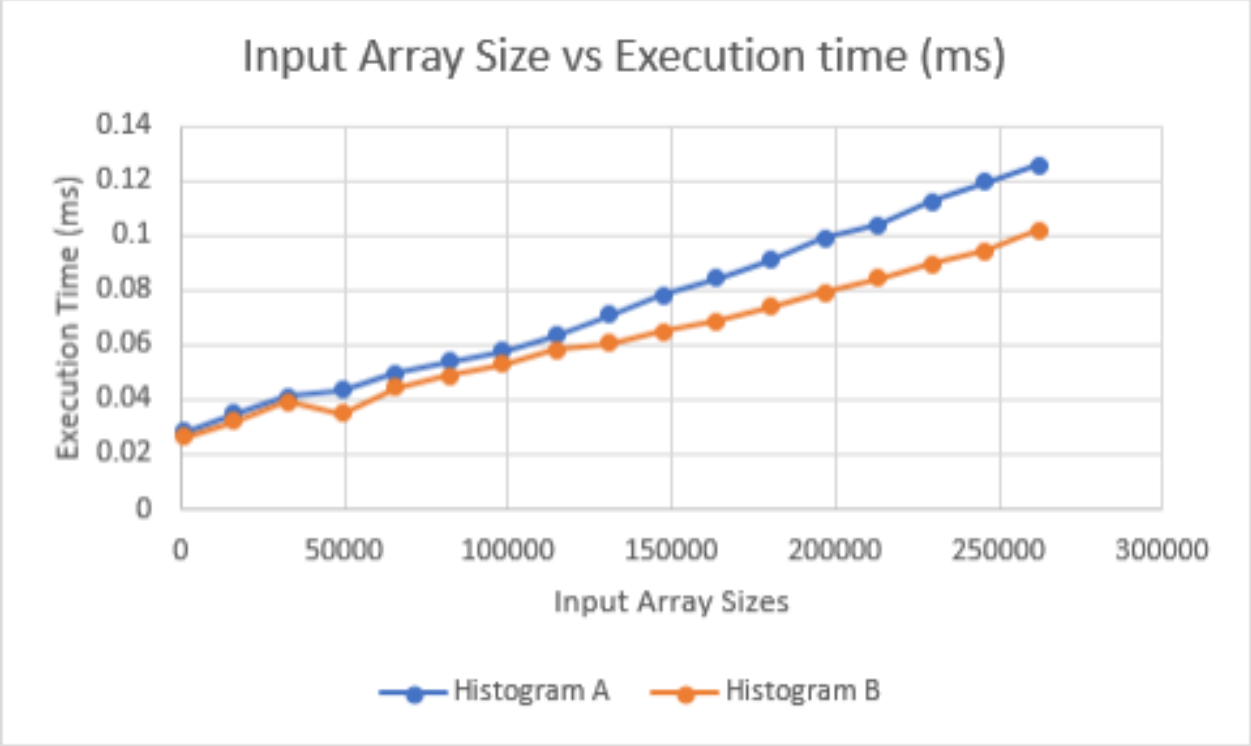
```
/*
============================================================================
Name : histogramA.cu
Author : Nithesh Koneswaran
Version :
Copyright : Your copyright notice
Description : CUDA compute histogram
============================================================================
*/
#include <stdio.h>
#include <numeric>
#include <stdlib.h>
#include <cuda.h>
#define ARRAY_SIZE 64
#define BLOCK_SIZE 32
#define BIN_COUNT 8
/* Performs the first level of scan. Does not handle multiple block size */
__global__ void simple_histogram(int *d_bins, int *d_in) {
  /* Gets the global index position of the thread */
  int global_index = threadIdx.x + blockDim.x * blockIdx.x;
  int local_index = threadIdx.x;
  int item = d_in[global_index];

  __shared__ int local_histogram[BIN_COUNT];
  /* Initialising bin stored in shared memory */
  if (local_index < BIN_COUNT) {
      local_histogram[local_index] = 0;
  }
  /* This makes sure each thread has loaded their values copied from global memory*/
  __syncthreads();
  if (global_index < ARRAY_SIZE) {
    int bin = item % BIN_COUNT;
    atomicAdd(&(local_histogram[bin]), 1);
  }
  /* This makes sure each thread has reached the same stage*/
  __syncthreads();

  /* Instead of atomic add, we could reduce the local histograms */
  atomicAdd((int*)&d_bins[local_index], local_histogram[local_index]);
  /* This makes sure each thread has reached the same stage */
  __syncthreads();
}
int main(void) {
  /** The number of bytes required to store the array containing values */
  const size_t INPUT_SIZE = ARRAY_SIZE * sizeof(int);
  /** The number of bytes required to store the number of bins*/
  const size_t BIN_SIZE = BIN_COUNT * sizeof(int);
  /* Initialising pointers to be used in the kernal */
  int *d_bins , *d_in;
  /* Initialising Array of dynamic size 'ARRAY_SIZE', this is the input
     that will be used to increment the bins */
  int h_in[ARRAY_SIZE];
```

```
    /* Initialising Array with size equal to the number of bins. Each item in the
       array corresponds to a bin. Each bin will contain a statistical integer
       calculated by simply incrementing the bin whenever an item in the input Array
       falls in the specified bin (this is done by a mod function in the kernal)*/
    int h_bins[BIN_COUNT];
    /* Begin recording */
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaError_t err;
    /* Setting values for our input array */
    for (int i = 0; i < ARRAY_SIZE; i++) {
        h_in[i] = i;
    }
    /* Setting values for the bins, default 0 */
    for (int i = 0; i < BIN_COUNT; i++) {
        h_bins[i] = 0;
    }
    /* Allocates memory for d_in in the GPU */
    cudaMalloc((void**) &d_in, INPUT_SIZE);
    /* Copies the array from HOST (cpu) to DEVICE (gpu) */
    cudaMemcpy(d_in, h_in, INPUT_SIZE, cudaMemcpyHostToDevice);
    /* Allocates memory for d_out in the GPU */
    cudaMalloc((void**) &d_bins, BIN_SIZE);
    /* Calculates the the number of blocks */
    const int GRID_SIZE = ARRAY_SIZE / BLOCK_SIZE;
     /* Begin recording */
    cudaEventRecord(start);
    /* Launching kernal with the following parameters, performs the first level of scan */
    simple_histogram<<<GRID_SIZE, BLOCK_SIZE>>>(d_bins, d_in);
    // Wait for GPU to finish before accessing on host
    err = cudaDeviceSynchronize();
    /* Copies the output array from device on to the Host's output array */
    cudaMemcpy(h_bins, d_bins, BIN_SIZE, cudaMemcpyDeviceToHost);
    /* Stops the timer*/
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Elapsed time was: %f milliseconds \n", milliseconds);
    /* Gets any errors */
    printf("Run kernel: %s\n", cudaGetErrorString(err));
    printf("\n");
```

```
    /* Outputs the results */
    for (int i = 0; i < BIN_COUNT; i++) {
      // Now output the resulting array:
      printf("Bin number: %d: Count: %d\n", i, h_bins[i]);
    }
    /* Frees the variables */
    cudaFree(d_in);
    cudaFree(d_bins);
    return 0;
}

```

Step 3 [5 marks]: Differences between the two implementations

Input Array Sizes	Histogram A	Histogram B
1024	0.028544	0.026944
16384	0.035328	0.032448
32768	0.041336	0.039776
49152	0.044064	0.035104
65536	0.050240	0.044992
81920	0.054272	0.049312
98304	0.057920	0.053432
114688	0.063968	0.058496
131072	0.071584	0.061056
147456	0.078464	0.065256
163840	0.084544	0.069184
180224	0.091368	0.074364
196608	0.099520	0.079272
212992	0.104000	0.084656
229376	0.112576	0.089736
245760	0.119688	0.094552
262144	0.125728	0.102382



From the set of data, we can see that as the input array size increases the execution times increases. From the two implementations of histogram we can see that Histogram B out performed Histogram A for all input array sizes. This is because Histogram B uses shared memory. [2] Each block will hold their own set of as (local) histogram (a shared variable array with n number of bins) which will, in parallelism, increment the corresponding bin for a value which is retrieved using the current thread's index to access an element from global memory. After all the threads in each block have updated their local histogram the next step is to add up the local histograms. It is because every block has their own set of bins (implemented as shared memory) instead of having just one set of bins (implemented as

global memory) to increment, this makes histogram A not scalable since the atomics used limits the amount of parallelism. This limitation is overcome through incrementing local histograms and then finally adding the histograms together, since this introduces some parallelism since multiple histograms are incremented simultaneously. A suggestion to improve histogram B is to reduce the local histograms together instead of using atomic add, the use of atomics again will limit the amount of parallelism. Using a reduction method to add the local histogram would be ideal.

References:

[1] “why shared memory is faster than global memory?” , stackoverflow [Online] , Available: https://stackoverflow.comq/uestions/28804760/why-shared-memory-is-faster-than-global-memory [Accessed: 28/04/19]:

[2] “Implementing Histogram Using Local Memory” , Udacity [Online] , Available: https://www.youtube.com/watch?v=B_IavYV8C94&list=PLGvfHSgImk4aweyWlhBXNF6XISY3um82_&index=156 [Accessed: 23/04/19]:

[3] “Efficient Parallel Scan Algorithms for GPUs”, research.nvidia.com [Online] , Available: https://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf [Accessed: 25/04/19]:

[4] “Warp-level Primitives”, devblogs.nvidia.com [Online] , Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/ [Accessed: 25/04/19]:

[5] “Segmented Scan”, Udacity [Online] , Available: https://www.youtube.com/watch?v=Yp2eDz6dvNA [Accessed: 25/04/19]:

[6] Lecture 10, “Week4: Evaluating Parallel Algorithms”, SurreyLearn [Online], Available: https://surreylearn.surrey.ac.uk/d2l/le/content/172335/Home [Accessed: 23/04/19]:

[7] ”Warps and Reduction Algorithms”, Homepages [Online], Available: http://homepages.math.uic.edu/~jan/mcs572/mcs572notes/lec34.html [Accessed: 23/04/19]:

[8] “Optimizing Parallel Reduction in CUDA”, NVIDIA [Online], Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf [Accessed: 18/04/19]: