Accelerated Computing
Dalton Bohning
Jordan Kremer

Prefix Sum Scalability Study

Introduction

Given an input array X, Prefix Sum (inclusive) produces an output Y of the running total of each index. For example: the output of [1,2,3] would be [1,3,6]. The simple approach is to iteratively traverse the input array X and set Y[i] equal to the cumulative sum for each index i. For large input sizes, it would be beneficial for this work to be done in parallel. More importantly, the scalability of any such approach would need to be studied. In this report, we detail our experiences and findings in implementing a Brent-Kung algorithm [1] [2] in CUDA and a similar algorithm with OpenMP [3]. The performance of each of these approaches is compared to each other and to an iterative implementation of Prefix Sum. We then discuss some drawbacks in our approach as well as other considerations.

Brent-Kung Paper

Brent and Kung [1] discuss a method of adding two *n*-bit binary numbers in parallel. This method focuses on implementing binary adders and can be generalized as a method of adding some *n* values in parallel. Brent and Kung provide a nice diagram for this [1] in Figure 1. On the surface the idea is simple: asynchronously place the addition of every other two values into the upper, place the addition of each of those values into the upper, etc. until the *n/2* values are added together and placed into *n*-1. For example, in parallel, the additions by index would be {[0]+[1], [2]+[3], …, [n-2]+[*n*-1]}, followed by {[1]+[3], [5]+[7], …, [n-3]+[n-1]}, …, {[n/2-1]+[n-1]}. This step is illustrated by the bottom half of the tree in Figure 1. The next step is to add the last *n/4* values together, and then the *n/8* values, etc. until every other 2 values are added. For example, in parallel, the additions by index for this step would be {[n*/2*]+[n-n*/4*-1]}, followed by {[n*/2*]+[n*/2+2*], [n-n*/4*-1]+[n-n*/4*+1], … [3]+[5]}, …, {[1]+[2], [3]+[4], … [n-3]+[n-2]}. This second step is needed in order to “carry through” the lower sums to the higher levels, since they were calculated in parallel and are relied upon recursively. This step is illustrated by the upper half of the tree in Figure 1.

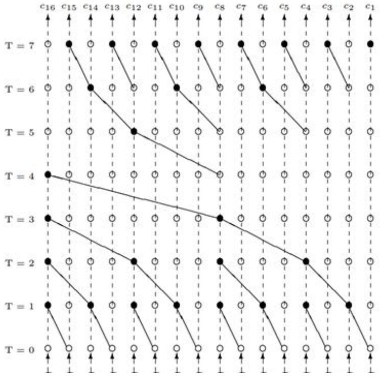


Figure 1. Brent-Kung 16-bit adder [1]

CUDA Implementation

Following Brent and Kung [1] and the textbook in chapter 8 [2], we implemented the Brent-Kung algorithm in a hierarchical fashion in CUDA. The algorithm is done in-place on the device. Performing the computation in-place as opposed to creating both an input X and output Y on the device gives us: (1) better performance, because there is less data movement, and (2) allows us to further scale the size of the input array with consideration of space restrictions.

The implementation consists of three phases. The first phase breaks the input array into equal-sized sections. Each CUDA block calculates the partial sums for one section based on the Brent-Kung algorithm previously discussed. The high-index partial sums for each section are stored in another array. Phase 2 executes the Brent-Kung algorithm on this array of partial sums. Phase 3 adds each of the values from phase 2 back to the values in phase 1 for the corresponding section. If necessary, phase 2 recursively executes phases 1, 2, and 3 when the partial sums themselves do not fit into a single CUDA block. Figure 2 illustrates this idea [2].

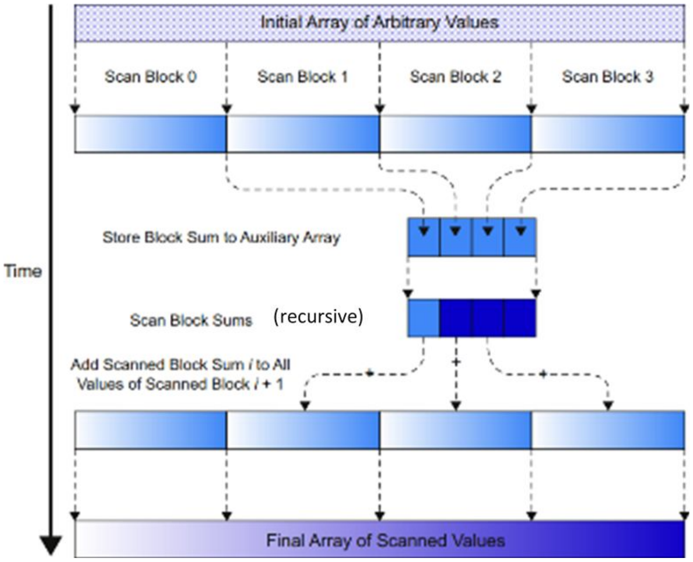
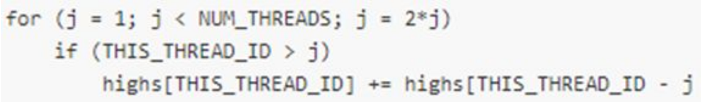

Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Time
Scan Block Sums
(recursive)
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
+
+
+
Final Array of Scanned Values

Figure 2. Hierarchical Brent-Kung [2]

OpenMP Implementation

We started with the implementation given in [3] and made minor adjustments for correctness. The backbone of the OpenMP implementation is similar to the CUDA implementation. The first step distributes the input array X equally to T threads. Each thread iteratively calculates the Prefix Sum for its section and stores the high value in a separate array. In the second step, these high values are “pushed up” in parallel, to calculate the Prefix Sum for the array of high values. Figure 3 shows the simplified code for step 2. Note that this does not include the necessary synchronization barriers. The third step adds the values from step 2 to the partial values from step 1, similar to how it is done in the Brent-Kung hierarchical implementation. Whereas the CUDA algorithm recursively performs phase 2, the OpenMP algorithm does not recursively perform step 2, since there is not a large number of threads. In fact, step 2 is a partial, simpler Prefix Sum implementation that works well on smaller inputs.



```
for (j = 1; j < NUM_THREADS; j = 2*j)
    if (THIS_THREAD_ID > j)
        highs[THIS_THREAD_ID] += highs[THIS_THREAD_ID - j
```

Figure 3. OpenMP - Step 2 without barriers

Scalability Study

Machine Specifications

We utilized two different machines for benchmarking the scalability of each implementation. The first was wino.cs.pdx.edu, which has a Quadro K620 and Intel(R) Xeon(R) CPU E3-1241 with 4 hyperthreaded cores. The second machine was babbage.cs.pdx.edu, which has 30 Intel Xeon E3-12xx v2 cores but unfortunately is not CUDA capable.

Testing Procedure

We ran the CUDA implementation on Wino and the OpenMP implementation on both Wino and Babbage. The iterative version was run on both systems and is compared against. Wino is expected to have better single-core performance, but Babbage has a much greater number of cores. Testing on both machines helped us understand the impact of additional cores even if they were comparatively slower core per core. Each implementation was executed on array sizes ranging 2^8 to 2^{28} . For the CUDA implementation, we tested section sizes of 1024 and 2048 and measured the total execution and memory execution times. For the OpenMP implementation, we tested and measured the total execution time for thread counts of 2, 4, 8, 16, 32. We also ran a simple iterative implementation of prefix sum for comparison. Each test was run three times with the median values taken for each sample.

Results and Analysis

Brent-Kung in CUDA

While testing, the execution time of the Brent-Kung implementation for section sizes of 1024 and 2048 for various array sizes was compared (*Figure 4*). Implementation with a section size of 1024 performed better than the section size of 2048 by a slight margin, 3.9% faster on array size 2^{28} , as the array size increased. 1024 is the reference section size in all subsequent discussion because this implementation proved to be superior.

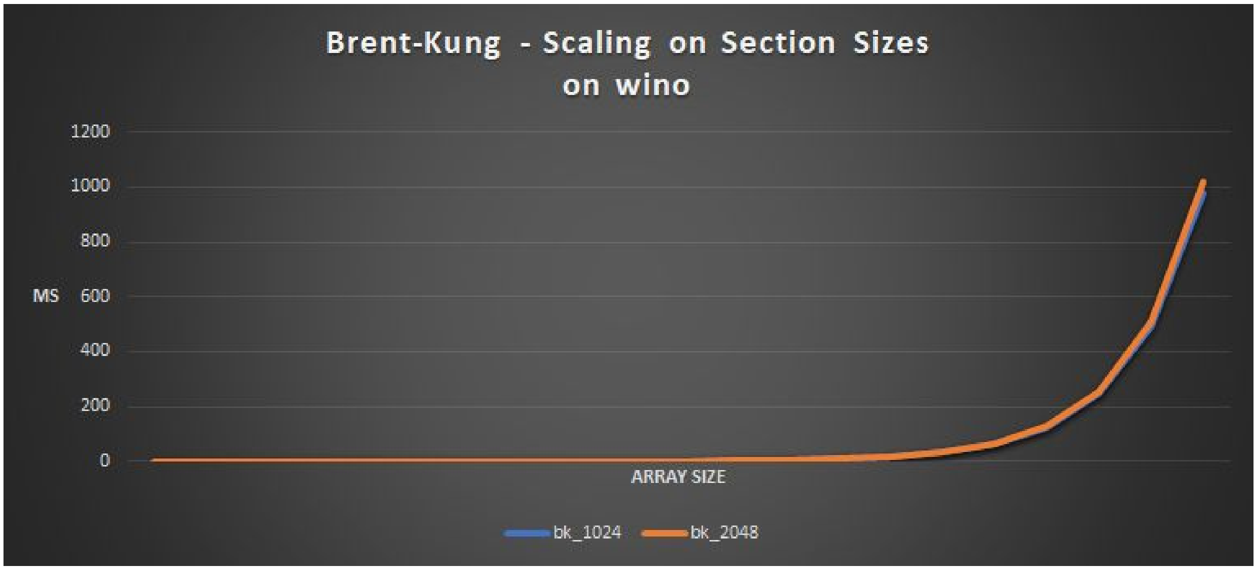


Figure 4

The iterative prefix sum initially outperforms the CUDA version (*Figure 5*), but as the size of the array approaches 2^{14} (16,384) the execution time of Brent-Kung outpaces the iterative version. However, the memory transfer time between the host and device is large enough to cause the iterative version to perform better overall. At 2^{16} elements, the CUDA implementation performs with a total time of 0.68 ms versus the iterative implementation speed of 0.44 ms. The iterative implementation at this array size is 35% faster than the CUDA version.

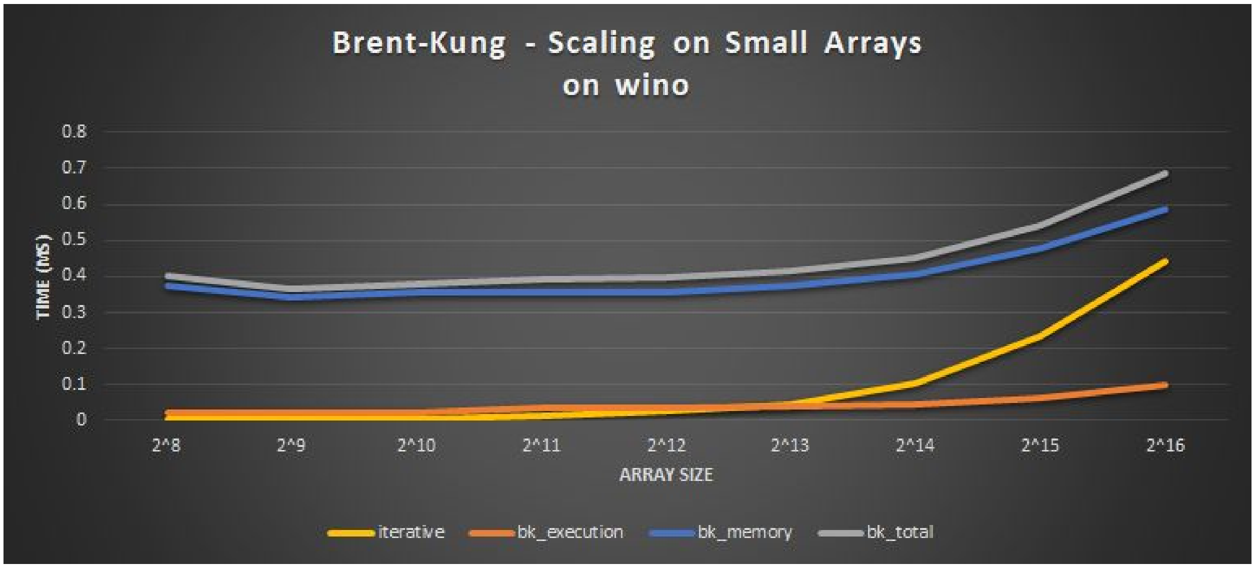


Figure 5

Execution time (ms) of CUDA implementation versus iterative

Measured	Time (ms) at 2^{16}	Time (ms) at 2^{28}
Iterative	0.443	900.15
CUDA Kernel Execution	0.099	339.16
CUDA Memory	0.584	639.06
CUDA Total	0.685	978.21

Table 1

The overall execution time of the Brent-Kung implementation is slower than the execution time of the iterative implementation as the array size increases to 2^{28} (*Table 1*).The data represented (*Figure 6*) shows scaling for arrays up to size 2^{28}. Arrays larger than 2^{28} were too large to be tested on because a GPU memory bottleneck was observed. At 2^{28} elements the CUDA kernel execution time was 339 ms, the memory execution time was 639 ms, the total CUDA execution 978 ms, and the iterative version 900 ms. As the array size grows the difference between the iterative execution and the CUDA total execution time grows.

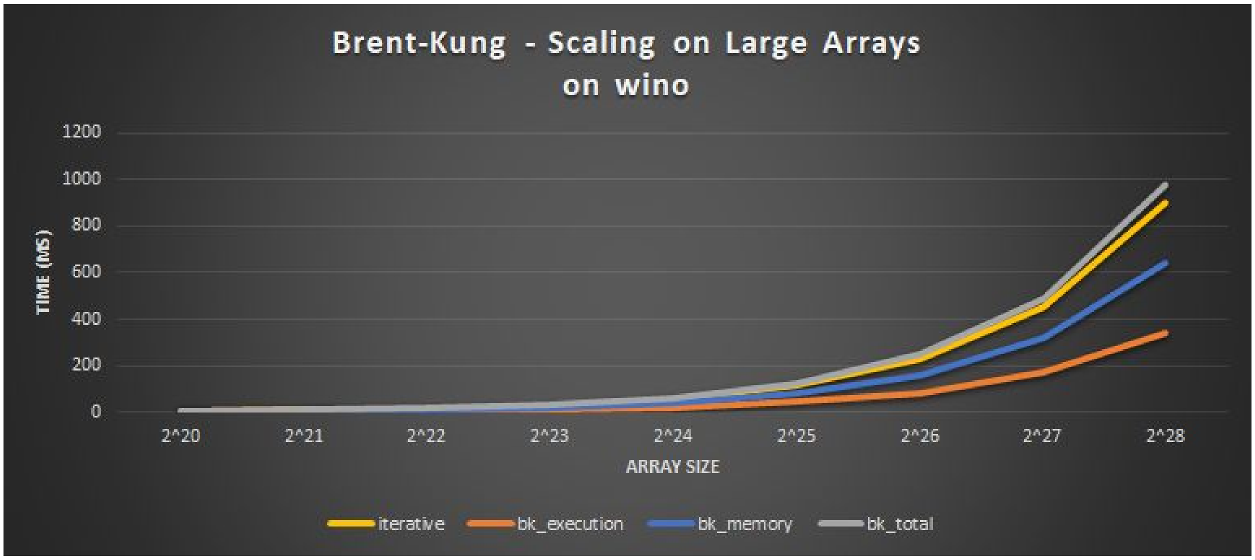


Figure 6

OpenMP implementation

For array sizes up to 2^{16}, the iterative implementation outperforms the OpenMP version on the babbage system (*Figure 7)*. However, as the array size increases the OpenMP version begins to outperform the iterative version. Increasing the number of threads reduces the performance in the trials. This data suggests that the overhead of threads is significant at this scale.

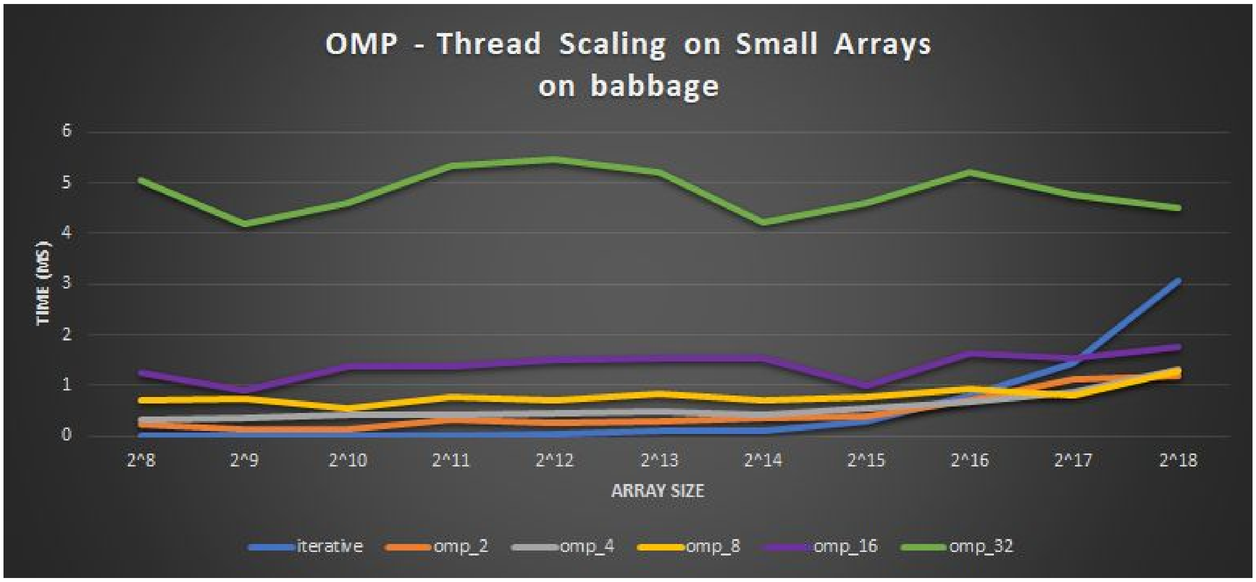


Figure 7

Execution time (ms) OpenMP vs. Iterative on Babbage

Implementation	Iterative	8 threads	16 threads	32 threads
2^{12} elements	0.0019	0.4249	1.5782	4.1141
2^{19} elements	3.6702	1.3661	1.7266	4.7177
2^{20} elements	7.4759	2.3213	1.8444	5.0126
2^{28} elements	1679.7148	268.2350	219.9445	152.5732

Table 2

As the array sizes increase up to 2^{28}, the performance trends reverse (*Table 2, Figure 7 and Figure 8*). The iterative version performed 11x slower than the OpenMP implementation at 2^{28} elements. 32-thread execution on the 30-core babbage system yielded the best performance at large array sizes, but performed the worst at lower array sizes ($2^8 - 2^{19}$). 32-threads began to show improvement over the iterative implementation at 2^{20} elements. As the array size grew, the 32-thread implementation showed the least increase in time. This increase is characterized with the smallest slope (*Figure 8*).

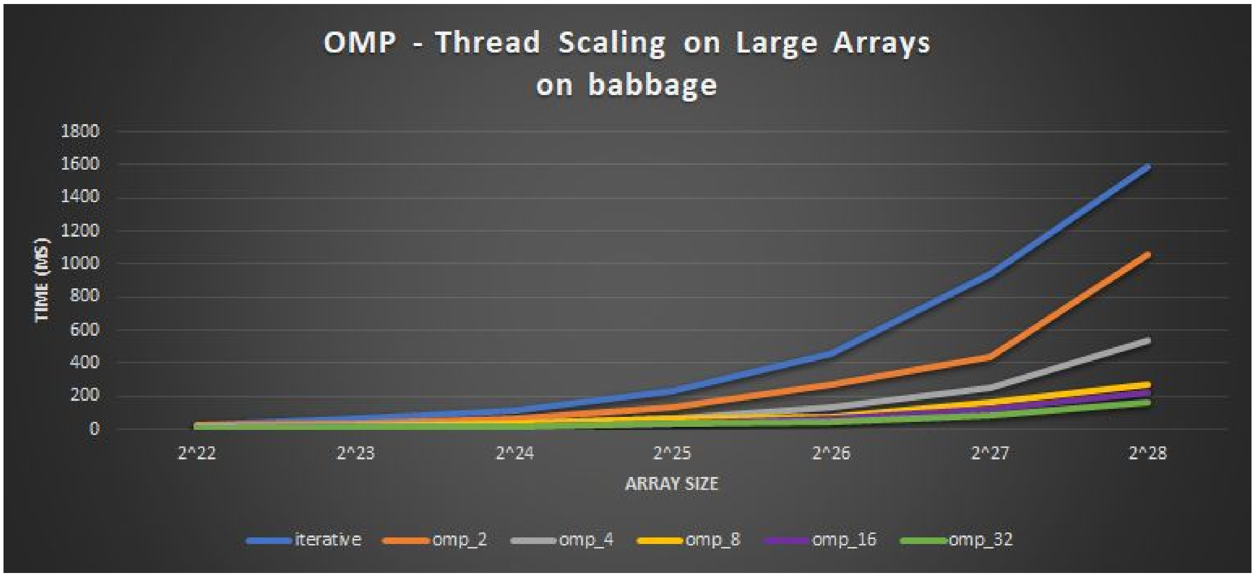


Figure 8

Since we had multiple machines on which to test the OpenMP implementation, we decided to do a comparison of the two. The wino system generally has better single-core performance, while the babbage system has more cores. At 2^{28} elements, Wino executed the prefix sum iteratively at 900 ms versus 1679 ms on Babbage (*Figure 7*). The Wino system outperforms Babbage up to 8 threads (*Figure 9*). This is not surprising, since Wino has 4 hyperthreaded cores. After 8 threads, babbage was able to continue improving performance, whereas Wino was not. This data suggests a tradeoff between machines with a few powerful cores and machines with a lot of weaker cores for doing large scale computation. The overhead from spawning multiple threads slows down the computation and does not make up for the extra time until a certain array size is reached, depending on how many threads are spawned. For the case of the 32-thread implementation, the overhead from the increase in threads became less significant as the array size approached 2^{20}.

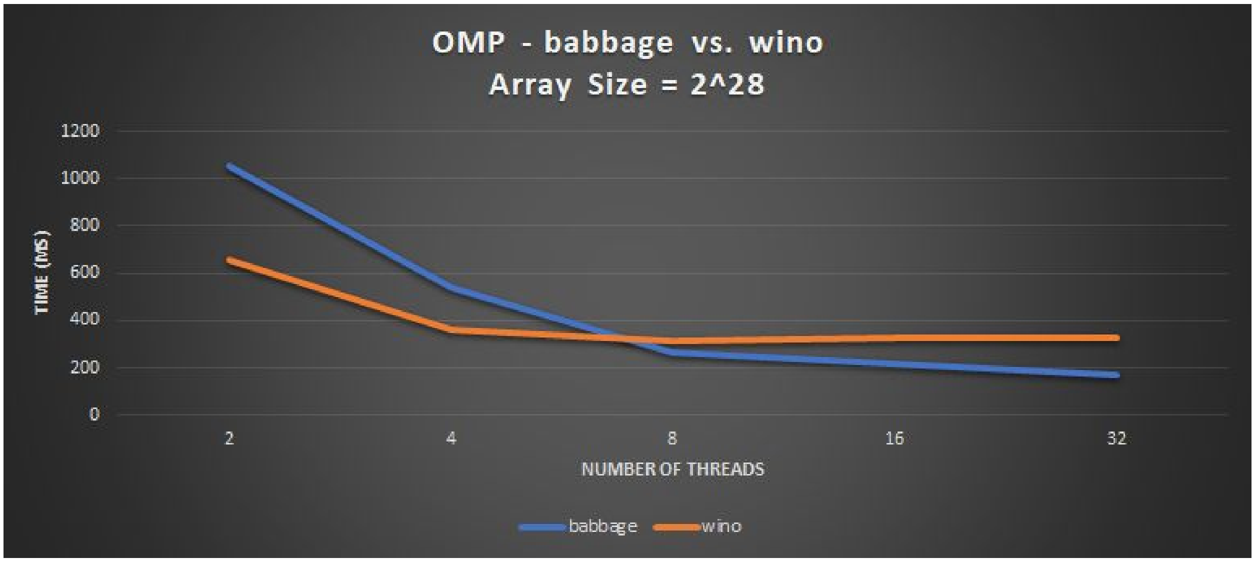


Figure 9

OpenMP versus CUDA versus Iterative

The Brent-Kung CUDA implementation performed worse than the iterative and 8-threaded OpenMP implementation on Wino, and worse than the 32-threaded OpenMP implementation on Babbage (*Figure 10, Table 3*). On an array size of 2^{28} , the 32-threaded implementation on Babbage performed 5.9x faster than the iterative, 6.4x faster than Brent-Kung, and 5.6x faster than 2 threads. For array sizes less than 2^{14} , the iterative version on Wino performed the best. Multithreaded implementations began to overcome to the iterative implementation for all arrays larger, with 32-thread implementation becoming the fastest at 2^{27} elements.

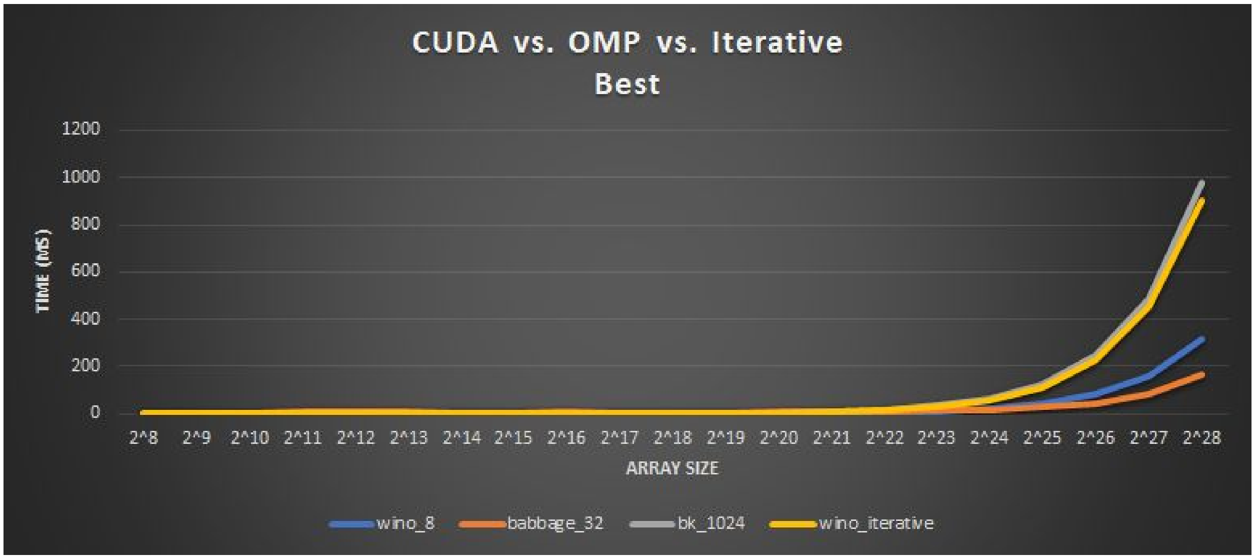


Figure 10

Execution Time (ms) of the Best Implementation for Each Method

Implementation	(Wino) Iterative	Wino OpenMP (8 threads)	Babbage OpenMP (32 threads)	CUDA
2^{12} elements	0.0278	0.3184	4.1141	0.3958
2^{19} elements	3.2288	1.0507	4.7177	2.6542
2^{20} elements	3.6521	2.6203	5.0126	4.8038
2^{28} elements	900.1491	316.7446	152.5732	978.2174

Table 3

Other Considerations

Hardware and Algorithms

It should be noted that this is not a robust comparison of the algorithms because we are comparing a specific GPU to specific CPUs with different capabilities. Better hardware and better algorithms for both CUDA and OpenMP likely exist which may help further analyze the difference between CPU and GPU implementations of parallel scan. The development effort and programming complexity of the Brent-Kung implementation was much higher than that of the OpenMP implementation. As argued by Vuduc [4], this problem may be a case where the additional complexity of CUDA does not yield a proportionate result in terms of speed and efficiency with the given hardware.

Hardware Availability

The Quadro K620 GPU only has 2 GB of GDDR3 and a compute capability of 5.0. This prevented us from scaling the array sizes beyond 2^{28} , and meant that the memory transfer between the host and device introduced a huge bottleneck. It is likely that a newer GPU would transfer memory more efficiently and increase the speed of the CUDA implementation.

At most, we had access to 30 cores for the OpenMP implementation, which prevented us from seeing any noticeable improvements beyond 32 threads. It would be interesting to see how our implementation scales beyond 32 threads, but this would require a machine with more cores. With a newer GPU and more processor cores, we could better assess the current state of the CUDA versus OpenMP implementations.

Precision of Data Types

An interesting issue was encountered when using floats in our calculations. The float data type is 32 bits, with 24 of those bits being used for the integer portion of the number. This means that only integers up to 16,777,216 and decimals that are powers of 2 can be accurately represented with a float. This issue became more apparent as we increased our array sizes to 2^{28} . In particular, the different implementations of Prefix Sum output slightly different values for higher array indices, since the rounding occurred in a different order. This suggests that a larger datatype, such as a 64-bit double, may be more appropriate for these larger input data sets. Of course, this would introduce an extra overhead that would need to be managed. For our purposes we avoided this issue altogether by (1) only assigning integer-equivalent values to our floats and (2) setting every 100th index to 1 with all others set to 0. Using this method, a value of at most 2,684,354 would be produced by an input array of size 2^{28} , which can be precisely represented in the 24 bits provided in the float data type.

Input Array Sizes

In testing each of the implementations, only arrays with input sizes that are powers of 2 were considered. This is not a realistic assumption that can be made about a general data set. It is possible that this assumption more strongly favored one of the Prefix Sum implementations, and that different array sizes would reveal further differences between the approaches. For example, it is possible that an input array of size 100 million would cause differences based on caching and memory behavior because the hardware typically operates on powers of 2.

Concluding Remarks

Our experiences implementing and testing have taught us that this type of study may involve unanticipated factors. There is not a simple way to choose one implementation that works best in all cases. An iterative approach to programming, data gathering, testing, validating, and analyzing the data proved to be the best approach. Moreover, others may choose to define the problem and approach it differently. While we found that the CUDA version of the Brent-Kung code performed worse than the iterative and OpenMP versions, we did not exhaustively study different implementations of the algorithm in CUDA and compare that data to the data we gathered.

Future work could include testing this CUDA implementation on Nvidia devices with different capabilities to gain insight into how the device architecture affects the results. OpenACC, a compiler directive that parallelizes code, may prove to be an efficient means of speeding up GPU code. OpenACC was not considered for this series of testing so no data was gathered on the efficiency of these compiler directives. As well, Nvidia currently has what is called unified memory. Unified memory essentially removes the need for explicit memory copies and allows data to be loaded to the GPU on demand; this overlaps memory copies with other computations [5]. Utilizing this unified memory on newer devices may yield better results with the CUDA implementation because of the more efficient memory transfer.

References

[1] Brent, and Kung. “A Regular Layout for Parallel Adders.” IEEE Transactions on Computers, C-31, no. 3, 1982, pp. 260–264., doi:10.1109/tc.1982.1675982.

[2] Kirk, David, and Wen-mei Hwu. Programming Massively Parallel Processors: a Hands-on with CUDA. Morgan Kaufmann Publishers, 2010.

[3] Engelen, Robert. Synchronous Shared Memory Parallel Examples. 2017, www.cs.fsu.edu/~engelen/courses/HPC/Synchronous.pdf.

[4] Vuduc et al, "On the Limits of GPU Acceleration," USENIX HotPar 10.

[5] Sakharnykh, Nikolay. “Maximizing Unified Memory Performance in CUDA.” NVIDIA Developer Blog, 2018, devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/.