CPE810 – GPU And Multicore Programming

Lab Report #4: Project

Lab Scan

Instructor: Zhou Feng

TA: Ali Aghdaei

James Fong

I pledge my honor that I have abided by the Stevens Honor System

Table of Contents

Objective

Implement a kernel to perform an inclusive parallel scan on a 1D list (with randomly generated integers). The scan operator will be the addition (plus) operator. You should implement the work efficient kernel in our lecture. Your kernel should be able to handle input lists of arbitrary length. To simplify the lab, the student can assume that the input list will be at most of length 2048*65535 elements. This means that the computation can be performed using only one kernel launch. The boundary condition can be handled by filling “identity value (0 for sum)” into the shared memory of the last block when the length is not a multiple of the thread block size.

Required Functions and Implementation Details

1. Allocate Device Memory

```
float blockscan(int* output, int* input, int length, bool bcao) {
int* d_out, * d_in;
const int arraySize = length * sizeof(int);

cudaMalloc((void**)&d_out, arraySize);
cudaMalloc((void**)&d_in, arraySize);
cudaMemcpy(d_out, output, arraySize, cudaMemcpyHostToDevice);

cudaMemcpy(d_in, input, arraySize, cudaMemcpyHostToDevice);
```

2. Copy Host Memory to Device

```
cudaMemcpy(d_out, output, arraySize, cudaMemcpyHostToDevice);

cudaMemcpy(d_in, input, arraySize, cudaMemcpyHostToDevice);


cudaMemcpy(d_out, output, arraySize, cudaMemcpyHostToDevice);

cudaMemcpy(d_in, input, arraySize, cudaMemcpyHostToDevice);
```

3. Initialize Thread Block and Kernel Grid Dimensions

```
bool canBeBlockscanned = N <= 1024;

time_t t;
srand((unsigned)time(&t));
int* in = new int[N];
for (int i = 0; i < N; i++) {
in[i] = rand() % 10;
}

if (argc >= 2) {
	int elements = argc;
	int numElements = sizeof(argc) / sizeof(argv[0]);

	for (int i = 0; i < numElements; i++) {
		test(atoi(argv[i]));
	}
	return 0;
}
```

```
	else if (argc == 1) {
		int elements[] = { 134215680, 2000000, 1000000, 10000, 5000, 4096, 2048,
2000, 1024, 1000, 500, 100, 64, 8, 5 };

		int numElements = sizeof(elements) / sizeof(elements[0]);

		for (int i = 0; i < numElements; i++) {
			test(elements[i]);
		}
		return 0;

	}
```

4. Invoke CUDA Kernel

```
if (bcao) {
prescan_arbitrary << <1, (length + 1) / 2, 2 * powerOfTwo * sizeof(int) >> > (d_out,
d_in, length, powerOfTwo);
}
else {
prescan_arbitrary_unoptimized << <1, (length + 1) / 2, 2 * powerOfTwo * sizeof(int) >> >
(d_out, d_in, length, powerOfTwo);

}
```

5. Copy Results from Device to Host

```
printf("%i Elements \n", N);

// sequential scan on CPU
int* outCPUSeq = new int[N]();
long time_host = CPU_Seq_Scan(outCPUSeq, in, N);
printResult("host    ", outCPUSeq[N - 1], (time_host/100));
printf("Throughput = %lf MElements/s, Time = %lf s\n", (1.0e2 * N / time_host),
(time_host * 1.0e-8));

//(1.0e6 * N / time_host)
// full scan
int* outGPU = new int[N]();
float time_gpu = scan(outGPU, in, N, false);
printResult("gpu     ", outGPU[N - 1], (time_gpu));
printf("Throughput = %lf MElements/s, Time = %lf s\n", (1.0e-3 * N / time_gpu),
(time_gpu));


// full scan with BCAO
int* outGPUBC = new int[N]();
float time_gpuoutGPUBC = scan(outGPUBC, in, N, true);
printResult("gpu bcao", outGPUBC[N - 1], time_gpuoutGPUBC);
```

```
printf("Throughput = %lf MElements/s, Time = %lf s\n", (1.0e-3 * N / time_gpuoutGPUBC),
(time_gpuoutGPUBC));

if (canBeBlockscanned) {
// basic level 1 block scan
int* out_1block = new int[N]();
float time_1block = blockscan(out_1block, in, N, false);
printResult("level 1 ", out_1block[N - 1], time_1block);
printf("Throughput = %lf MElements/s, Time = %lf s\n", (1.0e-3 * N / time_1block),
(time_1block));

// level 1 block scan with BCAO
int* out_1blockoutGPUBC = new int[N]();
float time_1blockoutGPUBC = blockscan(out_1blockoutGPUBC, in, N, true);
printResult("l1 bcao ", out_1blockoutGPUBC[N - 1], time_1blockoutGPUBC);
printf("Throughput = %lf MElements/s, Time = %lf s\n", (1.0e-3 * N /
time_1blockoutGPUBC), (time_1blockoutGPUBC));
```

6. Deallocate Device Memory

```
delete[] in;
delete[] outCPUSeq;
delete[] outGPU;
delete[] out_1block;
delete[] out_1blockoutGPUBC;
delete[] outGPUBC;
```

7. Implement The Work-Efficient Inclusive Scan Routine Using Shared Memory

```
__global__ void prescan_arbitrary(int* output, int* input, int n, int powerOfTwo)
{
extern __shared__ int temp[];// allocated on invocation
int threadID = threadIdx.x;

int ai = threadID;
int bi = threadID + (n / 2);
int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
int bankOffsetB = CONFLICT_FREE_OFFSET(bi);


if (threadID < n) {
temp[ai + bankOffsetA] = input[ai];
temp[bi + bankOffsetB] = input[bi];
}
else {
temp[ai + bankOffsetA] = 0;
temp[bi + bankOffsetB] = 0;
}


int offset = 1;
for (int d = powerOfTwo >> 1; d > 0; d >>= 1) // build sum in place up the tree
```

```
{
__syncthreads();
if (threadID < d)
{
int ai = offset * (2 * threadID + 1) - 1;
int bi = offset * (2 * threadID + 2) - 1;
ai += CONFLICT_FREE_OFFSET(ai);
bi += CONFLICT_FREE_OFFSET(bi);

temp[bi] += temp[ai];
}
offset *= 2;
}

if (threadID == 0) {
temp[powerOfTwo - 1 + CONFLICT_FREE_OFFSET(powerOfTwo - 1)] = 0; // clear the last
element
}

for (int d = 1; d < powerOfTwo; d *= 2) // traverse down tree & build scan
{
offset >>= 1;
__syncthreads();
if (threadID < d)
{
int ai = offset * (2 * threadID + 1) - 1;
int bi = offset * (2 * threadID + 2) - 1;
ai += CONFLICT_FREE_OFFSET(ai);
bi += CONFLICT_FREE_OFFSET(bi);

int t = temp[ai];
temp[ai] = temp[bi];
temp[bi] += t;
}
}
__syncthreads();

if (threadID < n) {
output[ai] = temp[ai + bankOffsetA];
output[bi] = temp[bi + bankOffsetB];
}
}
__global__ void prescan_large(int* output, int* input, int n, int* sums) {
	extern __shared__ int temp[];

	int blockID = blockIdx.x;
	int threadID = threadIdx.x;
	int blockOffset = blockID * n;

	int ai = threadID;
	int bi = threadID + (n / 2);
	int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
	int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
	temp[ai + bankOffsetA] = input[blockOffset + ai];
	temp[bi + bankOffsetB] = input[blockOffset + bi];
```

```
	int offset = 1;
	for (int d = n >> 1; d > 0; d >>= 1) // build sum in place up the tree
	{
		__syncthreads();
		if (threadID < d)
		{
			int ai = offset * (2 * threadID + 1) - 1;
			int bi = offset * (2 * threadID + 2) - 1;
			ai += CONFLICT_FREE_OFFSET(ai);
			bi += CONFLICT_FREE_OFFSET(bi);

			temp[bi] += temp[ai];
		}
		offset *= 2;
	}
	__syncthreads();


	if (threadID == 0) {
		sums[blockID] = temp[n - 1 + CONFLICT_FREE_OFFSET(n - 1)];
		temp[n - 1 + CONFLICT_FREE_OFFSET(n - 1)] = 0;
	}

	for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
	{
		offset >>= 1;
		__syncthreads();
		if (threadID < d)
		{
			int ai = offset * (2 * threadID + 1) - 1;
			int bi = offset * (2 * threadID + 2) - 1;
			ai += CONFLICT_FREE_OFFSET(ai);
			bi += CONFLICT_FREE_OFFSET(bi);

			int t = temp[ai];
			temp[ai] = temp[bi];
			temp[bi] += t;
		}
	}
	__syncthreads();

	output[blockOffset + ai] = temp[ai + bankOffsetA];
	output[blockOffset + bi] = temp[bi + bankOffsetB];

}
```

8. Handle thread divergence when dealing with arbitrary input sizes

```
void _checkCudaError(const char* message, cudaError_t err, const char* caller) {
if (err != cudaSuccess) {
fprintf(stderr, "Error in: %s\n", caller);
fprintf(stderr, message);
fprintf(stderr, ": %s\n", cudaGetErrorString(err));
```

```
exit(0);
}

}
```

Questions

1. Name 3 applications of parallel scan.

Steam Compaction – a parallel programming primitive that produces a reduced output stream consisting only of valid elements from an input stream [1]

String Comparison [2]

String Comparison, where multiple pattern matching is an important function in many varieties of security, network applications involving information retrieval, web filtering, intrusion detection, virus scanners, spam filters, and other tools to combat unwanted digital intrusions.

Thus, given an input list of arbitrary length, it can be preprocessed and used to search for multiple patterns in the input string. Thus, by creating a preprocessing phase of something, for example X, a data structure X is created. X is proportional to the length of the pattern set and tries to determine if any patters are occurring in the input string.

Counting Sort

Counting Sort is an algorithm that counts the number of occurrences of each number within an array, by sorting a collection of objects according to keys that are small integers. [6]

Prefix sum is applied to determine the position of all values within the output array. The algorithm operates by counting the number of objects with distinct key values and applies prefix sum on those counts to determine the position of each key value in the output sequence.

2. How many floating operations are performed in your reduction kernel? Explain.

The number of floating-point operations being performed by our reduction kernel is dependent on the total size of the array being inputted. The reduction kernel is being called in the Arbitrary Unoptimized function, where it will take up to a total of two in the first for loop with two different float operations, and three in the second introduced for loop operation. Afterwards, because both for loops are ran, it follows each call, dependent on the array size, by a factor of log 2. Thus, the number of floating operations being done will be performed by (6+4) log (n).

```
for (int d = powerOf2 >> 1; d > 0; d >>= 1) // build sum in place up the tree
{
	__syncthreads();
	if (threadID < d)
	{
		int ai = offset * (2 * threadID + 1) - 1;
		int bi = offset * (2 * threadID + 2) - 1;
		temp[bi] += temp[ai];
	}
	offset *= 2;
}

if (threadID == 0)
{
	temp[powerOf2 - 1] = 0; // clear the last element
}

for (int d = 1; d < powerOf2; d *= 2) // traverse down tree & build scan
{
	offset >>= 1;
	__syncthreads();
	if (threadID < d)
	{
		int ai = offset * (2 * threadID + 1) - 1;
		int bi = offset * (2 * threadID + 2) - 1;
		int t = temp[ai];
		temp[ai] = temp[bi];
		temp[bi] += t;
	}

}
```

3. How many global memory reads are being performed by your kernel? Explain.

The number of global memory reads being performed by the kernel is equal to the number of elements being done by the array. There are two different operations that are used for reading, loading input into shared memory.

```
temp[2 * threadID] = d_Input[2 * threadID]; // load input into shared memory
temp[2 * threadID + 1] = d_Input[2 * threadID + 1];
```

4. How many global memory writes are being performed by your kernel? Explain.

The number of global memory writes being performed by the kernel is equal to the number of elements being done by the array. There are two different operations that are used for writing and loading input into the array, as each follow up operation will perform an operation to the following array.

```
d_Output[2 * threadID] = temp[2 * threadID]; // write results to device memory
d_Output[2 * threadID + 1] = temp[2 * threadID + 1];
```

5. What is the minimum, maximum, and average number of real operations that a thread will perform? Real operations are those that directly contribute to the final output value.

The minimum number of real operations that a thread will perform are dependent on the given array size to the function. The elements inside are arbitrary, as the scan will cover all data for sum addition. Thus, the minimum number of real operations that will be done is a minimum of 1 real operation, as it only input for a maximum of 2 elements.

The maximum number of real operations that a thread will perform are dependent on the given array size, in which the maximum allowable amount of arrays for the program is

2048*65535. If the arrays are filled randomly with the maximum allowable value, then the maximum number of real operations the thread will perform is log n.

The average number of real operations that a thread will perform are dependent on the given array size divided by the total number of operations in the threads. The threads are given a maximum of 2 operations, so the number of real operations is determined by log 2n.

6. How many times does a single thread block synchronize to reduce its portion of the array to a single value?

A single thread block synchronizes to reduce its portion of the array to a single value by log n times; this will be done once it finishes the reduction phase and begins the down sweep tree traversal and begins to build the scan for the final product.

```
for (int d = 1; d < powerOfTwo; d *= 2) // traverse down tree & build scan
	{
		offset >>= 1;
		__syncthreads();

		if (threadID < d)
```

7. Describe what optimizations were performed to your kernel to achieve a performance speedup.

Optimizations performed to our kernel to achieve a performance speedup was the inclusion of avoiding banking conflicts as another algorithm for comparison. [3] In the experimentation tab, this is elaborated further and given in the results tab. The first scan algorithm is incorporated with variable padding to each shared memory array index computed,

where the index of the value is indexed, then divided by the number of shared memory banks. This optimization is directly compared with the CPU Sequential Scan and the GPU Scan.

8. Describe what further optimizations can be implemented to your kernel and what would be the expected performance behavior?

Further optimizations that can be implemented to our kernel are that can be implemented is by covering global memory access latency, by doing more computations per thread [4]. Most congestion of the scan code are done by the global memory latency and instruction overhead, which is caused by looping and address computation instructions. A technique, suggested by David Lichterman, processes eight elements per thread instead of two, by loading two float4 elements per thread rather than two float elements. Each thread now performs a sequential scan of each float4. The first three elements of each scan are stored in registers, and the total sum of the scans are inserted into the shared memory array. Partial sums from all threads in shared memory allows for identical tree-based scan, and scanned elements from shared memory to each of the partial sums stored in registers; the float4 values are then written into global memory.

The expected performance behavior would result in being twice as fast as the previous code, as a result of Brent's Theorem. This approach is often used for improving efficiency of different parallel algorithms. [4] [5]

9. Suppose the input is greater than 2048*65535, what modifications are needed to your kernel?

If the input is greater than 2048*65535, you would need to split up your kernel calls. You would need to first be able to split it up into different chunks with a maximum size of 2048*65535. Then, calculate the maximum allowable kernel call computations by one kernel call, and save that data (calling from memory to host). Then, you would need to call the next maximum allowable kernel call computation and repeat upon saving the calculations from memory to host, repeated recursively. Once all needed operations are done from each individual kernel call, you would repeat the same operation from all operated results and combine until you reach your desired prefix sum operation count.

10. Suppose you want to scan using a binary operator that's not commutative, can you use a parallel scan for that?

The parallel scan can be used for any operation that is associative, thus it does not need to be commutative. It makes it useful for many applications, for calculating well-separated pair decompositions of points to string processing. [7]

Experimentation

The major experimentation done was comparing three different algorithms;

The first algorithm is the sequential scan, which utilizes only the CPU to run all operations on a single thread of CPU. A loop is done over all elements in the input array, and random variables are included in each element of the array. The sum is computed for the previous element of the output array, and the sum is written to the current element of the array.

The second algorithm is a parallel scan on the GPU, which is a work efficient parallel scan. The GPU scan is more efficient than the naïve algorithm, as naïve implementation is work inefficient by log2n. Instead, based on the implementation presented by Blelloch, an algorithmic pattern using balanced threes on the input data, and sweeping it to and from the root to compute the prefix sum. The algorithm consists of two phases; the reduce phase, and the down-sweep phase, where the reduce phase involves traversing the tree from leaves to root and computes the partial sus at internal nodes of the tree. After this, the root node holds the sum of all nodes in the array. Following the down sweep phase, a traversal is done back down the tree from the root of the tree, and at erach step, each node at the current level passes its own value to its left child, and the sum of its value and former value of its left child to its right child.

The third algorithm scan is done with parallel scan on the GPU while implemented to avoid banking conflicts. Banking conflicts are avoided when accessing __shared__ memory arrays. And by adding a variable amount of padding to each shared memory, it allows for avoidance of we can avoid most bank conflicts.

The variations of the scan are done with different arrays, determined either by a predetermined list of elements or an inserted command line argument using the following command:

./scan -i <dim>

Where <dim> is the input vector size.

Comparisons can be found on graph 1 and 2, where each of the performances are compared in the conclusion.

Conclusion

The scan operation is an extremely versatile operation. It is applicable in a broad range of applications, and there are various significant speed ups that are available for scan using CUDA and is overall significantly faster speedup compared to a sequential implementation on a strong CPU. The increasing power of commodity parallel processors on GPUs allows for data-parallel algorithms to continue developing into the future.

Comparing the data, the sequential scan works best when performing operations for a small number of arrays. According to the resulting data in Graph 1, the throughput comparison between all three algorithms, the CPU Scan algorithm from about 2048 elements outperforms the GPU Scan and GPU Scan that avoids banking conflicts, and overall performs with a higher amount of throughput. However, this seems to cap at about 50 mega elements per second and does not exceed this. This trend continues until 5000 elements are passed into the array, where increasing the number of elements will result in both GPU algorithms to perform with a higher throughput. The time function, shown in Graph 3, shows that time also scaled linearly, and takes more time after the 5000 element threshold.

Later comparisons with a larger number of elements, as seen from Graph 2 and Graph 4, results in showing the throughput and time with a significantly larger throughput and time performance by both GPU Scan and GPU Scan avoiding Bank Conflicts. The CPU Scan remains consistent with a throughput average of about 50 mega elements per second, and thus drastically affects performance, the time metric increases linearly, and does not show drastic improvements. However, the GPU Scan algorithm performs with a stronger throughput, which also scales as

there are a larger number of elements inputted. This is the same trend with the GPU Scan avoiding Banking Conflicts.

Comparing GPU Scan with the GPU Scan avoiding banking conflicts, throughout most of the testing, the GPU Scan avoiding banking conflicts usually performs better over the GPU Scan, due to being able to utilize more banks with less conflicts, thus increasing performance on most instances.

Tables

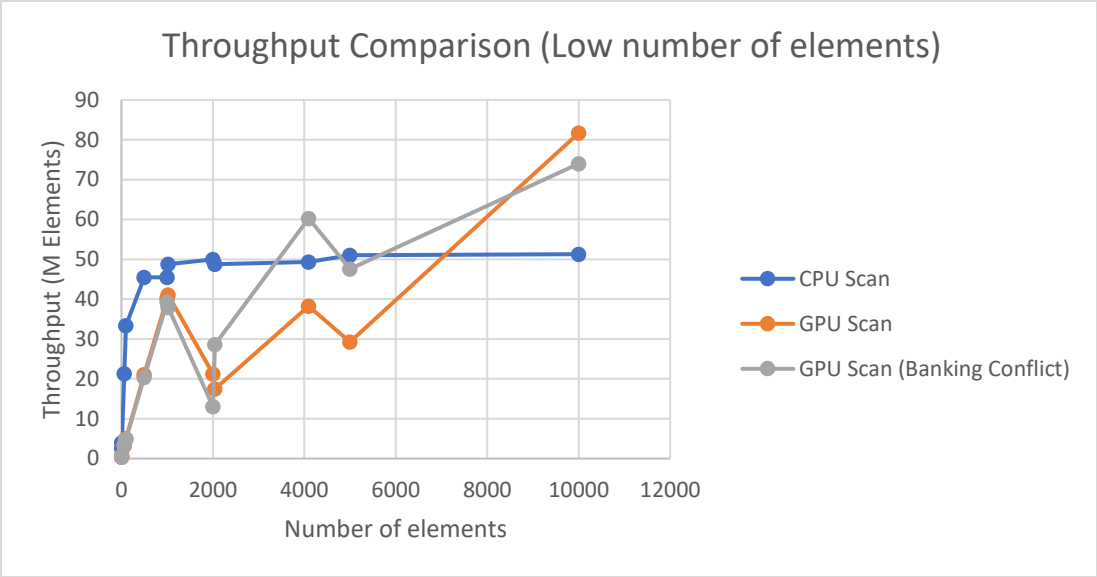
Number of Elements	Elements in Array (Inclusive)	CPU Scan Throughput (MElements/s)	GPU Scan Throughput (MElements/s)	GPU BCAO Throughput (Melements/s)
134215680	134215679	44.96718	1550.25192	1907.525317
2000000	1999999	45.02476	1306.438149	1316.260531
1000000	999999	44.01408	1039.345443	1122.485646
10000	9999	51.2820512	81.65664933	73.98200699
5000	4999	51.02040816	29.22745871	47.56468651
4096	4095	49.34939759	38.25463125	60.29203671
2048	2047	48.76190476	17.4529589	28.596961
2000	1999	50	21.22241034	13.02083271
1024	1023	48.76190476	41.0783058	37.91469087
1000	999	45.4545454	40.27062002	39.3081761
500	499	45.4545454	21.05795137	20.34505136
100	99	33.333333	4.97611457	4.83746151
64	63	21.333333	3.15457415	3.42465739
8	7	4	0.64935066	0.64766839
5	4	2.5	0.35112358	0.38580247

Table 1. Number of elements and Throughput from each scan.

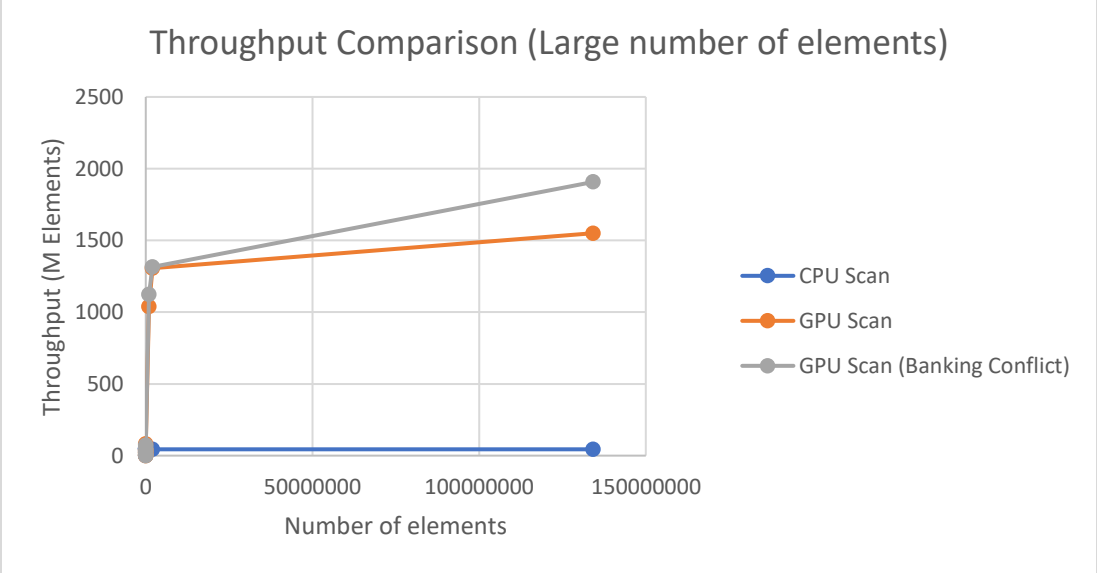
Number of Elements	Elements in Array (Inclusive)	CPU Scan Time (ms)	GPU Scan Time (ms)	GPU Scan Time Banking Conflict (ms)
134215680	134215679	4866	86.57636	70.361153
2000000	1999999	45	1.26	1.14
1000000	999999	21	0.962144	0.89088
10000	9999	0.201	0.122464	0.135168
5000	4999	0.1	0.171072	0.10512
4096	4095	0.084	0.107072	0.067936
2048	2047	0.042	0.117344	0.071616
2000	1999	0.042	0.09424	0.1536
1024	1023	0.022	0.024928	0.027008
1000	999	0.022	0.024832	0.02544
500	499	0.011	0.023744	0.024576
100	99	0.003	0.020096	0.020672
64	63	0.003	0.02088	0.018688
8	7	0.001	0.01232	0.012352
5	4	0.002	0.01424	0.01296

Table 2. Number of elements compared to time of scan completion.

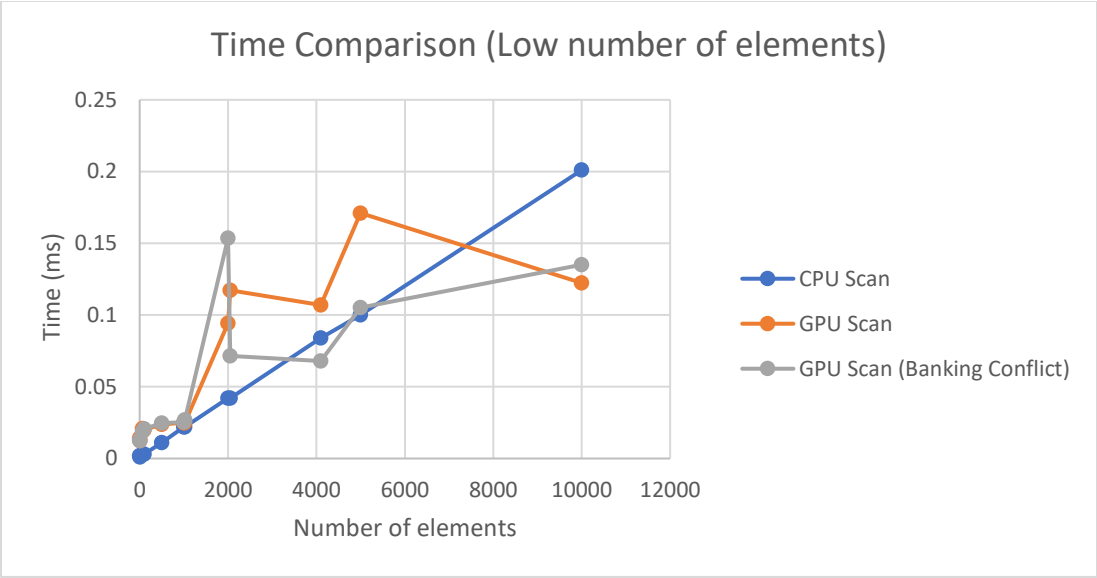
Graphs



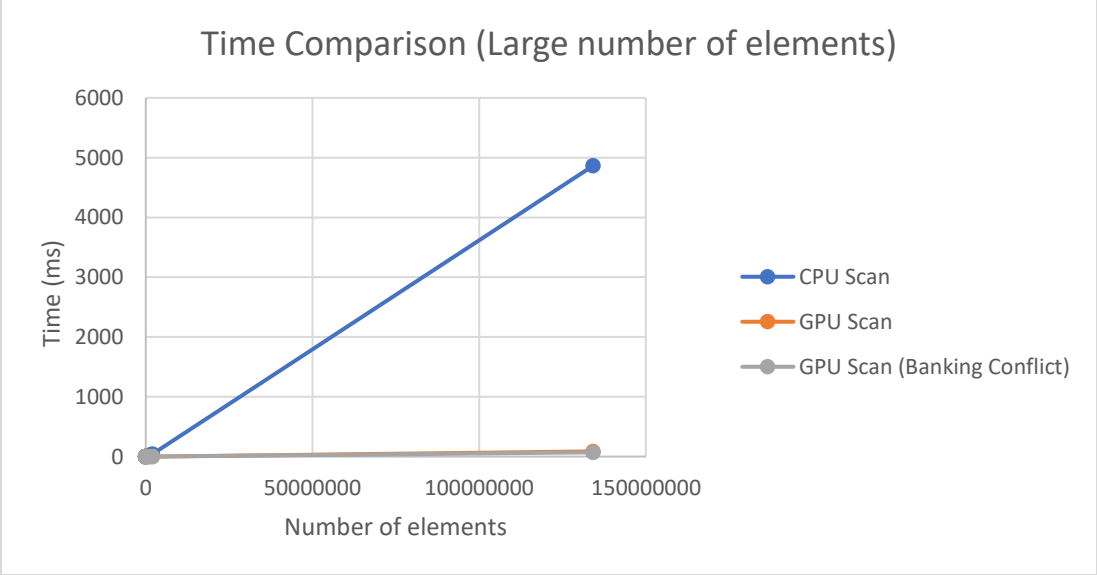
Graph 1: Throughput Comparison (Low number of elements)



Graph 2: Throughput Comparison (Large number of elements)



Graph 3. Time Comparison (Low number of elements)



Graph 4. Time Comparison (Large number of elements)

References

[1] https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[2] https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

[3] https://happy-sugar.life/book/a162eb5c5b3e4e832be8985dc25e6254/ch39.html

[4] https://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

[5] https://people.cs.vt.edu/yongcao/teaching/cs5234/spring2013/slides/Lecture10.pdf

[6] https://www.researchgate.net/publication/281234447_Multiple_string_matching_on_a_GPU_using_CUDA

[7] https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s18-lec37-slides-v1.pdf?version=1&modificationdate=1523898025646&api=v2