Daniel Pelis
CPE 810
11/20/2020

Lab 4: List Scan

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.

Instructions

Make a new folder for your project. Copy and modify the code of "scan" in CUDA sample code (6_Advanced folder) to include the following key functions:

1. allocate device memory
2. copy host memory to device
3. initialize thread block and kernel grid dimensions
4. invoke CUDA kernel
5. copy results from device to host
6. deallocate device memory
7. implement the work-efficient inclusive scan routine using shared memory
8. handle thread divergence when dealing with arbitrary input sizes

Your final executable can be run using the following command:

./scan -i <dim>

Where:

<dim> is the input vector size.

Implementation

To handle the specifications listed in the instructions above, I had to implement a kernel that could perform the parallel scan but also add the sum of each previous in a single pass. To achieve this, I followed a Stream-based Scan Algorithm, which consists of three parts.

The first part is the parallel list scan. For my kernel I chose to implement a Brent-Kung Scan Algorithm in a separate "device" function, which my kernel calls. Compared to the Kogge-Stone method, the Brent-Kung method is conceptually more complicated, but more work efficient. Since I used the Kogge-Stone algorithm, I declared my section size as a constant and calculated the thread block size to be half of the section size. The grid size is then set as the input vector size divided by the section size. Since the block size if half of the section size, each thread loads two values into the blocks shared memory, at the beginning of the Brent-Kung kernel. The first step of this algorithm calculates the sum for all the odd indexes in the section. Next it calculates the sum of all the indexes of the form $4 * n - 1$. This process is repeated by using the following equation and increasing the value of x for every iteration, $2^x * n - 1$. If the index exceeds the section size, then the thread does nothing for this iteration. If it falls within the section size, then the value at shared memory index is summed with the value at the index a stride lower in the shared memory. This part of the algorithm finishes once the 2^x value exceeds the size of the block dimension. At this point the reduction tree has completed and the reverse tree will begin. The reverse (distribution) tree works by again indexing with $2^x * n - 1$, but now the index is decreased at every iteration. In this loop, if the index plus the stride is greater than the section size then the thread does nothing. If it falls within the section size, then the value at shared memory index is summed with the value at the index a stride higher in the shared memory. One both loops have completed, each thread places the two variables its responsible for in shared memory. Now the device will return to the main kernel and move on to part two of the stream-based algorithm.

The second part of the kernel performs Adjacent Synchronization so that for the rest of the execution the thread blocks will execute in a measurable order. To do this each block is assigned a new id by having thread 0 of every block atomically adding to a counter in global memory and recording the result. Once the Brent-Kung scan is finished the first thread of every block will continually check its flag to see if the previous block has added its sum to the global array. Once block sees that its flag is non-zero, it will grad the previous sum, add it to its own sum and copy that to the shared array in global memory for the next block. It the increments the next block's flag so that it can begin, and the current block moves to the final step.

The last part of the kernel simply takes the sum of the previous block, collected in part two, and adds it to all the values of the current section. Since the block size is half the section size, this is done for both values the thread is responsible for.

Now that the kernel has finished, performance metrics are calculated, and the output is copied to the host. A list scan CPU function is also implemented to validate the results.

Questions

For all the expressions below, it is assumed that "N" stands for the input vector size.

1. Name 3 applications of parallel scan.

Examples of real-world applications of parallel scan algorithms are Stream Compaction, Summed-Area Tables, and Radix Sort. **Stream compaction** is the primary method for changing a vector filled with objects of various types into a vector with a elements of a single shared type. This is useful when trying to pull only the useful elements from a vector into a smaller vector to reach a higher computation efficiency. The operation in this example is a filtering operation. **Summed-Area Tables** (SATs) are 2D tables that hold the sums of the pixels between a specific location and the lower left corner of an input image. These tables are useful because they can be used to perform filters of differing width on an input image with a constant per pixel time. A use case of this application would be approximate depth of field rendering by applying a variable sized blur to an input image which is based on the depth of each pixel. The operation in this example is an addition operation. Lastly, **Radix Sort**, is an efficient sorting algorithm that uses parallel scan as a building block. This algorithm first divides the input array into chucks and sorts them in parallel. Then a recursive merge sort is run on the sorted chucks to continually merge them into larger sorted chunks, until the array is complete again.

Sources:

a. Nvidia GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA
    i. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

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

To begin calculating the number of FLOPs performed by my kernel I will focus on the operations performed by the Brent-Kung scan. This scan consists of two parts, the reduction tree, and the reverse tree. The reduction tree portion of the scan will perform $\text{SECTION\_SIZE} - 1$ operations, since it performs SECTION_SIZE /2 operations for the first step, N/4 for the second, and so on until finally performing N/x operations for the final step where $\frac{\text{SECTION\_SIZE}}{x} = 1$. Adding the number of operation for each step would be equal to $\text{SECTION\_SIZE} - 1$. The reverse tree portion of the scan follows a similar trend where the first step performs $(\frac{\text{SECTION\_SIZE}}{x} - 1)$ operations where $\frac{\text{SECTION\_SIZE}}{x} = 1$. The final two steps for this tree would perform $\frac{\text{SECTION\_SIZE}}{4} - 1$ and $\frac{\text{SECTION\_SIZE}}{2} - 1$ operations, respectively. Adding all the operations in this tree will prove to be $\text{SECTION\_SIZE} - 1 - log_2(\text{SECTION\_SIZE})$. Finally, by combining the operations performed by the two trees in the Brent-Kung scan we get a total of $2 * \text{SECTION\_SIZE} - 2 - log_2(\text{SECTION\_SIZE})$ operations. Since this happens in every thread, we must multiple this expression by N / SECTION_SIZE to get, $(\mathbf{2 * SECTION\_SIZE - 2} - \boldsymbol{log_2}(\mathbf{SECTION\_SIZE})) * \mathbf{N} / \mathbf{SECTION\_SIZE}$.

We can now look at the number operations performed in steps 2 and 3 of my kernel. In step 2 the only FLOP being performed is when the first thread of every block adds its sum to the previous sum and stores the resulting sum in global memory for the next block. This will result in several operations equal to the number of sections, which can be calculated as **N / SECTION_SIZE**. In step 3 every thread performs two more operations by adding the previous sum collected in step 2 to the two elements its responsible for. Ultimately, this will result in another **N** operations.

Now to get the total number of operations performed by the tread we can add the operations done by the individual steps. Finally, we get $(\mathbf{2} * \mathbf{SECTION\_SIZE - 2} - \boldsymbol{log_2}(\mathbf{SECTION\_SIZE}) + \mathbf{1}) * \frac{\boldsymbol{N}}{\boldsymbol{SECTION\_SIZE}} + \boldsymbol{N}$.

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

Again, splitting the kernel in the three steps its takes, we will first look at the global memory reads performed by step one, the Brent-Kung scan. In this will simply result in **N** memory reads since every thread will read two values from global memory with no overlaps. In step 2 the first thread of every block will perform an atomic addition to the block id counter. Since atomic additions combine the acts of reading, adding, and writing a value it will count as a global memory read. Therefore, the read to the block id counter will be performed $N / SECTION\_SIZE$ times. Next the first thread of every block also reads the previous block's sum from global memory which accounts for another $N / SECTION\_SIZE$ reads. Then the thread reads the sum of its own block so it can add it to the previous sum and pass it to the next block. This again results in $N / SECTION\_SIZE$ global memory reads. Finally, the thread will increment the next block's location in the flags array so that the next block knows to continue. This final steps results in $N / SECTION\_SIZE$ more global memory reads. In total step two accounts for $\mathbf{4} * (\boldsymbol{N} / \boldsymbol{SECTION\_SIZE})$. Step 3 does not contain any global memory reads, so we can just add steps 1 and 2 to get the total amount of reads. Therefore, the total amount of global memory reads performed by my kernel is, $\boldsymbol{N} + \mathbf{4} * (\boldsymbol{N} / \boldsymbol{SECTION\_SIZE})$.

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

Like the question above, I will look at the steps one at a time. In step 1 the Brent-Kung scan performs two memory writes per thread. Since the block size is equal to $SECTION\_SIZE$ / 2 and the grid size is $N \ / \ SECTION\_SIZE$, this means there are $N/2$ threads. Therefore, step 1 results in $2(N/2) \ = \ \boldsymbol{N}$ writes. Like the calculation for global memory reads, each atomic operation performed also performs a global memory write. In step two the first thread of every block will perform 2 atomic operations for incrementing the block id and incrementing the next block's flag. Each of these operations will happen $N \ / \ SECTION\_SIZE$ number of times since this is the grid size. During step 2 the thread 0 of every block will also write to the sums array in global memory to record its sum for the next block. Again, they will occur $N \ / \ SECTION\_SIZE$ number of times. In total $3(N \ / \ SECTION\_SIZE)$ global memory writes will occur in step 2. Finally, in step 3 each thread will write the two elements its responsible for to the output array resulting in another $\boldsymbol{N}$ number of writes. The total number of global memory writes for the kernel can be found by adding the writes in each step. Therefore, there are $\mathbf{2} * \boldsymbol{N} \ + \ \mathbf{3}(\boldsymbol{N} \ / \ \boldsymbol{SECTION\_SIZE})$ total global memory writes.

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.

To express the minimum, maximum, and average number of real operations that a thread will perform I will look at each individual step of the kernel. Due to the boundary checks in the kernel, we can determine that the last tread in each block will do the minimum number of real operations. In step one this thread will only perform one real operation during the reduction tree loop and 0 operations in the reverse tree loop before exceeding the boundary limit. This thread will not do any operations in step 2 but will perform two more operations in step three when adding the previous sum to its assigned elements. This brings the minimum number of real operations performed to be **3** operations total.

While the last thread in each block performs the least real operations, the first thread in every block performs the most. In step one, the first thread will perform $\log_2(SECTION\_SIZE)$ operations in the reduction tree and $\log_2(SECTION\_SIZE) - 1$ operations in the reverse tree. This thread also performs a real operations in step 2 when adding its block's sum to the previous block's sum. Lastly, this thread also adds the previous sum to the two elements it is responsible for in its block's section. In total the maximum number of real operations performed by a thread will be $\mathbf{2}(\mathbf{log_2}(\boldsymbol{SECTION\_SIZE})) + \ 2.$

The average number of real operations performed by a single thread will correspond to the threads towards the second half of their blocks. These threads will only perform one operation in the reduction tree and one operation in the reverse tree. Also, these threads perform two more operations in step three of the kernel. This brings the average number of real operations performed by a single thread to be **4**.

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

Assuming the process of reducing a thread blocks portion of the array to a single value refers to calculating the section's sum we can primarily look at step 1 of the kernel. In step 1, the kernel performs a Brent Kung scan which will sync the threads at every step of the reduction and reverse trees. We can express the number of iterations in the reduction tree as, $\boldsymbol{log_2}(\boldsymbol{section\_size})$ and the number of iterations in the reverse tree as $\boldsymbol{log_2}(\boldsymbol{section\_size}) - \mathbf{1}$. The thread blocks also sync at the end of the Brent Kung scan to ensure that every thread has added its partially summed elements to global memory. This sync account for **one** additional sync for every thread block. Lastly, each thread block does another sync when getting its assigned block id for adjacent synchronization. The first thread in every block will split off to index the global block id counter and the rest will wait to sync. This accounts for **one** final sync before the block gets the previous sections sum and calculates its own final sum. Adding the number of sync from every section together, we get a total of $\mathbf{2(log_2(section\_size)) + 1}$ syncs.

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

    There are two major decisions I made that ultimately optimized my kernel. The first was my choice of which parallel scan algorithm I would implement in step 1 of my kernel. Ultimately, I chose the Brent-Kung method over the Kogge–Stone method since it is theoretically more work efficient. This main stems from the fact that the Brent-Kung uses only N/2 number of threads. The number of resources being used for the Brent-Kung can be measured by the following equation, $N/2 * (2 * log_2(N) - 1)$. While the resources used by the Kogge–Stone can be represented as $N * log_2(N)$. In an example where we are measuring the executing time decrease due to parallelizing the scan, with 1024 input units being processed by 32 execution units, we would see a 3.2 times speed up from the Kogge-Stone and a 3.4 speed up from the Brent-Kung. One can see that although the two methods perform similarly, the Brent-Kung produces better results. The second major optimization I made was choosing to implement a Stream-based scan method over a Hierarchical one. The hierarchical scan method is effective in providing accurate results; however, it relies heavily on global memory reads to transfer data between kernels. To combat this, the Stream-based method performs all the steps of the hierarchical method, but in one kernel. By working in this way, the blocks can pass their sum to the next block and continue working on adding the previous sum to their elements. In hierarchical method the blocks would have to pass their sums to global memory and then wait for all the other blocks to do the same so the next kernel can be launched. The blocks can get the previous sum and continue working in the Steam-based method due to the use of adjacent synchronization . One downside to the adjacent synchronization is that it assumes that the blocks are executing in a specific order. To solve this issue, dynamic block index assignment is used to assign the blocks consecutive ids at runtime. Overall, the implementation of the Stream-based method heavily reduces the number of calls to global memory which tend to have very long latency.

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

    One further operation that I could implement into my kernel, would be replacing my Brent-Kung scan in step one with a Three-phase parallel scan. This replacement would further complicate my code but would also enable the use of corner turning. Additionally, my step one would no longer be limited by the number of threads in a block. Instead, it will be limited by the size of the shared memory since all elements in a sections have to fit into shared memory. The Three-phase parallel scan works by dividing the blocks section of input vector into smaller sections for which each thread will be responsible. Then each thread helps load adjacent values into the shared memory until the whole block's section is stored. These adjacent memory calls are what enable corner turning for this method and enable memory coalescing. Now the first phase beings and each thread calculate the sums for its sub-section. Then in phase two, either the Kogge–Stone or Brent–Kung algorithm is used to scan that last element in every sub-section. This will result in the last element of every subsection having its final value. In the final phase each thread works to add the last value of the last element in the previous sub section to the elements in its sub section. In this step the threads ignore their final element since it already has its final value. Like stated early this method is not limited by the maximum number of threads in a block since it uses a much smaller number of threads. Instead, this method is limited by the size of the shared memory. To combat this limitation, I could implement a

Hierarchy-method rather than a Stream-based, to utilize global memory. This would drastically slow down my kernel, but it would allow me to handle must larger inputs. This decision depends on the data I have running the scan on, but for this lab I will be focusing on speed and I will use keep the Stream-based method.

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

   In the case where the input is greater than 2048*65535, my kernel would run into issues where the device cannot allocate the necessary number of threads required to run. As a response, I would have to modify my kernel to use a Hierarchical method rather than a Stream-based method. The Stream-based method that my kernel currently uses, relies heavily on storing information in shared memory. Although this has the benefits of less latency experienced from calls to memory, it also causes my kernel to be limited by the number of threads a device can launch in parallel. The Hierarchical method splits the same three steps of my kernel into separate kernels. This requires the separate kernels to now store everything in global memory so that information can be passed between the kernels. The reliance on global memory makes this method slower than the Stream-based method. However, when reaching the 2048*65535 element limit, the Hierarchical method can add extra levels to handle larger input sizes. The ability to add extra levels to the hierarchy enables this method to handle arbitrary input sizes.

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

    If the desired scan is using a binary operator that is not commutative, then a parallel version of the scan cannot be implement. If the operator being used is not commutative then it is assumed that the operations would be executed in a specific order. If the operations were done out of order they would result in incorrect values. Since, parallel executions cannot guarantee any specific execution order, we cannot guarantee that the non-commutative operations will be executed correctly.

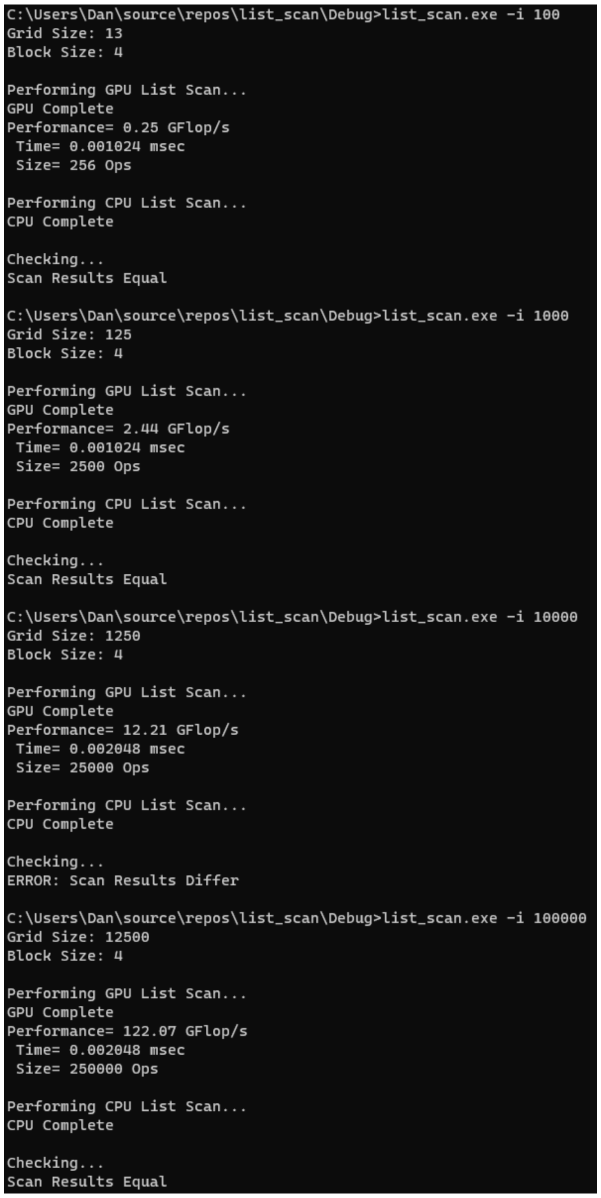
Experimentation

To test the performance of my kernel I ran tests with varying input vector size and section size. The section size effectively varies the size of the sub sections created which in turn effects the size and number of thread blocks being utilized. The input vector also influences these parameters, so varying both should allow me to observe a trend between the two.

	SECTION_SIZE = 8	SECTION_SIZE = 32	SECTION_SIZE = 128	SECTION_SIZE = 1024	SECTION_SIZE = 2048
VEC_SIZE = 100	0.25	0.32	0.34	2.09	3.96
VEC_SIZE = 1,000	2.44	2.79	2.91	2.97	4.96
VEC_SIZE = 10,000	12.21	13.75	14.45	14.83	14.85
VEC_SIZE = 100,000	122.07	137.33	287.05	292.60	293.08

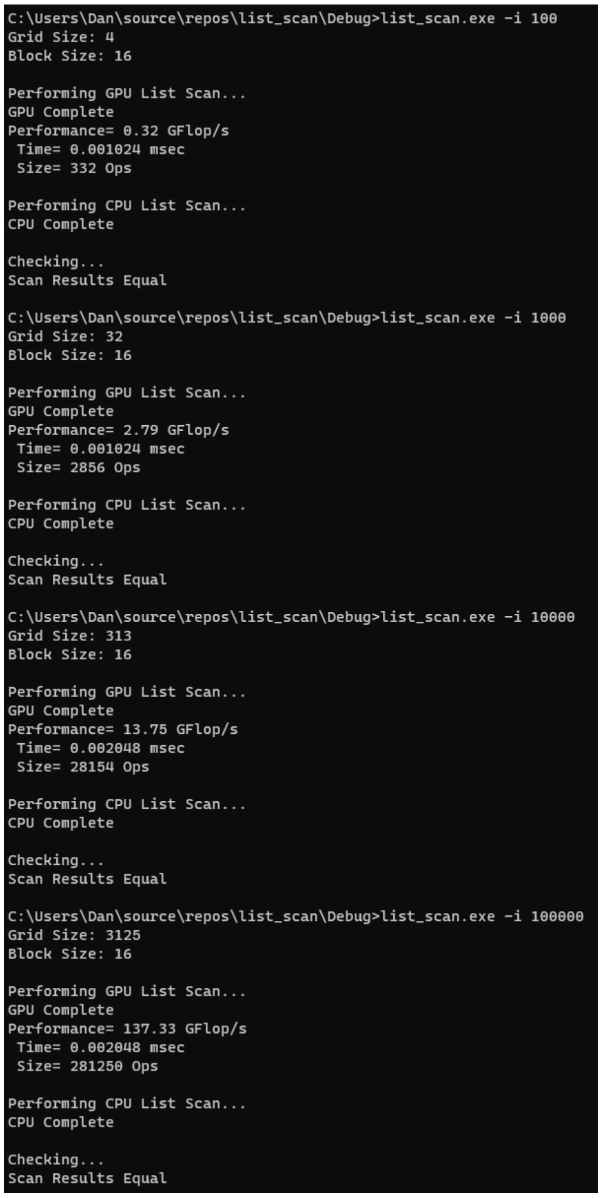
From the results of the tests above I can deduce that as input vector size increases the GFLOPs performance generally also increases. This occurs most likely because the kernel has many more operations to perform and the runtime does not increase very much due to the use of shared memory. We also see a similar trend when increasing the section size. As the section size increases so does GLOPs performance, since there are less calls to global memory when the blocks are sharing their sums. Since the section size is greater, there are less blocks used. Therefore, there are less section sums that need to be read and wrote to global memory. The effect of this decrease in global memory usage because very clear in the higher input vector sizes, since for these runs the number of blocks created for small sections is massive.

Images



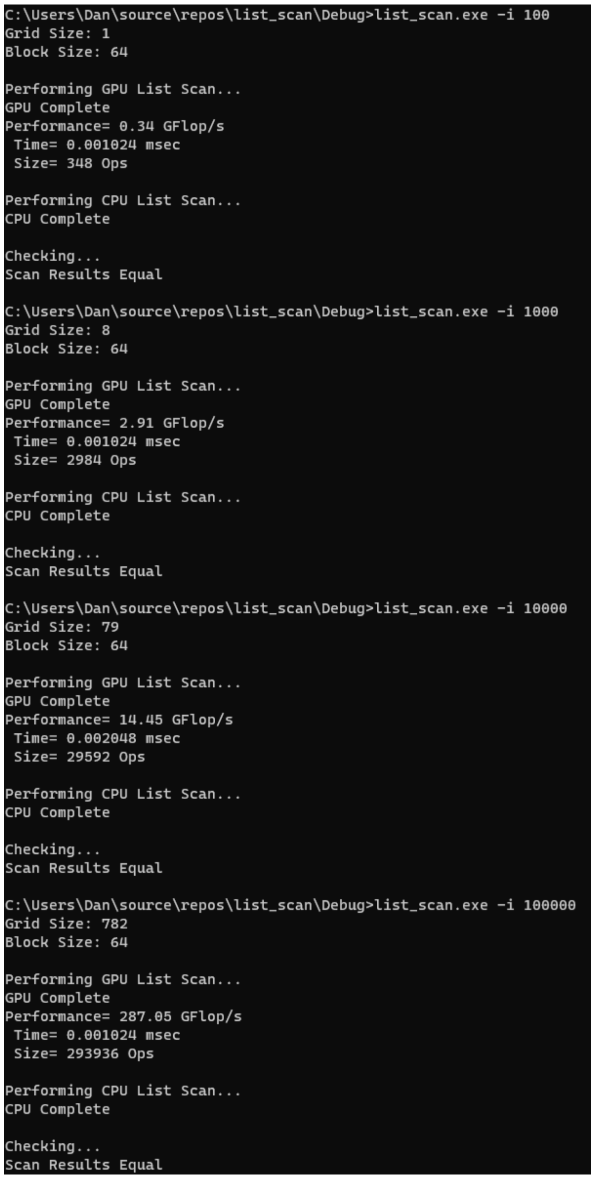
```
C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100
Grid Size: 13
Block Size: 4

Performing GPU List Scan...
GPU Complete
Performance= 0.25 GFlop/s
 Time= 0.001024 msec
 Size= 256 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 1000
Grid Size: 125
Block Size: 4

Performing GPU List Scan...
GPU Complete
Performance= 2.44 GFlop/s
 Time= 0.001024 msec
 Size= 2500 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 10000
Grid Size: 1250
Block Size: 4

Performing GPU List Scan...
GPU Complete
Performance= 12.21 GFlop/s
 Time= 0.002048 msec
 Size= 25000 Ops

Performing CPU List Scan...
CPU Complete

Checking...
ERROR: Scan Results Differ

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100000
Grid Size: 12500
Block Size: 4

Performing GPU List Scan...
GPU Complete
Performance= 122.07 GFlop/s
 Time= 0.002048 msec
 Size= 250000 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal
```

Section_Size = 8



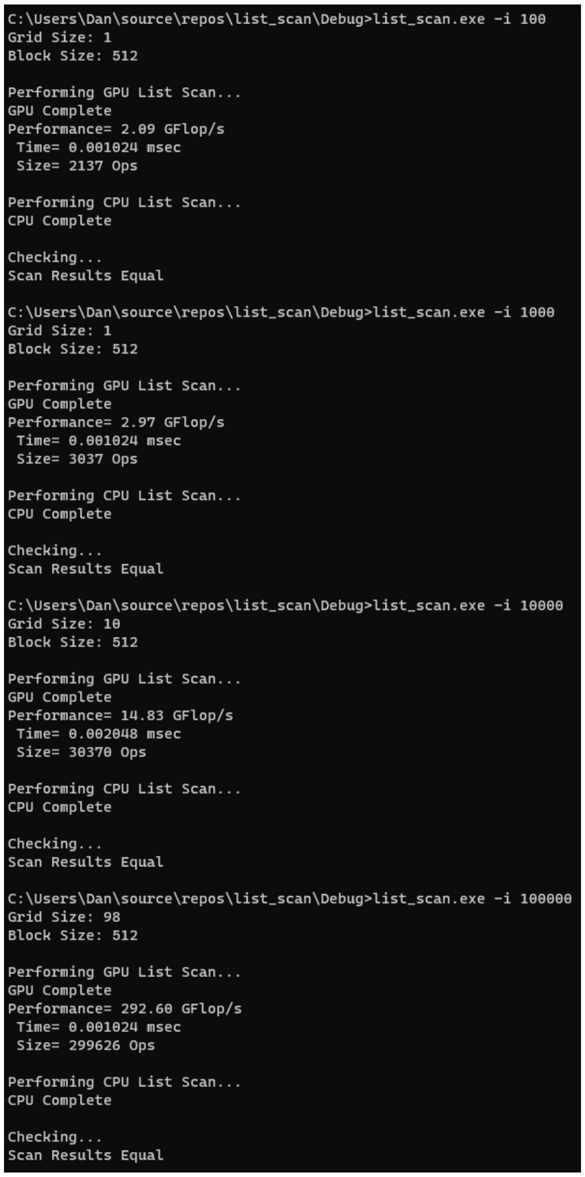
```
C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100
Grid Size: 4
Block Size: 16

Performing GPU List Scan...
GPU Complete
Performance= 0.32 GFlop/s
 Time= 0.001024 msec
 Size= 332 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 1000
Grid Size: 32
Block Size: 16

Performing GPU List Scan...
GPU Complete
Performance= 2.79 GFlop/s
 Time= 0.001024 msec
 Size= 2856 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 10000
Grid Size: 313
Block Size: 16

Performing GPU List Scan...
GPU Complete
Performance= 13.75 GFlop/s
 Time= 0.002048 msec
 Size= 28154 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100000
Grid Size: 3125
Block Size: 16

Performing GPU List Scan...
GPU Complete
Performance= 137.33 GFlop/s
 Time= 0.002048 msec
 Size= 281250 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal
```

Section_Size = 32



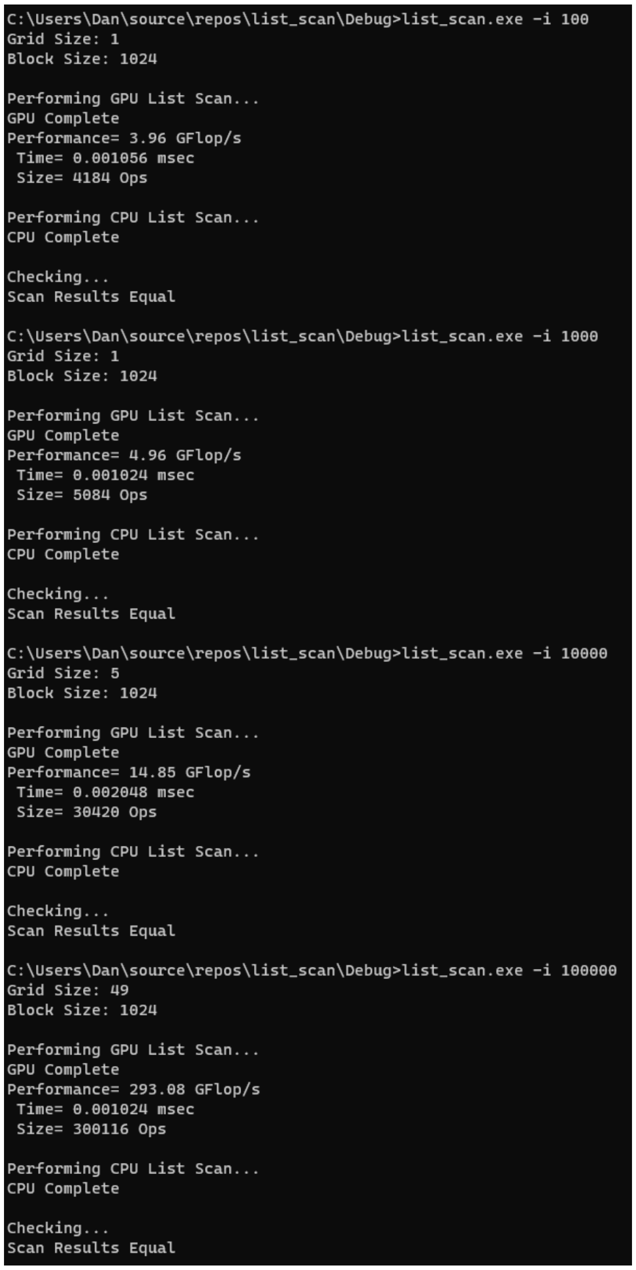
```
C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100
Grid Size: 1
Block Size: 64

Performing GPU List Scan...
GPU Complete
Performance= 0.34 GFlop/s
 Time= 0.001024 msec
 Size= 348 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 1000
Grid Size: 8
Block Size: 64

Performing GPU List Scan...
GPU Complete
Performance= 2.91 GFlop/s
 Time= 0.001024 msec
 Size= 2984 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 10000
Grid Size: 79
Block Size: 64

Performing GPU List Scan...
GPU Complete
Performance= 14.45 GFlop/s
 Time= 0.002048 msec
 Size= 29592 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100000
Grid Size: 782
Block Size: 64

Performing GPU List Scan...
GPU Complete
Performance= 287.05 GFlop/s
 Time= 0.001024 msec
 Size= 293936 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal
```

Section_Size = 128



```
C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100
Grid Size: 1
Block Size: 512

Performing GPU List Scan...
GPU Complete
Performance= 2.09 GFlop/s
 Time= 0.001024 msec
 Size= 2137 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 1000
Grid Size: 1
Block Size: 512

Performing GPU List Scan...
GPU Complete
Performance= 2.97 GFlop/s
 Time= 0.001024 msec
 Size= 3037 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 10000
Grid Size: 10
Block Size: 512

Performing GPU List Scan...
GPU Complete
Performance= 14.83 GFlop/s
 Time= 0.002048 msec
 Size= 30370 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100000
Grid Size: 98
Block Size: 512

Performing GPU List Scan...
GPU Complete
Performance= 292.60 GFlop/s
 Time= 0.001024 msec
 Size= 299626 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal
```

Section_Size = 1024



```
C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100
Grid Size: 1
Block Size: 1024

Performing GPU List Scan...
GPU Complete
Performance= 3.96 GFlop/s
 Time= 0.001056 msec
 Size= 4184 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 1000
Grid Size: 1
Block Size: 1024

Performing GPU List Scan...
GPU Complete
Performance= 4.96 GFlop/s
 Time= 0.001024 msec
 Size= 5084 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 10000
Grid Size: 5
Block Size: 1024

Performing GPU List Scan...
GPU Complete
Performance= 14.85 GFlop/s
 Time= 0.002048 msec
 Size= 30420 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal

C:\Users\Dan\source\repos\list_scan\Debug>list_scan.exe -i 100000
Grid Size: 49
Block Size: 1024

Performing GPU List Scan...
GPU Complete
Performance= 293.08 GFlop/s
 Time= 0.001024 msec
 Size= 300116 Ops

Performing CPU List Scan...
CPU Complete

Checking...
Scan Results Equal
```

Section_Size = 2048