Kyle Salitrik
CMPSC 450
Homework 4 Report

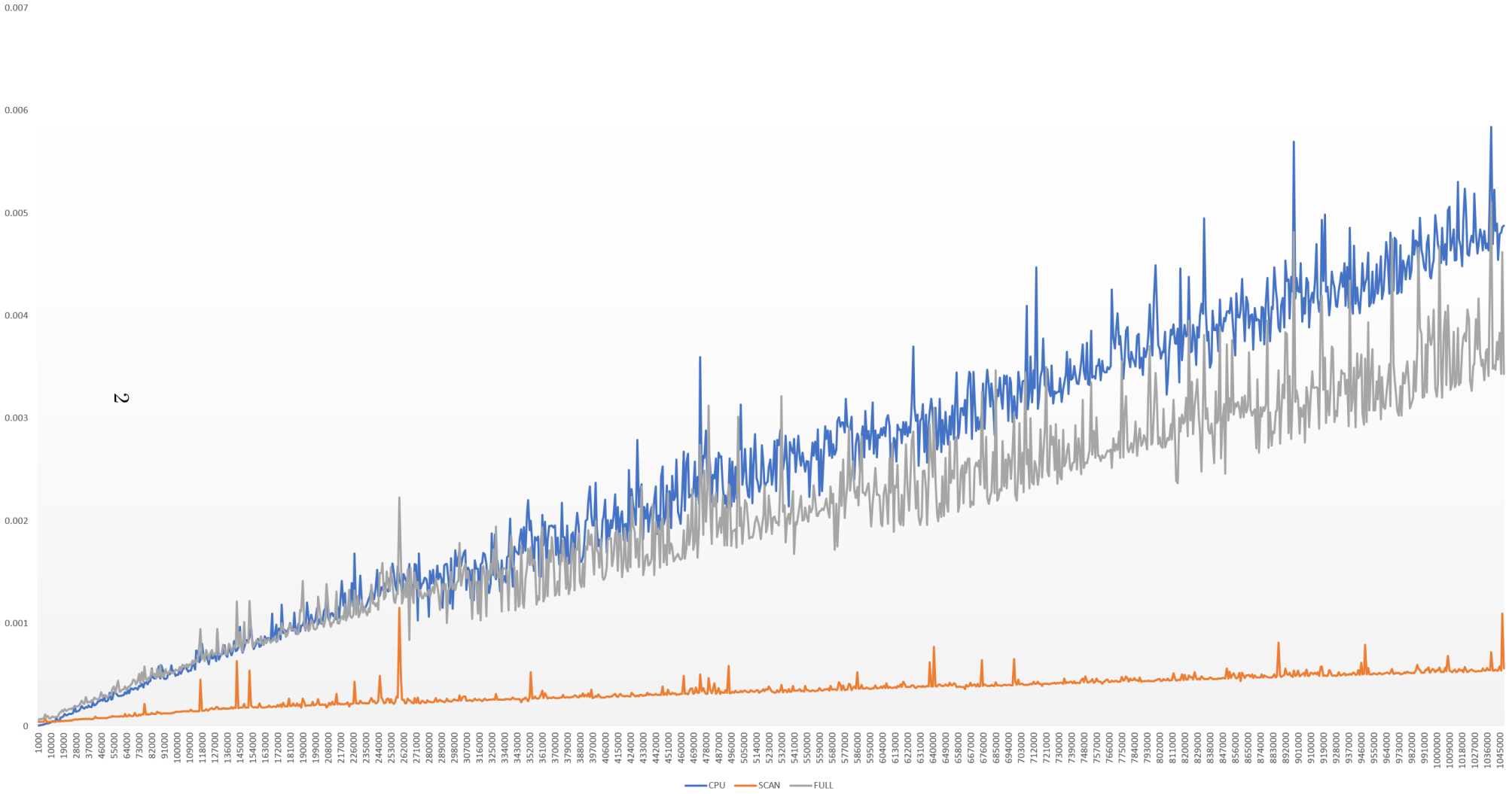
Serial vs Parallel

After adapting the CUDA code (ref: https://github.com/amosgwa/Exclusive-Scan-CUDA, https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html) to handle arrays larger than 1024, the program was set to run for M = 1000 to M = 10,000,000 in steps of 1000, where M is the size of the vector. Times were collected at the following sections in the code:

- Before copying array to GPU
- Before running the GPU calculations
- After running the GPU calculations
- After copying the array from the GPU to the host
- Before running the serialized code
- After running the serialized code

This data is plotted in the figure below. Unsurprisingly, at low values of M, the serial computation is significantly faster due to the amount of time that it takes to copy the memory from the host to the GPU. The data ends up slightly noisy and inconsistent using such a fine step, however at around 261000 elements, a clear divergence starts where the GPU calculation time begins to be less even when copying all of the elements is considered.

In contrast, if one looks at only the prefix sum calculation time, it increases at a significantly slower rate than the CPU runtimes. Looking at the actual percentage increase, the GPU calculation only increases by $\approx 54.46x$ while the CPU time increases by $\approx 1626x$, with the total GPU execution time increasing by only $\approx 13x$. This shows that both the CPU and GPU implementations are bound by computation time after a certain point, where memory access becomes irrelevant. The following figure includes a graph of the runtimes.


0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
CPU
SCAN
FULL

Code Appendix

```
#include <sys/time.h>
#include <cublas_v2.h>
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>

__global__ void prescan(double *input, double *output, double *scratch, int N)
{
    extern __shared__ double temp[];

    int thread = blockDim.x * blockIdx.x + threadIdx.x;
    int blockThread = threadIdx.x;
    int arrayIndex = 2 * blockThread;

    int offset = 1;

    temp[arrayIndex] = input[2*thread];
    temp[arrayIndex + 1] = input[2*thread + 1];

    for (int d = blockDim.x; d > 0; d = d / 2)
    {
        __syncthreads();

        if (blockThread < d)
        {
            int ai = offset * (arrayIndex + 1) - 1;
            int bi = offset * (arrayIndex + 2) - 1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

    if (blockThread == 0)
    {
        if (scratch) scratch[blockIdx.x] = temp[N - 1];
        temp[N - 1] = 0;
    }

    for (int d = 1; d < blockDim.x * 2; d *= 2)
    {
        offset = offset / 2;
        __syncthreads();

        if (blockThread < d)
        {
            int ai = offset * (arrayIndex + 1) - 1;
            int bi = offset * (arrayIndex + 2) - 1;
            double t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
    __syncthreads();

    output[2*thread] = temp[arrayIndex];
    output[2*thread + 1] = temp[arrayIndex + 1];
}
```

```
59
60 __global__ void prescanSum (double *to_add, double *result, int N)
61 {
62     double addition = to_add[blockIdx.x];
63     int thread = blockDim.x * blockIdx.x + threadIdx.x;
64     result[thread] += addition;
65 }
66
67 void scanCPU(double *f_out, double *f_in, int i_n)
68 {
69     f_out[0] = 0;
70     for (int i = 0; i < i_n -1; i++)
71     {
72         f_out[i+1] = f_out[i] + f_in[i];
73     }
74
75 }
76
77 double myDiffTime(struct timeval &start, struct timeval &end)
78 {
79     double d_start, d_end;
80     d_start = (double) (start.tv_sec + start.tv_usec / 1000000.0);
81     d_end = (double) (end.tv_sec + end.tv_usec / 1000000.0);
82     return (d_end - d_start);
83 }
84
85 int main(int argc, char **argv)
86 {
87     // Running parameters
88     int numThreads = 1024;
89     int N = atoi(argv[1]);
90
91     // Init CUDA size variables
92     int numBlocks = N / numThreads;
93     if (N % numThreads != 0) numBlocks++;
94     int vecLen = numBlocks * numThreads;
95     const dim3 blockSize(numThreads / 2, 1, 1);
96     const dim3 gridSize(numBlocks, 1, 1);
97
98 //     printf("Number of Blocks: %d\nNumber of Threads: %d\nVecLen: %d\nN: %d\n",
    numBlocks, numThreads, vecLen, N);
99 //     printf("Blocksize: %d\nGridsize: %d\n",blockSize.x, gridSize.x);
100
101     // Host Memory Allocation
102     double *host_CPU, *host_input, *host_GPU, *host_scratch, *host_addition;
103
104     host_CPU = (double *) calloc(vecLen, sizeof(double));
105     host_input = (double *) calloc(vecLen, sizeof(double));
106     host_GPU = (double *) calloc(vecLen, sizeof(double));
107     host_scratch = (double *) calloc(vecLen, sizeof(double));
108     host_addition = (double *) calloc(vecLen, sizeof(double));
109
110
111     double d_gpuFullTime, d_gpuScanTime, d_cpuTime;
112     timeval fullStart, fullEnd, scanStart, scanEnd;
113
114     // Device Memory Allocation
115     double *dev_GPU, *dev_input, *dev_scratch, *dev_addition;
116     cudaError_t err;
117     err = cudaMalloc((void **) &dev_input, vecLen * sizeof(double));
```

```
118     if (err != cudaSuccess) fprintf(stderr, "ERROR in Malloc of dev_input: %s\n",
        cudaGetErrorString(err));
119     err = cudaMalloc((void **) &dev_GPU, vecLen * sizeof(double));
120     if (err != cudaSuccess) fprintf(stderr, "ERROR in Malloc of dev_GPU: %s\n",
        cudaGetErrorString(err));
121     err = cudaMalloc((void **) &dev_scratch, vecLen * sizeof(double));
122     if (err != cudaSuccess) fprintf(stderr, "ERROR in Malloc of dev_scratch: %s\n
        ", cudaGetErrorString(err));
123     err = cudaMalloc((void **) &dev_addition, vecLen * sizeof(double));
124     if (err != cudaSuccess) fprintf(stderr, "ERROR in Malloc of dev_addition: %s\
        n", cudaGetErrorString(err));
125
126     err = cudaMemset(dev_input, 0, vecLen * sizeof(double));
127     if (err != cudaSuccess) fprintf(stderr, "ERROR in memset of dev_input: %s\n",
        cudaGetErrorString(err));
128     err = cudaMemset(dev_GPU, 0, vecLen * sizeof(double));
129     if (err != cudaSuccess) fprintf(stderr, "ERROR in memset of dev_GPU: %s\n",
        cudaGetErrorString(err));
130     err = cudaMemset(dev_scratch, 0, vecLen * sizeof(double));
131     if (err != cudaSuccess) fprintf(stderr, "ERROR in memset of dev_scratch: %s\n
        ", cudaGetErrorString(err));
132     err = cudaMemset(dev_addition, 0, vecLen * sizeof(double));
133     if (err != cudaSuccess) fprintf(stderr, "ERROR in memset of dev_addition: %s\
        n", cudaGetErrorString(err));
134
135
136     // Generate array values
137     for (int i = 0; i < N; i++)
138     {
139 //        host_In[i] = (double) (rand() % 1000000) / 1000.0;
140         host_input[i] = (double) i * 1.0;
141     }
142
143     gettimeofday(&fullStart, NULL);
144     // Copy from host to device
145     cudaMemcpy(dev_input, host_input, vecLen * sizeof(double),
        cudaMemcpyHostToDevice);
146
147     // Perform scan on CUDA
148     // START SCAN AND GET SCAN START TIME
149     gettimeofday(&scanStart, NULL);
150
151     prescan<<<gridSize, blockSize, numThreads * sizeof(double)>>>(dev_input,
        dev_GPU, dev_scratch, numThreads);
152     cudaDeviceSynchronize();
153     err = cudaGetLastError();
154     if (err != cudaSuccess) fprintf(stderr, "ERROR in prescan: %s\n",
        cudaGetErrorString(err));
155
156     // ACCUMULATE ADDITIONS
157     prescan<<<dim3(1,1,1), blockSize, numThreads * sizeof(double) >>>(dev_scratch
        , dev_addition, NULL, numThreads);
158     cudaDeviceSynchronize();
159     err = cudaGetLastError();
160     if (err != cudaSuccess) fprintf(stderr, "ERROR in creating addition vector: %
        s\n", cudaGetErrorString(err));
161
162     // ADD TO ELEMENTS FOR TRUE SUM
163     prescanSum<<<gridSize, dim3(numThreads,1,1)>>>(dev_addition,dev_GPU, N);
164     cudaDeviceSynchronize();
```

```
165     err = cudaGetLastError();
166     if (err != cudaSuccess) fprintf(stderr, "ERROR in addition: %s\n",
        cudaGetErrorString(err));
167
168     // GET SCAN END TIME
169     gettimeofday(&scanEnd, NULL);
170
171     // Copy memory back and get full end time
172     err = cudaMemcpy(host_GPU, dev_GPU, vecLen*sizeof(double),
        cudaMemcpyDeviceToHost);
173     if (err != cudaSuccess) fprintf(stderr, "ERROR in memcpy back to host: %s\n",
         cudaGetErrorString(err));
174
175     gettimeofday(&fullEnd, NULL);
176
177
178     err = cudaMemcpy(host_scratch, dev_scratch, vecLen*sizeof(double),
        cudaMemcpyDeviceToHost);
179     if (err != cudaSuccess) fprintf(stderr, "ERROR in memcpy back to host: %s\n",
         cudaGetErrorString(err));
180     err = cudaMemcpy(host_addition, dev_addition, vecLen*sizeof(double),
        cudaMemcpyDeviceToHost);
181     if (err != cudaSuccess) fprintf(stderr, "ERROR in memcpy back to host: %s\n",
         cudaGetErrorString(err));
182
183     d_gpuFullTime = myDiffTime(fullStart, fullEnd);
184     d_gpuScanTime = myDiffTime(scanStart, scanEnd);
185
186     // Perform serial scan
187     gettimeofday(&fullStart, NULL);
188     scanCPU(host_CPU, host_input, N);
189     gettimeofday(&fullEnd, NULL);
190     d_cpuTime = myDiffTime(fullStart, fullEnd);
191
192
193     for (int i = 0; i < N; i++)
194     {
195         if (host_CPU[i] - host_GPU[i] > 0.1)
196         {
197           printf("Results do not match! c[%i]=%f, g[%i]=%f\n", i, host_CPU[i], i,
         host_GPU[i]);
198             break;
199         }
200     }
201
202     printf("%i\t%f\t%f\t%f\n", N, d_cpuTime,d_gpuScanTime,d_gpuFullTime);
203 }
```