Parallel Subset Sum

Laura Yao and Devin Qu

Link to Code: https://github.com/gss223/15418-project

Summary

We implemented and optimized CPU and GPU-based parallel algorithms to solve the subset sum decision problem for positive inputs. The CPU implementation uses OpenMP and AVX2 vector instructions to achieve up to 7.9x speedup on 8 threads and ~35x speedup on 64 threads on Bridges-2. The GPU implementation uses CUDA and achieves up to 12.34x speedup on the GHC machines.

Background

The subset sum problem (SSP) is an NP-hard problem in computer science where there is a multiset of integers S and an integer T, and one must determine whether there exists a subset of S whose sum is T. When all inputs are positive, the problem is known to be NP-complete [1]. The optimization version (i.e., determine the subset with the greatest sum that is most T) is a special case of knapsack and has applications in resource allocation. In general, the computational complexity depends on the number of integers N and the target sum T. Being NP-hard, even the best algorithms are exponential in the size of S or T.

One class of deterministic algorithms is those exponential in N. For instance, the most naive approach is to brute force every subset of S and check whether it sums to T, giving a running time of $O(N \bullet 2^N)$. Horowitz and Sahni published an improved algorithm based on meet-in-the-middle in 1974 with complexity $O(\frac{N}{2} \bullet 2^{N/2})$ at the cost of $O(2^{N/2})$ space [2]. Rather than generating all possible sums for S, it partitions S into two halves of $\frac{N}{2}$ elements and generates a sorted set of possible sums in each half. Determining the solution can then be found in $O(2^{N/2})$ time using two pointers, where one pointer iterates through one of the halves in ascending order and another pointer iterates through the other half in descending order.

Another class of algorithms is exponential in the size of T (or polynomial in the value of T) and uses dynamic programming. In general they compute, for each state $(i,\ s)$ for $0 \leq i \leq N$ and $0 \leq s \leq T$, whether a subset of the first i integers in S sums to s. As a base case, $(i = 0, s = 0)$ is true, as the sum of an empty set is 0. Then, for each integer S_i for $1 \leq i \leq n$, if (i, s) is true, $(i + 1, s)$ and $(i + 1, s + S_i)$ are true, representing not including and including S_i in the subset, respectively. This algorithm works in time $O(NT)$ since there are NT states and transitions are made on $O(1)$ time. In 1999, Pisinger discovered a $O(N \bullet max(S_i))$ algorithm for knapsack, which can be applied to SSP [3].

The exponential algorithms are fairly straightforward to parallelize: they involve computing the sum of many subsets with no dependencies, meaning that these sums can be computed in a data-parallel fashion. Additionally, they have good locality, since almost all of the memory accesses are to the input set of numbers. However, they are only viable on the smallest of inputs. On the other hand, while the pseudo-polynomial dynamic programming algorithms are much faster in practice, they are much more difficult to parallelize. The iterations are inherently sequential and each iteration is dependent on the previous one, since the algorithms use each input integer to update all of the possible sums before moving on to the next. SIMD execution is possible, since the transitions are to either add an input integer or not, so several sums can be updated at once. For example, if we are currently on integer S_i and sum s we can update the value of $s,\ s\ + 1, s + 2,\ldots, s + 7$ simultaneously using the results $s - S_i, s - S_i + 1, s - S_i + 2,\ldots,\ s - S_i + 7$, assuming we store results as 32-bit integers and use 256-bit vectors. This results in a nice constant factor speedup but doesn't solve the dependency issue and thus we have no way of taking advantage of multiple cores. Thus, we will need some way to create parallel work.

Approach

CPU

Initial Ideas

The main observation behind the parallel CPU approach is that the order elements of the set are added together doesn't matter, we only wish to compute whether or not the target sum is possible. So we can divide the original set into several disjoint subsets, compute the set of possible sums in each subset, and combine these results to determine the final result. Ignoring this combination step for now, our initial approach was to begin with the entire set S (stored as an array), and recursively compute the set of possible sums in its left and right halves before combining these into the overall set of possible sums. Being a divide and conquer algorithm, we started with using OpenMP's tasks for the recursive calls, stopping the recursion once the subarray length fell under a threshold. At this point we could directly run the standard dynamic programming algorithm on the subarray to compute the set of possible sums.

Another idea we briefly tried was to run the naive $O(N2^N)$ algorithm at very small subarray sizes such as those at most 20. The idea was that for large target sums, this time complexity would be better than $O(NT)$. However, we quickly realized that on larger inputs we would never want our recursion to reach this small of a subarray and that it was almost strictly better to revert to the dynamic programming algorithm at a larger subarray size.

However, using tasks introduces the overhead of task management, and since we are setting a threshold for the recursion, we already know that the recursion will stop at a certain subarray size. This means that the naive algorithm will be called on roughly the disjoint subarrays of our threshold length. So instead of a recursive divide and conquer algorithm, we can solve it iteratively by dividing the original array evenly

among the threads using OpenMP's parallel for, having each thread run the naive algorithm, and then combining these results. Combining the results could be done in a manner similar to the recursive algorithm but in reverse order: we merge the results within adjacent pairs to get a new set of results (representing the set of possible sums from pairs of threads), then merge the results of adjacent pairs of those (the set of possible sums from groups of 4 threads), and so on until we are left with a single array of possible sums. Then all that is left is to check whether T is possible. This can be visualized in Fig 1.

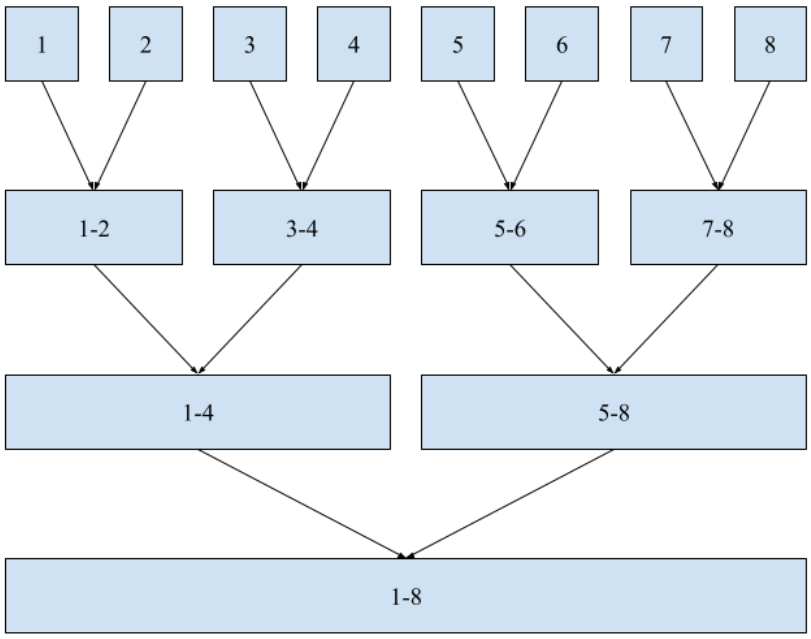

Figure 1. Iterative, bottom up approach to SSP using 8 threads. Each box represents the solution for some subarray (i.e., the set of possible sums for the subarray) and the number represents which threads' integers are used (e.g., the box with 1-4 stores the set of possible sums using integers from the first 4 threads. Initially, each thread computes the set of possible sums for an independent subset of integers, and the results are gradually combined together.

Within each of these "layers", we use OpenMP's parallel for with static scheduling to combine multiple pairs of results simultaneously, avoiding the overhead of using tasks.

Combining Results

The subproblem of combining results can be formulated as: given two boolean arrays a and b of length $T + 1$, compute the boolean array c such that $c_k = true$ iff there exists $0 \leq i, j \leq T$ where $i + j = k$, $a_i = true$, and $b_j = true$ for all $0 \leq k \leq T$. This is expressed succinctly in Fig. 2.

$$c_k = \bigcup_{i+j \; s.t. \; 0 \leq i,j \leq T \; and \; i+j=k} a_i \cap b_j$$

Figure 2. The process of combining results from boolean arrays a and b into a new array c.

However, instead of using boolean operations, if we treat 0 as $false$ and any positive integer as $true$, we can instead calculate c using addition in place of boolean OR and multiplication in place of boolean AND.

$$c_k = \sum_{i+j \; s.t. \; 0 \leq i,j \leq T \; and \; i+j=k} a_i \bullet b_j = \sum_{i=0}^{k} a_i \bullet b_{k-i}$$

Figure 3. The process of combining results in terms of arithmetic operations.

We can notice that if we think of a and b as arrays of integers instead of booleans, the definition of c is exactly that of polynomial multiplication, where a and b are arrays storing the coefficients of two polynomials (i.e., we are multiplying the polynomials $a_0x^0 + a_1x^1 + ... + a_Tx^T$ and $b_0x^0 + b_1x^1 + ... + b_Tx^T$). In this case, c will store the coefficients of their product, where $c_s = 0$ iff it is not possible to form sum s and c_s is nonzero iff it is possible to form sum s. This means that to compute the final set of possible sums, we repeatedly multiply a pair of arrays storing possible sums until we are left with only one array. Unfortunately, naively multiplying polynomials of degree n takes time $O(n^2)$, and since our arrays are of length $O(T)$, combining one pair of results will take time $O(T^2)$. This needs to

be done roughly twice the number of threads times, and if T is large, this ends up slower than simply running the naive algorithm on the entire input array, even if evaluated in parallel.

DFT

In order to multiply polynomials faster than $O(n^2)$, there are two important observations. First, a degree d polynomial can be uniquely defined by $d + 1$ distinct points. Second, if all we care about is the value of the product at a point x, this can be computed as $c(x) = (a \bullet b)(x) = a(x) \bullet b(x)$ [4]. In our case, we have two degree $T + 1$ polynomials, and their product will have degree $2T + 2$, so in order to multiply them we need $2T + 3$ distinct points. To simplify things for later, we round the $2T + 3$ up to the nearest power of 2, called n. So in order to multiply a and b, we need to evaluate them at n distinct points, and we pad our arrays a and b with 0 coefficients until they have length n. A convenient set of points to use are the nth roots of unity, or solutions to the equation $w^n = 1$. These can be expressed in the form of $exp(\frac{2\pi ik}{n}) = cos(\frac{2\pi k}{n}) + sin(\frac{2\pi k}{n})i$ for $0 \leq k < n$. A very nice property is that all of the roots can be expressed in terms of the principle nth root, $w_n = w_n^1 = exp(\frac{2\pi i}{n})$. Thus, we have the n roots $w_n^0, w_n^1, ..., w_n^{n-1}$. Now, we must evaluate a and b at these n points, effectively turning our polynomials from their "vector of coefficients form" into a "vector of values form". After this we get new arrays $a' = [a(w_n^0), a(w_n^1), ..., a(w_n^{n-1})]$ and $b' = [b(w_n^0), b(w_n^1), ..., b(w_n^{n-1})]$. We can express this as a matrix-vector product, giving us the Discrete Fourier Transform, as shown in Fig. 4.

$$\begin{bmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Figure 4. The DFT of a sequence a of length n

Then by multiplying the values of a' and b' at the n roots, we get our desired product c in value form, where $c_i = a_i \bullet b_i \ \forall\ 0 \leq i < n$, taking time $O(n)$. It turns out that since the DFT matrix is invertible, we can also convert a polynomial from value form into coefficient form by multiplying the vector by the inverse of the DFT matrix, as shown in Fig. 5. We can see that the inverse DFT matrix is almost the same as the DFT matrix, except the exponents are negated and there is a $\frac{1}{n}$ term. Using properties of sin and cos, we see that $w_n^{-k} = exp(\frac{-2\pi ik}{n}) = cos(\frac{-2\pi k}{n}) + sin(\frac{-2\pi k}{n})i = cos(\frac{2\pi k}{n}) - sin(\frac{2\pi k}{n})i$. This means if we have an algorithm for the DFT, we can just reuse it for the inverse DFT with slightly different roots and dividing all terms by n at the end.

$$\frac{1}{n}\begin{bmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Figure 5. The inverse DFT of a sequence y of length n

Using the DFT and inverse DFT, we can express our polynomial multiplication as shown in Fig. 6.

$$c = a \bullet b = InverseDFT(DFT(a) \odot DFT(b))$$

Figure 6. Polynomial multiplication using the Discrete Fourier Transform. ⊙ represents an element-wise product.

We also considered a slight modification known as the Number Theoretic Transform, where , all operations use integers and are performed modulo a large prime p. In this case, rather than the complex roots of unity, we need an integer w such that $w^n \equiv 1\ (mod\ p)$ and $w^0,\ w^1,\ \ldots,\ w^{n-1}$ are distinct. Certain primes, colloquially known as "NTT primes", have the form $p = c2^k + 1$, where there exist primitive roots for all powers of two up to 2^k. A common choice is $998244353 = 119 * 2^{23} + 1$ [5]. However, this approach introduces essentially an extra modulo operation for all arithmetic operators, and trying to

optimize this with techniques such Montgomery multiplication or Barrett reduction proved to be less performant and/or far more tedious than just using complex floating-point numbers. To combat results becoming too large, since we only care about whether or not a sum is possible, after the inverse DFT we round each result to 0 or 1, making the precision issue negligible.

Naively computing the DFT takes time $O(n^2)$, leaving us no better off than just naively multiplying our polynomials in coefficient form (and no better off than just running the naive algorithm on one thread). So now we need a way to efficiently compute the DFT.

FFT

The Fast Fourier Transform is an efficient divide and conquer algorithm for computing the DFT in $O(n \log n)$ time, typically attributed to Cooley and Tukey in 1965 [6]. Beginning with our polynomial in coefficient form $a = [a_0, a_1, \ldots, a_{n-1}]$, we divide its terms into into two halves based on the parity of their index $e = [a_0, a_2, \ldots, a_{n-2}]$ and $o = [a_1, a_3, \ldots, a_{n-1}]$. The two halves in a polynomial form can be seen in Fig. 7. We can see that our original polynomial can be expressed in terms of these halves: $a(x) = e(x^2) + xo(x^2)$.

$$e(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{\frac{n}{2}-1}$$

$$o(x) = a_1 x^0 + a_3 x^1 + \ldots + a_{n-1} x^{\frac{n}{2}-1}$$

Figure 7. The two polynomial halves formed from dividing up the coefficients in *a*.

Now, suppose we recursively apply the FFT to both halves. We now have $e' = FFT(e)$ and $o' = FFT(o)$ (i.e., our two halves evaluated at the $\frac{n}{2}$th roots of unity) and must combine these results to

compute $FFT(a)$. Since the $\frac{n}{2}$th roots of unity are just every other nth root, the first $\frac{n}{2}$ terms of a' can be computed directly from e' and o': $a'(w_n^{\,k}) = e'_k + w_n^{\,k} o'_k$. The last $\frac{n}{2}$ terms can be computed as $a'(w_n^{\,k}) = e'_{k-n/2} - w_n^{\,k} o'_{k-n/2}$. Thus, from e' and o', we can compute a' in $O(n)$ time with the pattern shown in Fig. 8, known as the butterfly.

$$\text{For } k = 0 \ldots \frac{n}{2} - 1$$

$$a'_k = e'_k + w_n^{\,k} o'_k$$

$$a'_{k+\frac{n}{2}} = e'_k - w_n^{\,k} o'_k$$

Figure 8. Computing the FFT of a given the FFT of its even and odd halves.

With this FFT algorithm, we can now multiply our a and b polynomials as shown in Fig. 9.

Given polynomials a and b of length n

$$a' = \text{FFT}(a)$$

$$b' = \text{FFT}(b)$$

$$c' = a' \odot b'$$

$$c = \text{InverseFFT}(c')$$

Return c

Figure 9. Multiplying two polynomials using FFT.

We started with this standard FFT implementation, and given that it was a recursive divide and conquer algorithm, the natural approach was again to utilize OpenMP's tasks to compute the FFT of each of the original array's halves, then use OpenMP's parallel for to combine the results from the two halves. This straightforward implementation turned out to be slower than running it single-threaded. One reason is likely due to the creation of too many tasks and the task management overhead overriding any performance gain. Another is that this recursive implementation has very poor spatial locality: each recursive call allocates two new auxiliary arrays to be operated on. Combined with the large number of created tasks, the resulting memory accesses are essentially random and suboptimal for caches. We initially considered the standard technique of setting a threshold to limit the amount of tasks created, but similar to the transformation of iteratively solving the SSP on subarrays, there was a much more performant optimization.

Bit Reversal

Given that we compute the FFT on power-of-2 sized arrays, there is a very structured pattern to the order in which the butterfly transform is applied (Fig. 10).

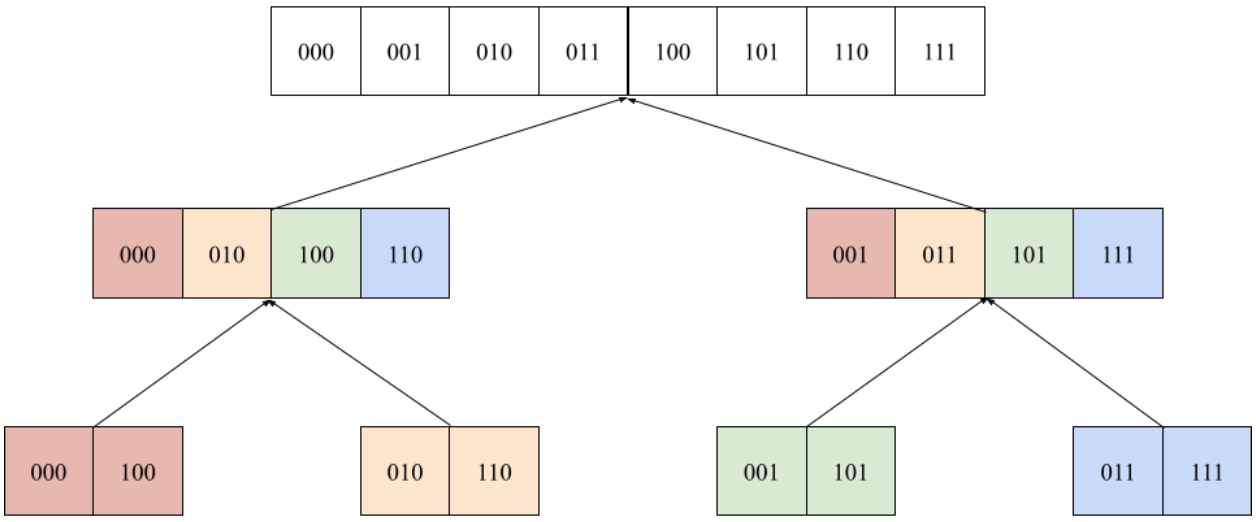

Figure 10. How the elements in the original array are split at each level of recursion during FFT. At each layer, the colors indicate pairs of elements that we apply the butterfly on.

We can notice that at the very last level, the elements are reordered based on their indices after reversing the bits. We then apply the butterfly transform on subarrays of length 2, then subarrays of length 4, and get the final result. This means that instead of recursion, we can first reorder the elements based on the bit reversed indices, then compute the FFT iteratively. It turns out that given some function $reversebits$ that reverses the bits in its input, the bit reversal ordering can be computed by swapping elements a_i and $a_{reversebits(i)}$ iff $i < reversebits(i)$ [7]. Then, for each power-of-2 m between 2 and n, we consider all the disjoint subarrays of length m. We have already computed the FFT of the first and second halves separately. Now we simply apply the same butterfly transform as in the recursive algorithm over the entire

length m subarray to get the FFT of the subarray. This subarray will then be used as one of the halves for another subarray, which is equivalent to returning the computed FFT in the recursive algorithm.

Not only does this algorithm avoid the need for recursive calls, it is entirely in-place. We avoid the overhead of managing tasks and the time it takes to allocate all of the auxiliary arrays and the poor spatial locality of the typical recursive implementation. Since the different "blocks" within a layer are independent, they can be computed in parallel. So within a layer, we used OpenMP's parallel for to compute the butterfly transform on several blocks on multiple cores simultaneously. We opted to use static scheduling since the amount of work done is essentially identical on all iterations, meaning we can avoid any unnecessary overhead from task management.

SIMD

Another opportunity for parallelism lies in the butterfly transform. We can notice from the butterfly transform formula in Fig. 8 that we are performing the exact same operation with the same inputs in each iteration, just with a different index. This is the exact setup for the data parallel parallelism model. The basic idea can be seen in Fig. 11. A slight modification we have to make is that rather than computing the nth roots of unity on the fly in the standard implementation (e.g., keeping a cumulative product and multiplying it by the principal root on every iteration), we must compute several roots ahead of time and store them in an array to be loaded later.

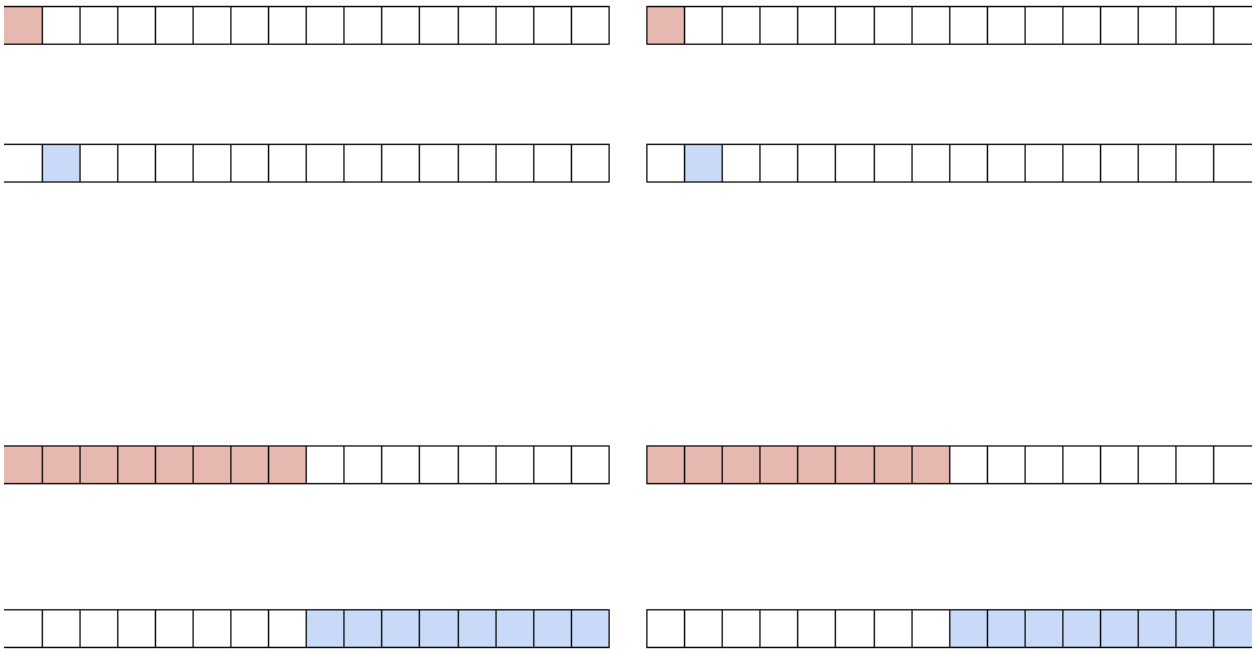

Figure 11. The elements butterfly transform has been applied to during two iterations. The top two depict the standard scalar computation while the latter two are using 8-wide vectors.

Our first approach to SIMD was using ISPC's foreach construct, allowing us to use our scalar code with almost no modifications and gain the speedup from the vector instructions that ISPC compiles to. Originally, our implementation used 64-bit doubles, but since we are not too concerned about precision (due to rounding results to 0 or 1 at the end), we opted to instead use 32-bit floats. Since our target machine is Bridges-2, whose CPU supports AVX's 8-wide vector units, we could load and operate on 8 floats at a time. Unfortunately, since complex numbers consist of 2 floats each, this effectively meant we could only operate on 4 complex numbers at a time. Additionally, implementing complex number

multiplication involves multiplying the different components of the complex numbers together, as seen in Fig. 11.

$$(a + bi)(c + di) = (ac - bd) + (ac + bd)i$$

Figure 12. Complex number multiplication

Since standard complex number implementations involve storing the two floats in adjacent addresses, if we wanted to implement the complex number multiplication (since ISPC does not support std::complex) loading the complex numbers from memory directly into vectors, multiplication would require using gather and scatter operations to collect the different components of the complex numbers into a single vector and store the results back into memory. To avoid this issue, we split each complex number into its real and imaginary parts. This way, all complex number operations will only ever have to load and store contiguous addresses in memory, and each CPU can perform 8 butterfly transforms at a time. We added this optimization for both the butterfly transforms during the FFT itself and during the element-wise multiplication in the second step of polynomial multiplication after computing the FFT of the two input arrays.

However, we were unable to get ISPC working on Bridges-2, so we decided to directly implement the SIMD instructions using Intel's AVX2 intrinsics for C++. So during layers with blocks of length 16 and above, we used the __m256 datatype to store 8 floats at a time, allowing us to compute the butterfly for 16 elements at a time (8 from each half). For the layer with blocks of length 8, we use the same operations but with __m128 to compute the butterfly transform for the entire block at once. Layers with blocks of length 2 and 4 are difficult to optimize in this fashion, since the vector extends past the block, and computing the butterfly transform on multiple blocks at once introduces scatter and gather operations because of how the terms are combined. For this reason, we opted to not optimize these two layers with SIMD and instead reverted to our previous scalar code. Similar to the initial ISPC implementation, we

added SIMD to the element-wise vector product, which was a straightforward complex number multiplication, allowing each CPU to compute the product of 8 elements at a time.

Further Optimizations

The next observation is that many computations are repeated during the overall algorithm. For instance, every time we wish to compute the FFT of some polynomial, we must first swap elements according to the bit reversal order. This means that we compute the bit reversal of all integers from 0 to $n - 1$ every time we compute the FFT. We initially implemented the naive approach, which simply iterated through the bits from 0 to $log_2 n$ and if the bit was set, we set the bit in the reversed position in the output integer. We first optimized this by reversing 8 bits at a time after storing the reversal of all 256 8-bit patterns in a constexpr array and looping over entire bytes at a time. This cut the number of iterations by a factor of 8. Since the result of reversing the bits of an integer will always be the same, we first compute all the bit reversal results we will need before starting the algorithm and look them up when needed. Additionally, since the size of our polynomials are constant (we truncate the unnecessary terms after the multiplication), the indices that need to be swapped during the reordering process is also constant. This means that after precomputing the bit reversal results, we store all the pairs of indices that must be swapped ahead of time. When we reorder the elements in FFT, we simply loop over this array of pairs that must be swapped, rather than computing the bit reversal and checking whether it is smaller than the original index.

An added advantage of always computing the FFT of the same size polynomial every time is that we know the roots of unity we will end up needing. In addition to precomputing the bit reversal swaps, we also compute all nth roots of unity in parallel using OpenMP, since the computation is independent of each other and all the precomputation takes place before starting the actual SSP algorithm. Since we will only need power-of-2 roots of unity, all the roots we will need are contained within these n roots. For instance, at the very top level, we will use the n roots. At the second level we use every 2nd root, at the

third level we use every 4th root, and so on. This is because $\frac{2\pi ik}{n/2} = \frac{2\pi i(2k)}{n}$. However, only precomputing the nth roots of unity is inefficient for the SIMD butterfly code, because at lower levels we will have to gather values that aren't contiguous in memory (e.g., at the second level we will need to use every other root). To alleviate this, for every root of unity for smaller powers of 2, we simply copy the corresponding value from the original nth roots into a new array. This means that during the butterfly transform step, we simply use the array of roots corresponding to the level we are currently at, and we only perform contiguous loads and stores from/to memory.

GPU

For the CUDA implementation we targeted the GHC machine GPUs: NVIDIA GeForce RTX 2080 B GPUs. These GPUs have 46 Streaming Multiprocessors, 7982 MB global memory available (about 8 GB), and a CUDA cap of 7.5 which means that the GPU supports features of CUDA version 7.5. There is one of these GPUs in each GHC machine.

The initial idea was to take the original naive dynamic programming algorithm and provide some basic parallelization along the inner loop where the values are updated within the dynamic programming array based on the current weight/set element being evaluated.

```
C/C++
for (uint32_t i = l; i < r; i++) {
    const uint32_t x = w[i];
    for (uint32_t j = T; j >= x; j--) {
        dp[j] = dp[j] || dp[j - x];
    }
}
```

However, the original algorithm does not map well to the CUDA structure. In order to properly utilize the blocks and to parallelize the update of the dp array, we utilized two arrays (dp_previous and dp_current). Instead of computing the entire dp matrix, we only keep two rows in memory to reduce memory usage and the global memory that needs to be used by CUDA. These two arrays allow us to update all the elements of a row at the same time since there aren't any dependencies on other elements within the same row. The simple CUDA implementation generates at least T threads (where T is the target sum) and each thread will update its corresponding value in the dp row. We still loop through all the weights and swap the current and previous row after each weight computation, synchronizing the device to make sure that the values have been properly updated.

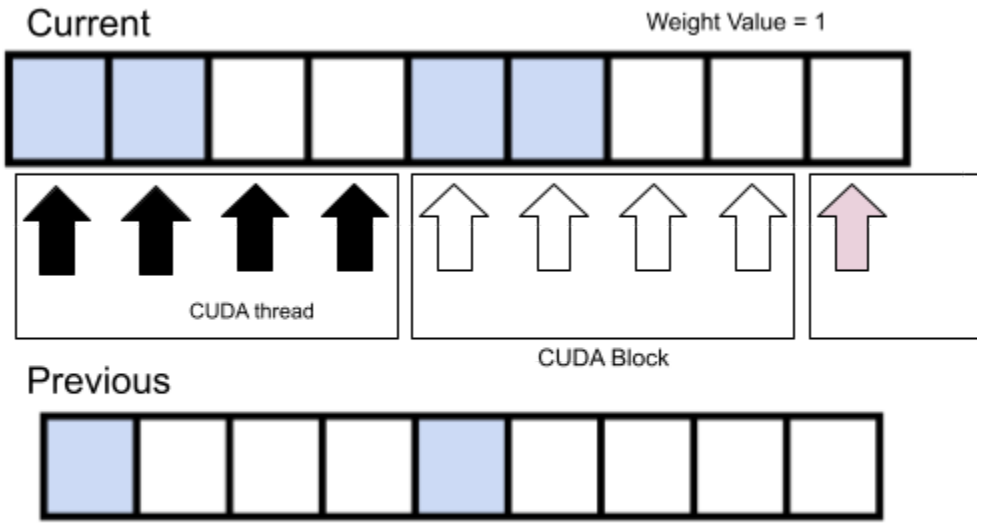

Figure 13. Describes the distribution of work in the simple CUDA implementation.

From this, we followed a similar path to the parallel CPU algorithm for the CUDA implementation based on FFT, dividing the set into different disjoint subsets and computing the dynamic programming algorithm at these smaller subarrays as shown in Figure 1.

To further parallelize the dynamic programming aspect, we incorporated components from the simple CUDA implementation. Each block would compute the dynamic programming array for one subarray and each thread in the block would be responsible for its own segment of the dynamic programming array to update. This allowed us to eliminate part of the inner loop from the naive sequential implementation as well as parallelize the subset computations. To make sure that the updates were synchronized across a block, we utilized __syncthreads() at the beginning and end of each iteration on the weights in the block which is likely a bottleneck in the parallelization process as it causes all the threads in a block to wait before continuing on each value in the set. Even with this synchronization, the code is still able to parallelize really well across blocks which means that the hyperparameters chosen can have a really big effect on the results depending on the test case.

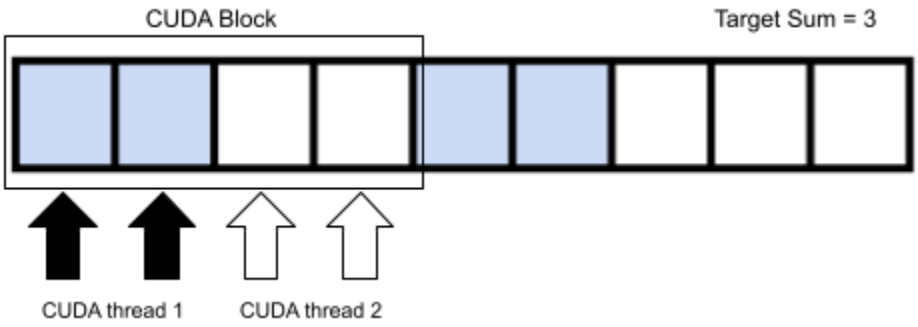

Figure 14. Describes how each CUDA thread is in charge of updating a specific section of the dp row for each subset within a global DP array.

The issue with CUDA implementations is the mapping of data to the different threads and blocks. Since the dynamic programming array is extremely large, it needs to be placed on global memory rather than shared memory which decreases the performance overall. Thus, we needed to compute a large global dynamic programming array that contained both the current and previous dp rows for each block (each subarray). Like the previous simple CUDA implementation, we kept the 2 row dp implementation for a better memory utilization and instead of swapping the rows (which is not possible in CUDA), we had to switch the current and previous index to keep track of which array to update at each step.

We then mapped the final “current” rows from each block into a separate array to properly track the values using the same intervals previously defined to replace the inner loop of the sequential algorithm. From there, we followed a similar approach to the parallel CPU implementation by utilizing FFT to merge the pairs of subset dynamic programming results together.

As discussed earlier, by performing a convolution on the two polynomials generated from the two pre-calculated dp subarrays, we can combine them together until we reach one array. To do this, we utilized the cuFFT library [8]. However, with this library, that meant we needed to convert the original integers to a complex format and that it was difficult to parallelize the merging process outside of the FFT component. Despite this, we still utilized CUDA to perform the pointwise multiplication step to try to get as much parallelism as possible. The pointwise multiplication step was done by generating a fixed value of blocks and threads where each thread would update one value in the dp row with the correct complex multiplication. One big challenge in this part was the constant memory operations moving data between the GPU and CPU with each FFT call. To keep it consistent, we computed the merging on the complex values and extracted the result from this array at the end.

Further Optimizations

We considered a bit-wise computation where instead of integer arrays we considered bit shifting and setting bits within an array to provide a more memory efficient implementation of the dp algorithm. We’ve implemented a sequential version of the algorithm with this characteristic to test if it provides a performance benefit outside of the memory optimization. Since our test cases are so large, it is necessary to utilize an array of ints rather than bitset which creates logic that requires a lot more overhead. This overhead comes from needing to access each int and then setting or extracting the specific bit value within the same dynamic programming loop. How this worked is that we took the original dynamic programming array and divided it by 64 such that there is now size

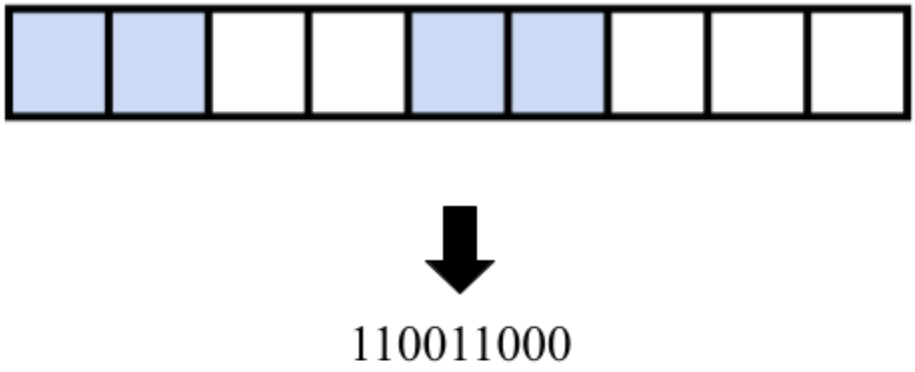

Figure 15. Describes the transform from array of ints to bits within an uint64_t.

Some future optimizations upon our CUDA implementation could be to utilize this bit-wise representation of the dynamic programming array. This would allow us to use less global memory and reduce the size of memory computations done. Additionally, some of the mappings could possibly be done in shared memory which would increase the speed in which memory accesses could be completed especially for the parallelized dynamic programming section. Lastly, during the FFT and merge section there would be less memory needing to be moved around from the GPU and CPU which could also improve performance.

Results

Since the runtime of SSP algorithms depends on the number of input elements and the value of the target sum, we tested on three different cases that varied these parameters: 10^4 integers with a 10^6 target, 10^5 integers with a 10^5 target, and 10^6 integers with a 10^4 target. For all of these cases, our primary metrics were wall clock time and speedup compared to the standard $O(NT)$ dynamic programming algorithm running on a single core as described in Background.

CPU

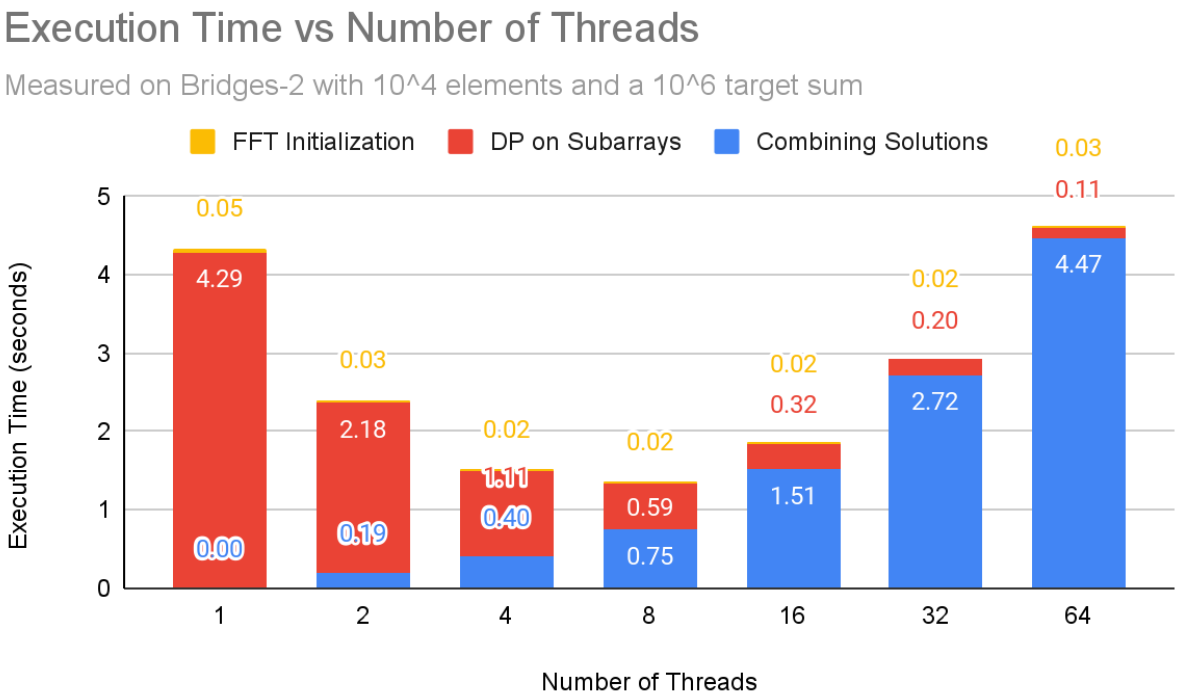

Figure 16. Execution time of the three components in the parallel CPU algorithm on 10^4 elements and a target sum of 10^6

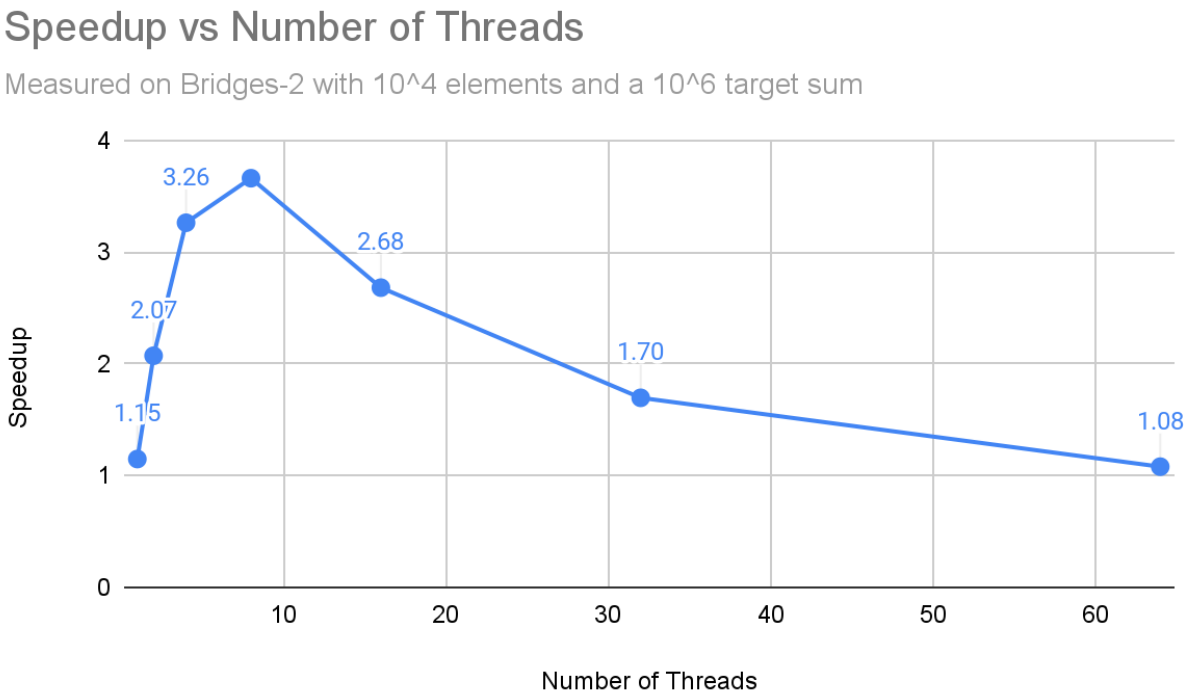

Figure 17. Overall speedup for the parallel CPU algorithm on 10^4 elements and a target sum of 10^6

The first test consisted of a set of 10^4 integers and a target sum of 10^6. We can see from the execution time graph that the second step of computing the set of possible sums in each thread's subarray achieves near-linear speedup. This is as expected, since the original input set is divided roughly evenly among all the threads and all threads are running on their own core at the same time. However, the final step of repeatedly combining solutions also scales near-linear with the number of threads. At a large number of threads, this step completely dominates the running time, at which point the algorithm scales extremely poorly. We can see this effect in the speedup graph, where the speedup is very good for 1, 2, and 4, threads until it peaks at 3.66x on 8 threads and begins to fall.

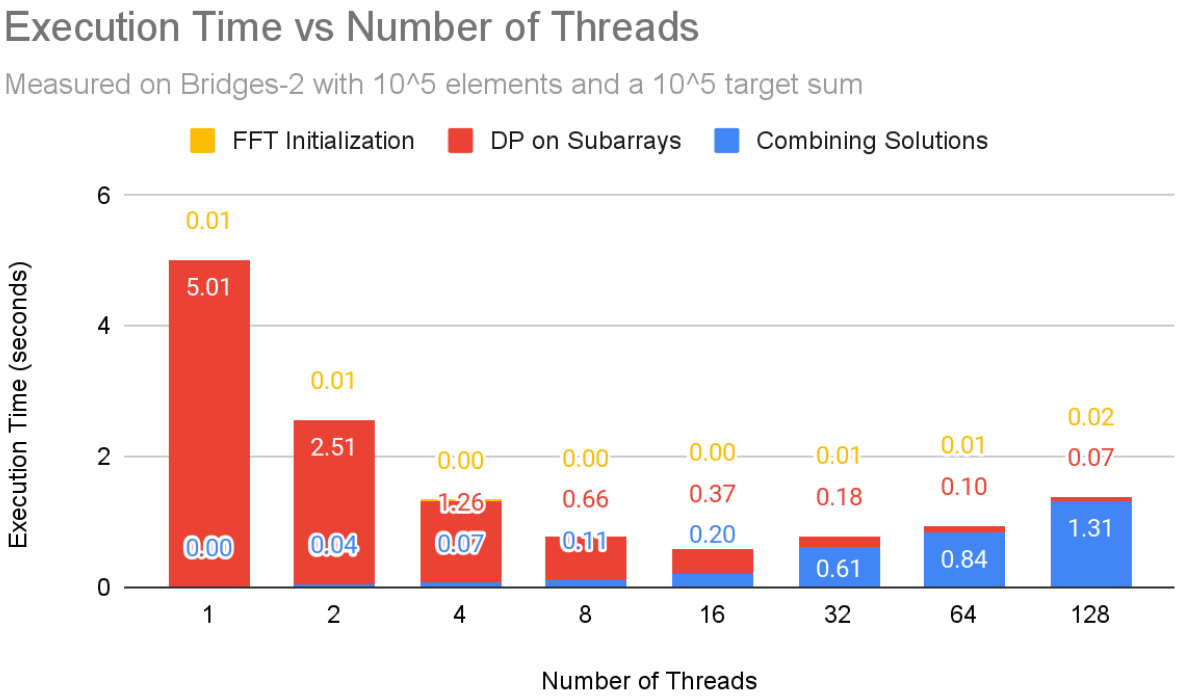

Figure 18. Execution time of the three components in the parallel CPU algorithm on 10^5 elements and a target sum of 10^5

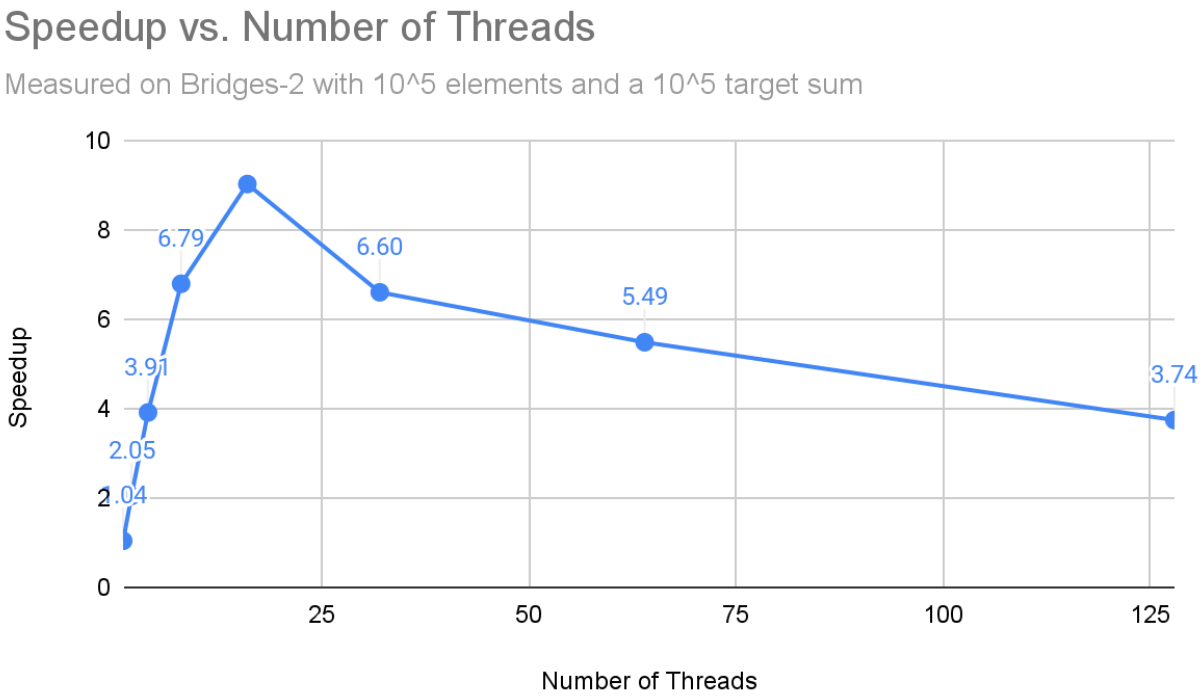

Figure 19. Overall speedup for the parallel CPU algorithm on 10^5 elements and a target sum of 10^5

Similar to the previous test, the execution of the second step achieves near-linear speedup, while the combining solutions step slowly dominates the runtime as the number of threads increases. Again, we can see that the speedup is good when running on 1, 2, 4, and 8 threads, before peaking at 9.03x at 16 threads and starting to fall.

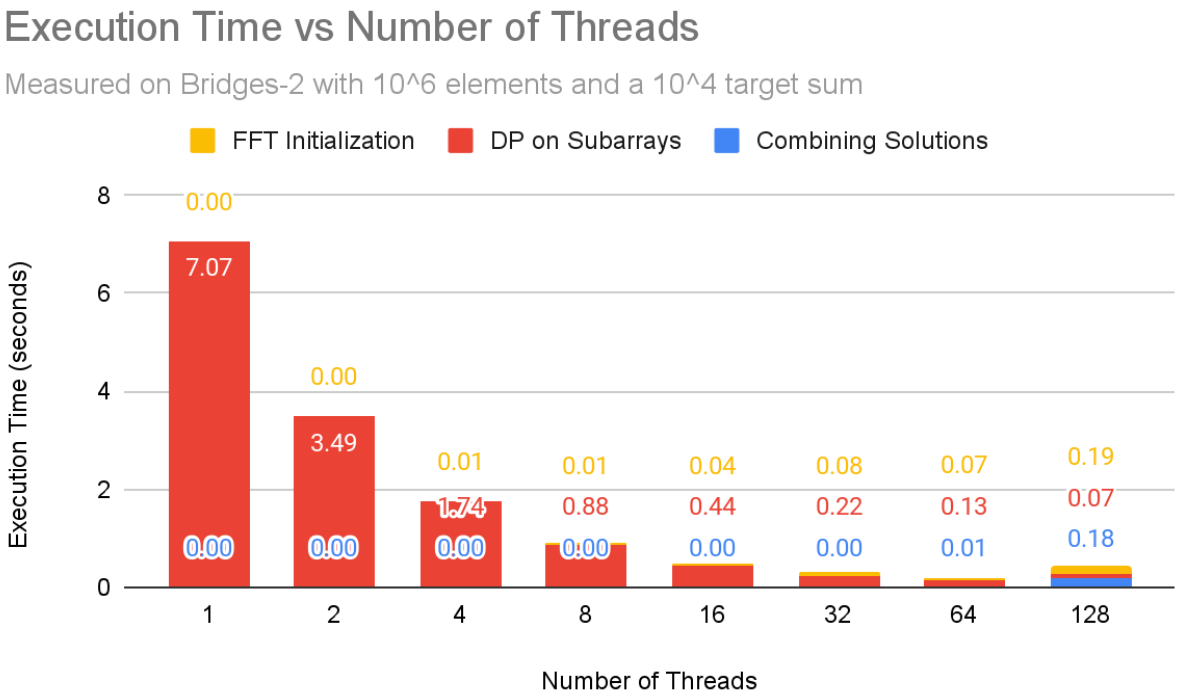

Figure 20. Execution time of the three components in the parallel CPU algorithm on 10^6 elements and a target sum of 10^4

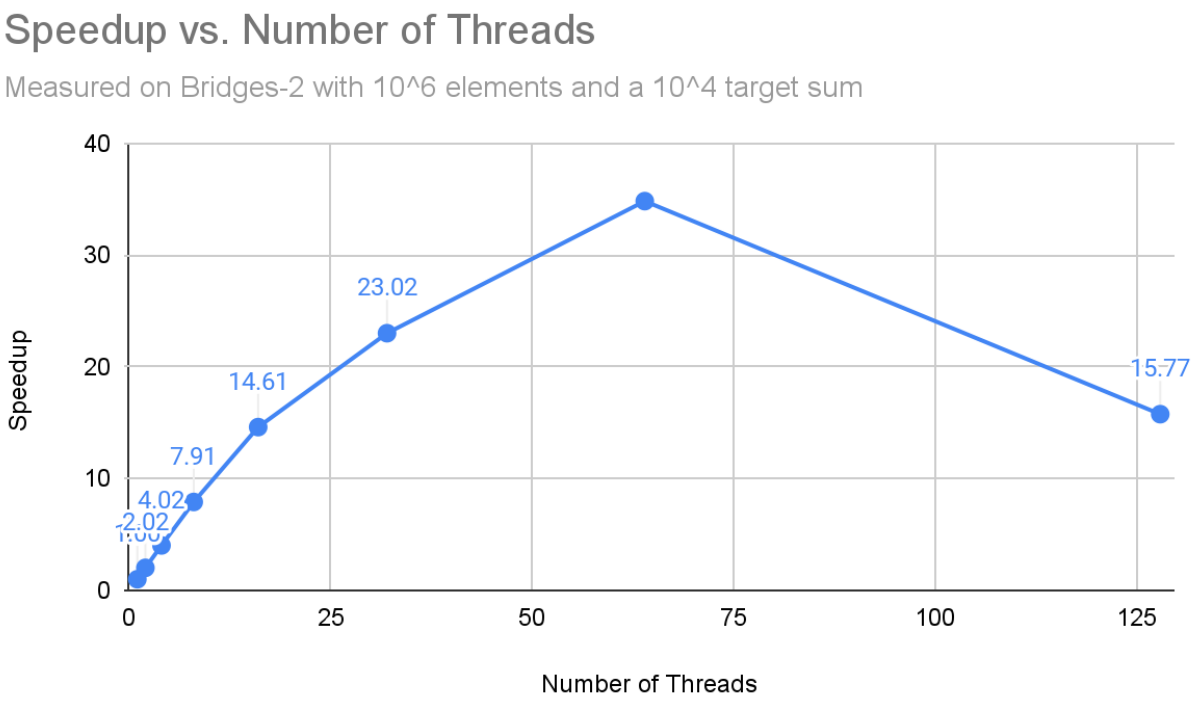

Figure 21. Overall speedup for the parallel CPU algorithm on 10^6 elements and a target sum of 10^4

Unlike the previous two tests, the time spent on combining solutions made up a very small proportion of the total running time, except on 128 threads. This meant that the time spent running the standard dp algorithm made up most of the execution time, and as seen before, this part achieves near linear speedup. This can be seen in the speedup graph, which scales near-linearly up to 32 threads, where it achieves a 23.02x speedup and peaks at a 34.84x speedup on 64 threads.

Analysis

In general, it appears that the runtime and speedup are determined by how large the target sum is. Intuitively this makes sense based on the algorithm. Regardless of how many integers are in the input set, we always divide them evenly among all the threads, resulting in this stage having near linear speedup since the threads can run simultaneously on all cores. This means that in the third stage, the number of solutions to combine is exactly equal to the number of threads. During this stage the runtime depends only

on the number of solutions to combine and the target sum. And since the number of solutions to combine is simply the number of threads, the runtime is effectively determined by the target sum. With a small target sum (the third case), this stage took very little time except for 128 threads, resulting in a large speedup. With a large target sum (the first case), this stage took an extremely long time, resulting in very poor speedup.

Additionally, the algorithm still suffers from a lack of parallelism, as there are times when not all cores will be utilized. Both the combining solutions step and FFT work bottom-up and are dependent on solutions in the next lower layer to be computed first. When they get to the uppermost layer, there are only two objects in the next lower layer, and only one thread will work on combining them. The algorithm also potentially suffers from over parallelism at times, since at lower levels in the combining solutions step, there are many concurrent FFT calls. The FFT procedure itself utilizes OpenMP's parallel for, creating more threads. This increases the amount of overhead in managing the threads, reducing speedup. It may be possible to do a better job of controlling the amount of parallelism so there aren't too many threads running at a time, for instance by running lower levels of the FFT sequentially.

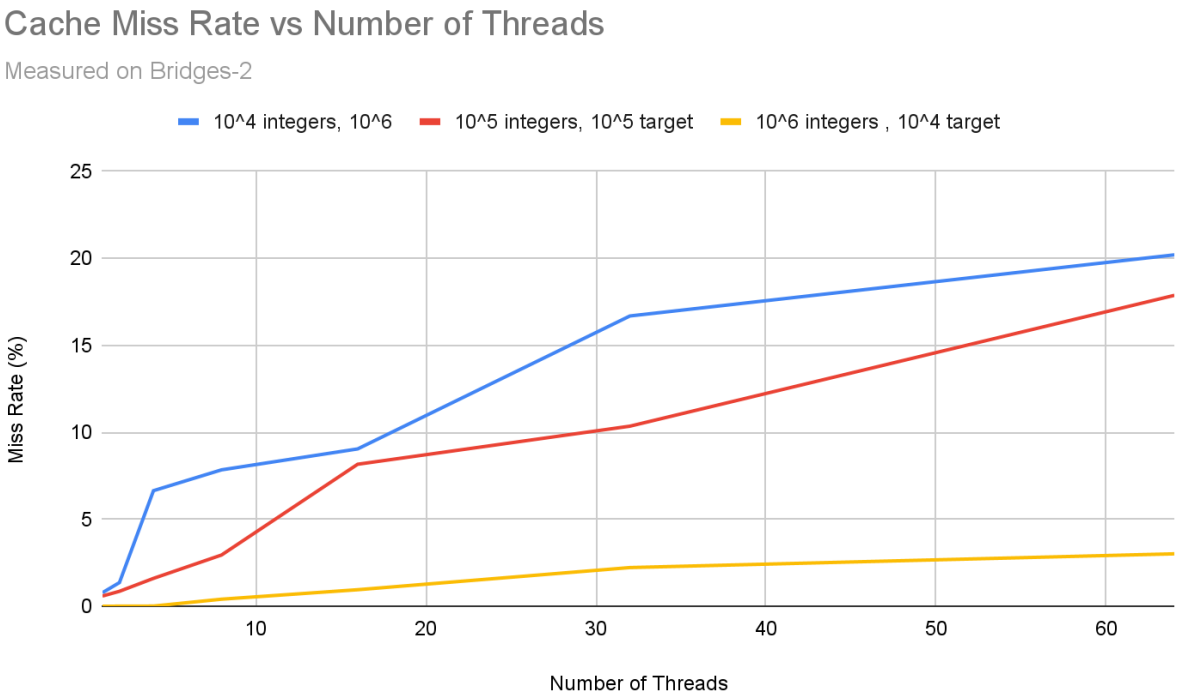

Figure 22. Cache miss rate versus number of threads for the three test cases

Another big issue is the algorithm being very memory bound. We can see from Fig. 22 that a larger target sum results in a higher cache miss rate. This is likely due to the over parallelism as mentioned earlier. With so many simultaneous FFT calls, each working on different arrays, after a context switch it is likely that most of the thread's array is no longer in the cache. This may also be due to the heavy use of precomputation, which essentially trades off computation for an additional memory access. However, in a program that is already memory bound, this may not be a good choice to make. Additional testing is required to determine if it would be better to perform the computation of bit reversals and roots of unity on the fly to reduce bandwidth requirements and improve spatial locality.

The CUDA implementations have the following speedups versus the naive sequential implementation on each test case with a fixed set of hyperparameters, in this case blocks with 256 threads and ((number of elements)/4096) number of blocks (where 4096 is the "block_factor"). The test cases are named by the number of elements in the set _ the target sum value.

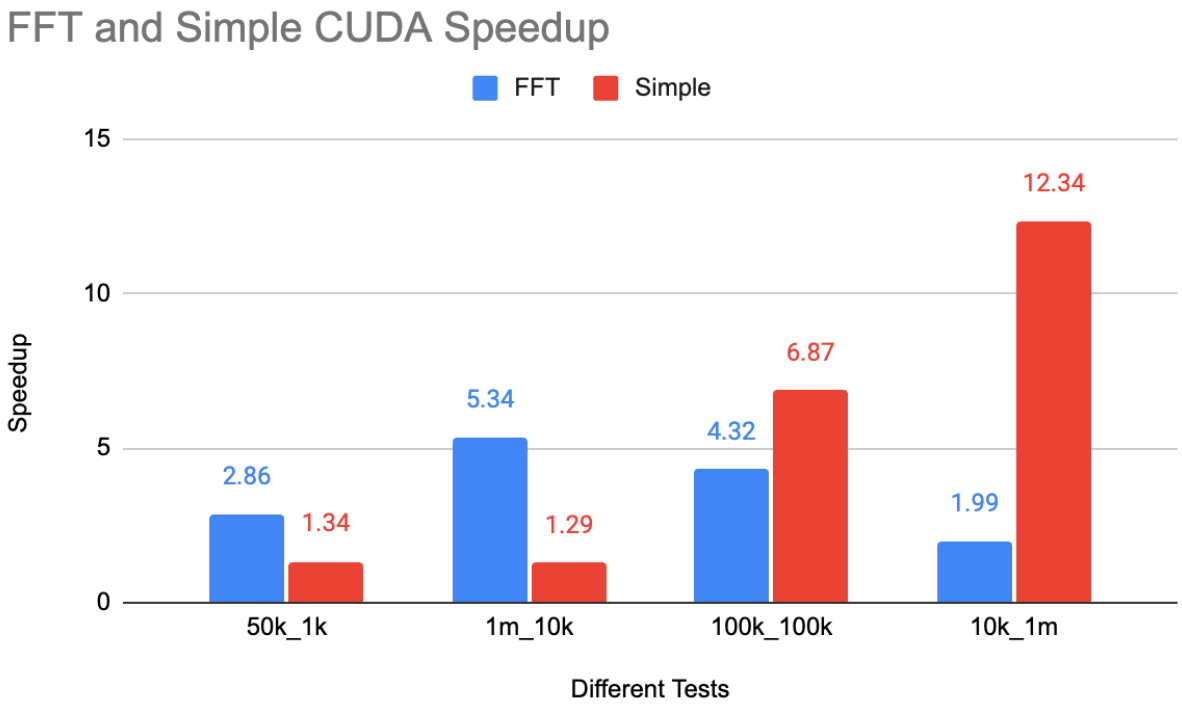

Figure 23. Speedups for the FFT and Simple CUDA implementations.

The reason why the speedup dramatically decreases for the FFT implementation in the 100k and 1m implementations (for the target sum) is because of the forced sequential nature of some of the transforms we implemented in the middle of the function to turn the values into complex values and reshape the array into a 2d array. If we eliminated the 2d aspect of the array function, rendering it difficult to use/understand and the indexing logic to be quite confusing, we would be able to exploit the parallelism better. This is a reason why the mapping to CUDA is difficult because the parallelizing of this component is much simpler with a CPU implementation and ability to directly parallelize on the for loop to assign values into the 2d matrix. We expect that the speedup should be equivalent or outperform the simple implementation once

this optimization is applied.

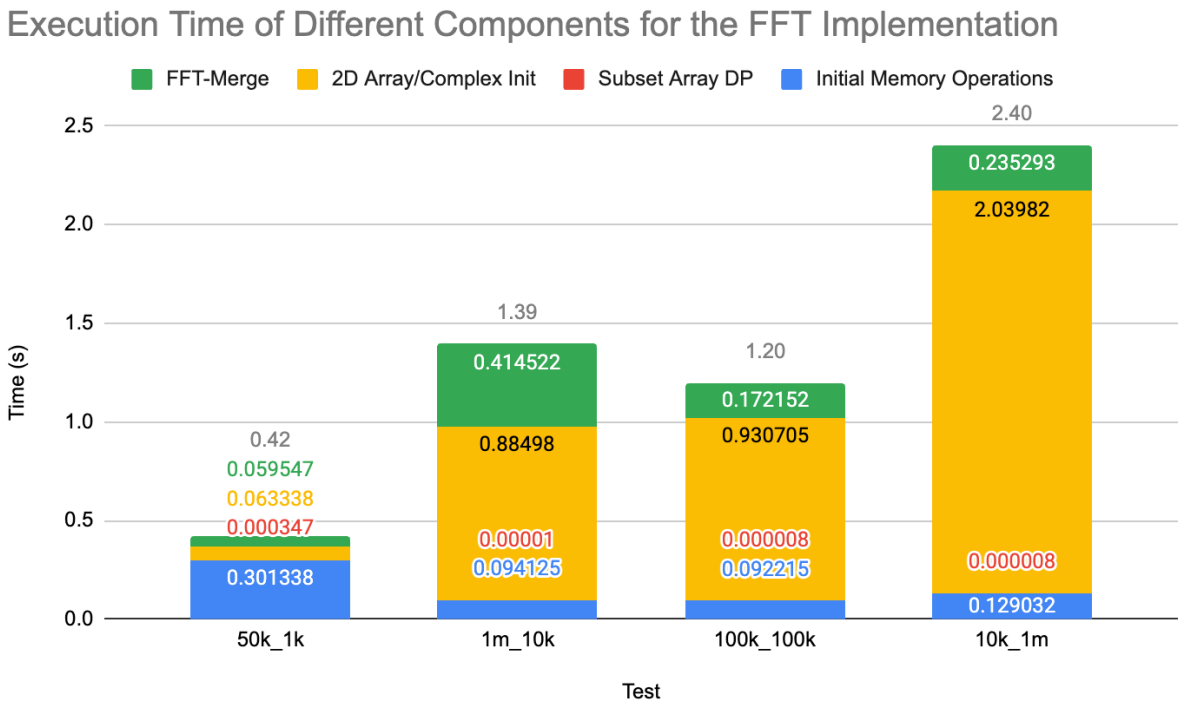

Figure 24. Execution Time on different components of the FFT Implementation on various tests.

In Figure 24 above, we see that the bottleneck on the FFT implementation especially for the scenes with larger target sums is at the 2d/complex initialization step since it is written as sequential in our implementation. In the future, we can parallelize this using a CUDA kernel function and complex indexing to complete the FFT-Merge step. Out of all the steps, the subset array DP step is the quickest, demonstrating the effectiveness of the parallelization both into subsets and within the kernel function to assign new values to the dp rows.

Additionally, we see that the initial memory operations take up a significant portion of time, demonstrating how our CUDA implementation is also limited by the global memory operations and allocation that needs to occur.

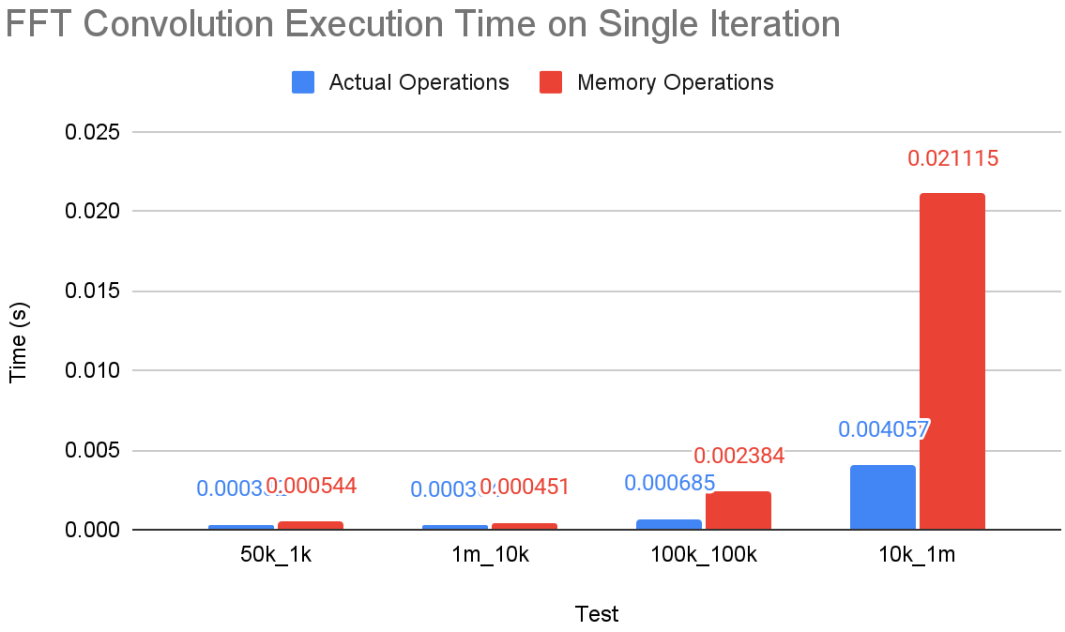

Figure 25. Memory Operations execution time vs FFT/multiplication on a single iteration of the FFT convolution.

We also performed an analysis of the FFT component in Figure 25, timing the memory operations vs actual FFT/multiplication that occurred. We found that for most scenes, more time was spent on the memory operations than the actual calculations which further demonstrates the memory limitations and the bottleneck caused by needing to move the data from the GPU to CPU and vice versa.

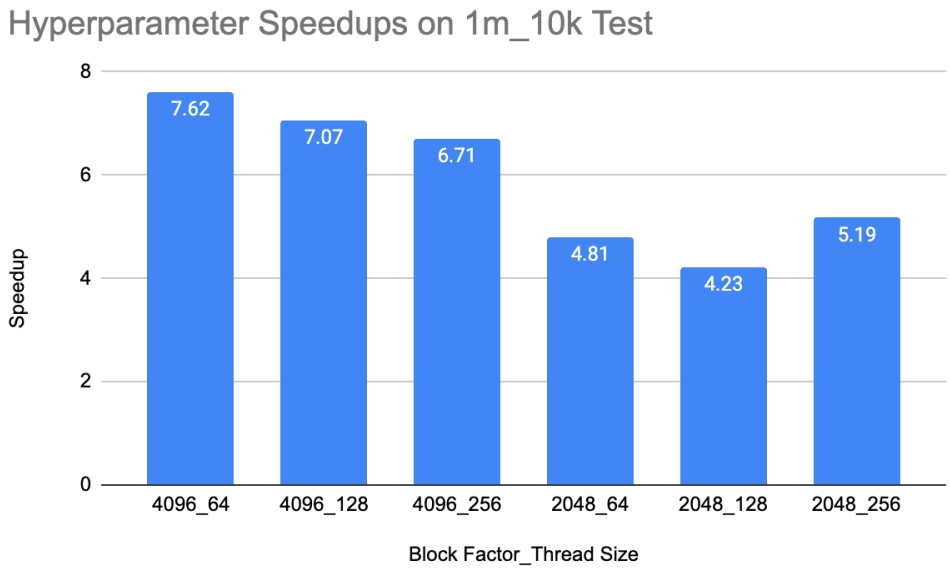

Figure 26. Hyperparameter speedups on a specific test given distinct block factors and thread sizes.

We found that the hyperparameters used also had a large impact on the speedup for the FFT implementation. In Figure 26, we see that both the block factor and the number of threads within each block had an effect on the speedup. This is because these factors influence both the FFT-Merge aspect and the Subset DP aspect of the implementation. In the Subset DP aspect, the block factor and thread size define how finely the arrays are split into subarrays and within the CUDA kernel function, how much each thread needs to update sequentially. In the FFT-Merge aspect this determines the number of iterations needed which can limit the speedup significantly. We also found that this varies depending on the test which follows due to the number of values and the size of the dp array requiring more of less work.

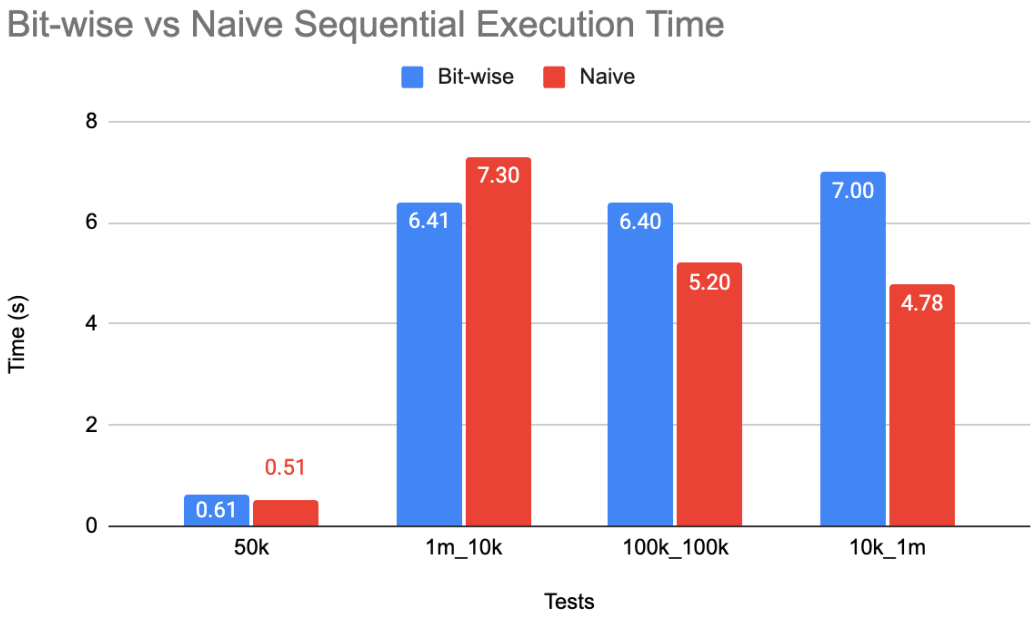

Figure 27. Basic analysis of execution time for the two sequential implementations.

We also performed a basic analysis in Figure 27 above of the sequential bit-wise implementation vs. the original sequential implementation and found that the bit-wise implementation tended to perform worse, likely due to the overhead caused by the additional checking and setting of bits.

Overall, we think that the CPU is a better choice for parallelization just due to the complexity of memory assignment and bottleneck of memory operations that is required in GPU implementations.

List of Work

Devin (55%): OpenMP implementation, Naive Sequential implementation, Paper

Laura (45%): CUDA implementations, Bit-wise Sequential Implementation, Paper

References

[1] Kleinberg, J., & Tardos, E. (2006). *Algorithm Design*. Pearson.

[2] Horowitz, E., & Sahni S. (1974). Computing Partitions with Applications to the Knapsack Problem. *Journal of the ACM, 21*(2), 277-292. https://doi.org/10.1145%2F321812.321823

[3] Pisinger, D. (1996). Linear Time Algorithms for Knapsack Problems with Bounded Weights. *Journal of Algorithms, 33*(1), 1-14. https://doi.org/10.1006%2Fjagm.1999.1034

[4] Reducible. (2020, November 14). *The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?* [Video]. YouTube. https://www.youtube.com/watch?v=h7apO7q16V0

[5] Uustalu, T. (2023, January 13). *[Tutorial] FFT*. Codeforces. https://codeforces.com/blog/entry/111371

[6] An algorithm for the machine calculation of complex Fourier series https://doi.org/10.1090%2FS0025-5718-1965-0178586-1

[7] *Fast Fourier transform*. (n.d.). Algorithms for Competitive Programming. Retrieved December 14, 2023 from https://cp-algorithms.com/algebra/fft.html

[8] *cuFFT API Reference*. cuFFT 12.3 documentation. (2023, November 14). https://docs.nvidia.com/cuda/cufft/index.html