Introduction

For my project, I decided to benchmark multiple parallelizable algorithms, each of which involves a different parallelism pattern, via TBB and handwritten CUDA kernels. Ultimately I decided to explore the following algorithms, based on the benchmarking strategy of a paper I presented early in the semester:

1. Collatz conjecture: divides a number by 2 if it is even and multiplies it by 3 and increments if it is odd, repeating the process until it converges to 1.
2. 2MM: performs the matrix operation $\alpha ABC + \beta D$, where α and β represent scalar values and A, B, C, and D represent square matrices
3. Ising kernel: randomly updates positive and negative spins in a two-dimensional grid based on the values of the surrounding spins

For each program, I wrote a sequential version, a version that utilized TBB, and a version that made use of handwritten CUDA kernels. I initially wanted to use Thrust in conjunction with CUDA on the GPU, but decided to write my own kernel after realizing Thrust wouldn't fulfill my needs for the Ising kernel, as well as achieving more speedup using my own kernel on the Collatz conjecture as opposed to Thrust.

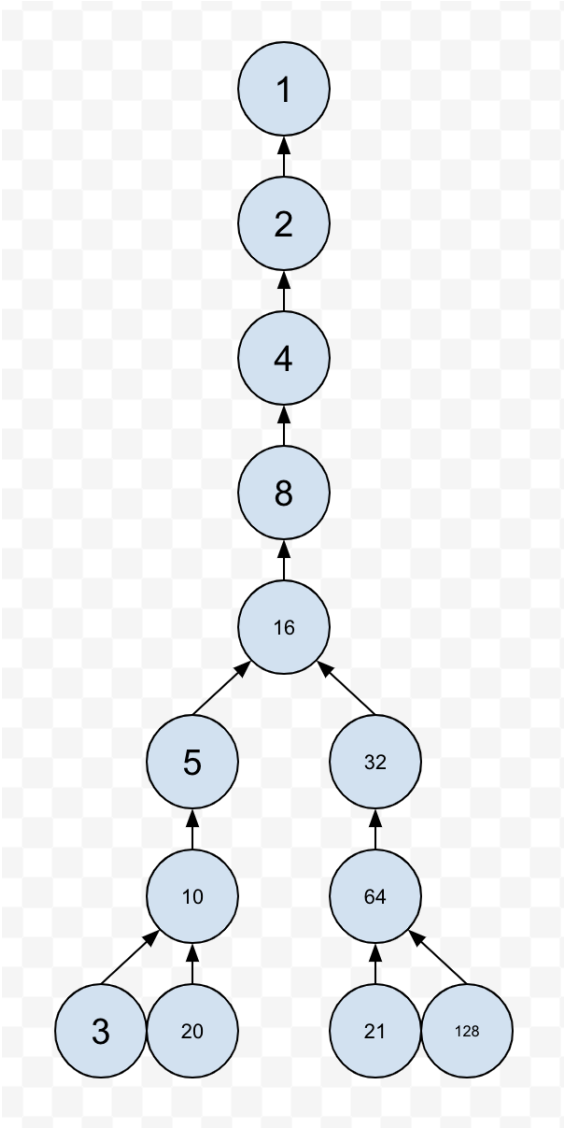
Verifying correctness of the Collatz conjecture and 2MM programs was straightforward, as mathematical operations only have one correct result, and any difference between the sequential and parallel versions is a clear indication of incorrect code. The Ising kernel, on the other hand, can only be "verified" via white box code review, as it is possible that certain operations will occur in a different order in the parallel version than the sequential version, resulting in a different final state.

For benchmarking, I tested each program with both small and large amounts of data to more closely examine the effects of overhead on performance, and see if the cost incurred was one-time or recurring. There are two versions of the sequential program that have the same code but are run on different compilers: one on G++ alongside the TBB code, and one on NVCC alongside the CUDA code. This partitioning was done in order to properly verify correctness without having to worry about TBB and CUDA interoperability. In general, the two versions have largely similar runtimes, making the two parallel programs comparable to each other. I then created two graphs for each program: one that shows all four programs, and one that only compares the parallel programs.

Collatz Conjecture

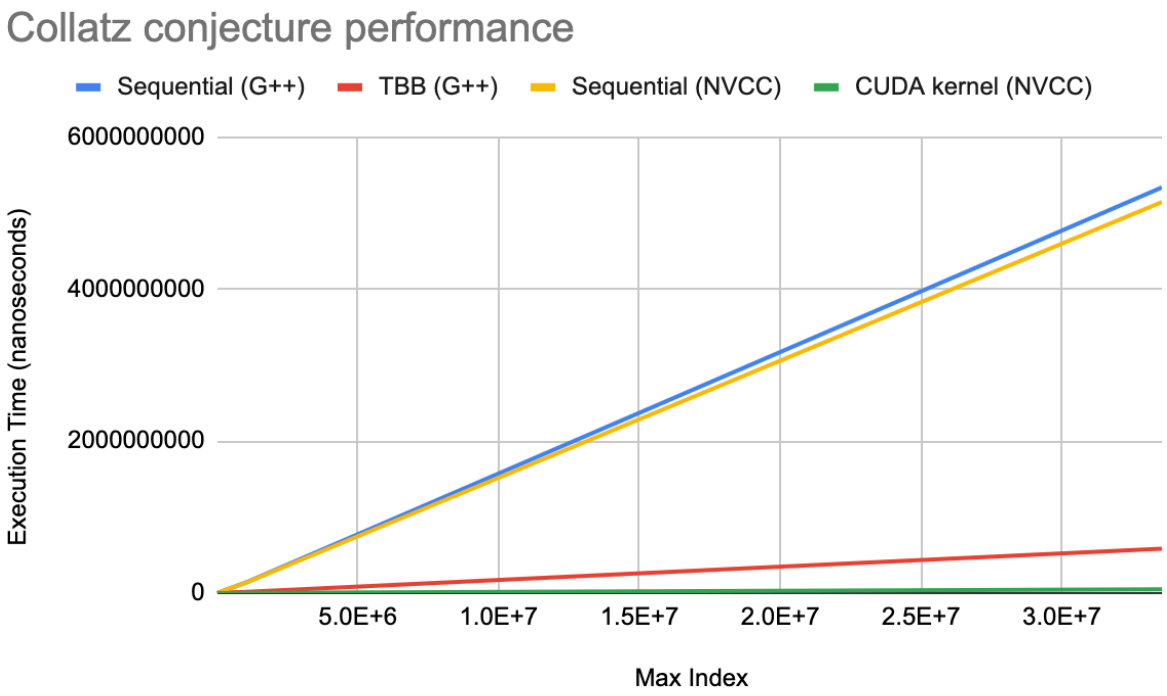
The Collatz conjecture was easy to code a sequential implementation for. Via repeated iteration of a simple while loop and if statement, the process of obtaining a vector of results ranging from 1 to the upper bound was straightforward, as was optimizing the serial algorithm by taking a dynamic programming approach that allowed later computations to rely on earlier results and skip past repeated calculations.

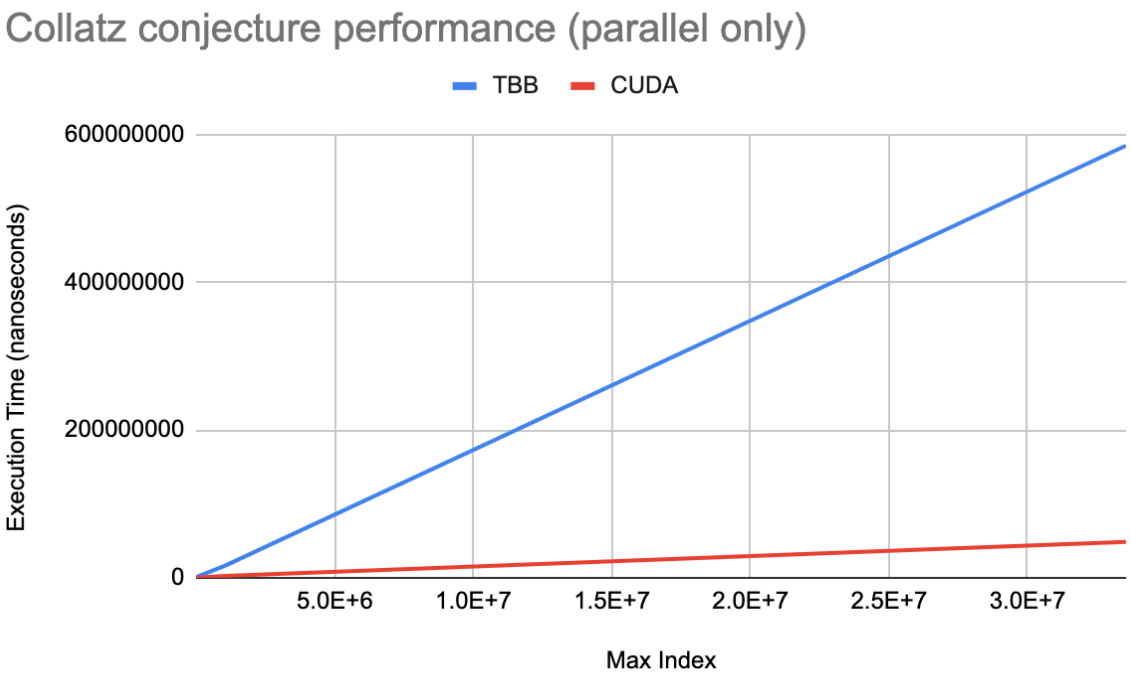
In contrast to the relative ease with which a sequential solution was implemented, I ran into issues attempting to implement dynamic programming as part of a parallel solution. I initially thought the problem would involve implementing a parallel scan, which incrementally produces outputs based on the results from previous ones. However, I quickly ran into a problem, which was that a scan needs to proceed in a clearly-defined order. For another problem like summation or Fibonacci calculations, determining the order would be easy, as outputs for both of these algorithms clearly rely on the one or two outputs immediately preceding them respectively. However, with an algorithm like the Collatz conjecture, there is no clearly-defined order available, as the number of iterations to converge can vary widely, with smaller numbers frequently converging to larger ones. For example, 3 jumps to 10, which jumps to 5, which jumps to 16. This unpredictability substantially increased the difficulty of implementing a parallel dynamic programming approach. The below figure demonstrates the difficulty of figuring out an order for a parallel scan operation.



Initially I tried to simply use a map pattern, but this didn't work, as many of the results were incorrect due to the previous results they relied on not yet being initialized. I briefly considered changing the order of the algorithm to instead compute the conjecture in the order of the number of steps it would take to reach 1 (e.g 2, 4, 8, 16, 32/5, 64/10…), but decided against this as the numbers would get large very quickly and parallelization isn't worth it for such a small number of steps. In the end I made the decision to go back to basics and individually compute each sum. I took effectively the same approach with my TBB and CUDA kernel approaches, as I ultimately treated the problem as an embarrassingly parallel map. While both programs were slower over a smaller range of numbers due to overhead, my TBB program almost managed to achieve a speedup of 10x over a higher range of numbers (2^25), while my CUDA kernel achieved a speedup of over 100x.

Max Index	Sequential (G++)	TBB (G++)	Sequential (NVCC)	CUDA kernel (NVCC)	TBB speedup	CUDA speedup
32	1188.2	512883.8	1099.6	187364.4	0.002316704096	0.005868777633
1024	69815.6	687841.2	65072.6	180668.6	0.1014995903	0.3601765885
32768	3389353.8	1320035.6	3298850.8	240613.8	2.567623025	13.710147976
1048576	140884263.6	16600759.6	136889671.6	2393972.6	8.486615492	57.1809684
33554432	5337064452	585049232.8	5145431170	48748636.4	9.122419367	105.5502584





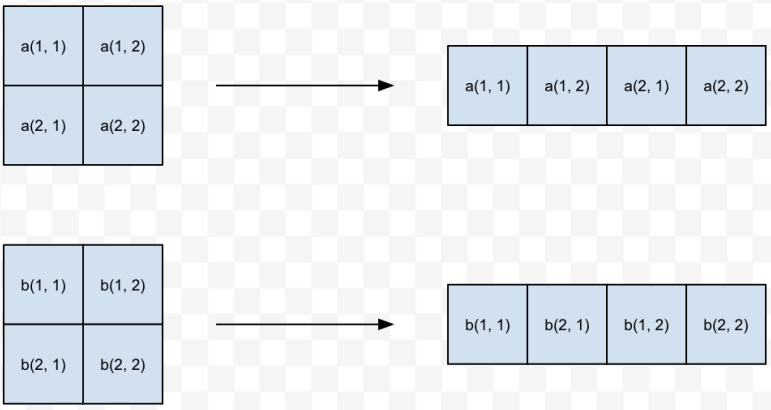
2MM

The 2MM program is ripe with opportunities when it comes to parallelization. Each scalar multiplication is an embarrassingly parallel map, as each value in the matrix being operated upon can be multiplied by the scalar value without it affecting anything else. The final matrix add

falls under similar logic, as each value in one matrix corresponds to a value to be added in the same position in the other. The two matrix-to-matrix multiplications that occur represent a more involved map-reduce operation, as each cell in the resulting product matrix is set to the dot product of its corresponding row in one matrix and its corresponding column in the other. Finally, many of these functions can be invoked in parallel, such as the βD scalar multiplication, the αA scalar multiplication, and the BC matrix multiplication.

When designing my TBB solution, I implemented all these patterns simultaneously and then experimented with a number of different combinations to see what would achieve the most speedup. Ultimately I found that implementing the map pattern worked effectively for the scalar multiplication, final matrix addition, and outer loops of the matrix multiplication (i.e computing each cell in parallel, but still doing each computation sequentially, as opposed to via a reduce). However, parallelizing the inner loop of the matrix multiplication actually slowed the program down, potentially because of having to do several column-wise vector iterations for the second matrix involved in the multiplication. Additionally, invoking the independent operations in parallel didn't appear to have any demonstrable effect on performance, possibly because the operations were computationally intensive enough to require large amounts of CPU space and not leave enough computing power for other operations to be executed simultaneously. In implementing the map pattern, I was able to achieve a speedup of just over 3x at larger matrix sizes, though interestingly enough there was more speedup achieved with 512x512 matrices (3.18x) than with 2048x2048 matrices (3.03x). This may be because the number of computations to be done with respect to matrix-to-matrix multiplication increases at a quadratic rate, making larger matrices much more demanding computationally.

When I was implementing my CUDA kernel solution, I ran into the additional hurdle of having to pass my matrices into global functions as one-dimensional arrays. This limitation didn't pose much of a challenge when it came to the relatively simple tasks of parallel scalar multiplication and matrix addition, but yet again matrix multiplication posed a challenge. I wasn't able to figure out how to correctly parallelize it, but I was able to devise a sequential solution that involves transposing the 1D representation of the second matrix being multiplied and then calculating the value in the i-th row and j-th column of the product matrix with dimension size n by taking the dot product of the i-th n-size chunk of the first matrix and the j-th n-size chunk of the transpose of the second matrix. (This explanation may be a bit dense, so I've attached a diagram to go with it)

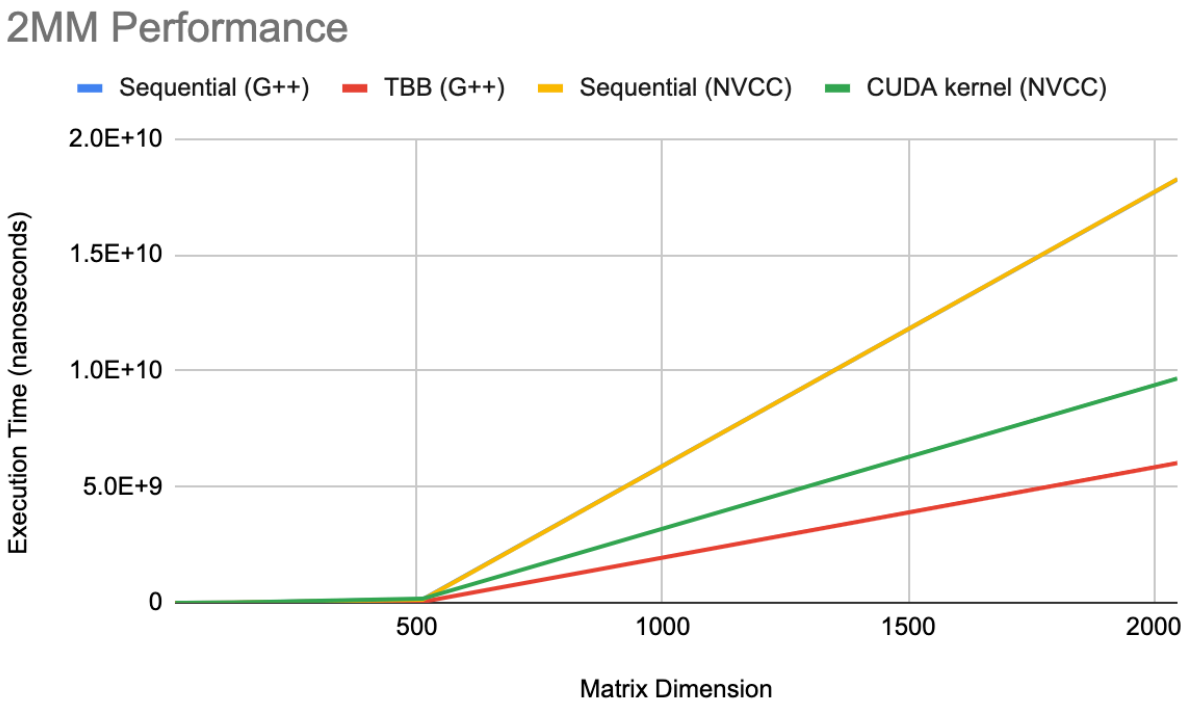


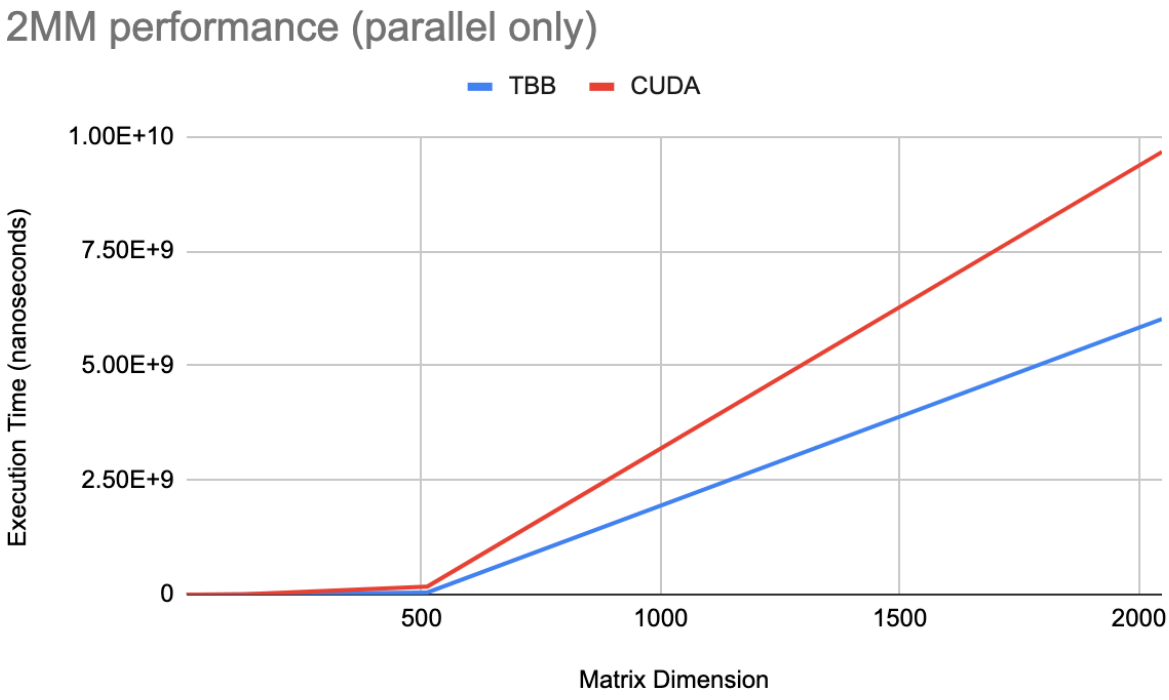
(b was transposed for the purpose of matrix multiplication)

c = a * b
dimension size = 2
c(1, 1) = 1st 2-size chunk of a * 1st 2-size chunk of b = a(1, 1) * b(1, 1) + a(1, 2) * b(2, 1)
c(1, 2) = 1st 2-size chunk of a * 2nd 2-size chunk of b = a(1, 1) * b(1, 2) + a(1, 2) * b(2, 2)
c(2, 1) = 2nd 2-size chunk of a * 1st 2-size chunk of b = a(2, 1) * b(1, 1) + a(2, 2) * b(2, 1)
c(2, 2) = 2nd 2-size chunk of a * 2nd 2-size chunk of b = a(2, 1) * b(1, 2) + a(2, 2) * b(2, 2)

Due to the matrix multiplication and prior requisite transposition algorithm running serially, my CUDA kernel speedup is relatively modest, at about 1.89x on average, although unlike with the TBB implementation, the speedup continues to improve with size.

Matrix Dimension	Sequential (G++)	TBB (G++)	Sequential (NVCC)	CUDA kernel (NVCC)	TBB speedup	CUDA speedup
8	2664.6	19268.2	874.2	319786.2	0.1382900323	0.002733701454
32	41410.8	183574.6	41909.4	477696.8	0.2255802273	0.08773221843
128	2067648	795145.4	1995981.2	5005281.6	2.60033951	0.398775006
512	136019164.6	42682651	133605842.6	171314434.6	3.186755307	0.7798866623
2048	18258675169	6018662665	18268037527	9668476962	3.033676447	1.889443146



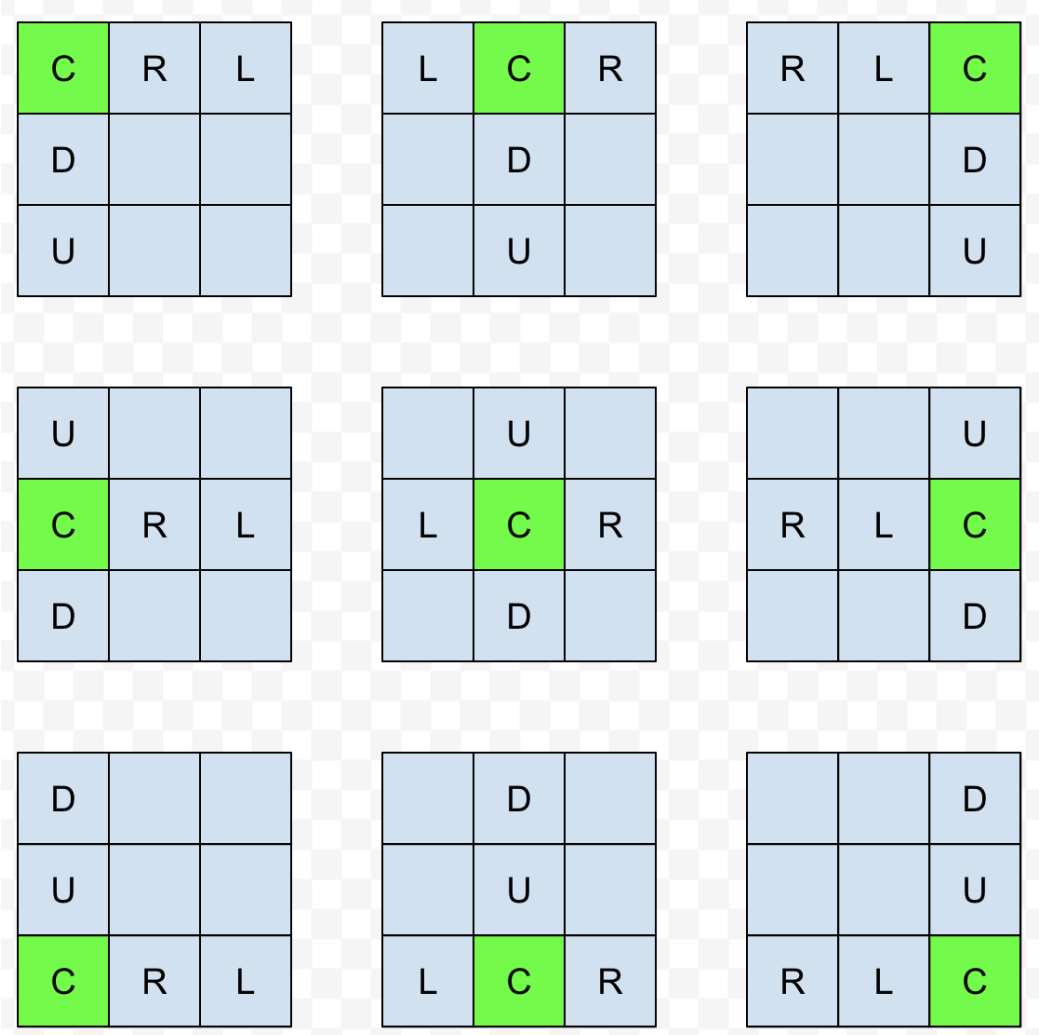


Ising Kernel

The Ising kernel involved a stencil pattern from the start, and it was challenging to implement in parallel due to issues with deadlock. The decision on whether or not to change a given spin is

contingent on a calculation involving the spins above, below, to the left, and to the right of it, so I initially decided to lock everything in “up -> left -> center -> right -> down” order to avoid deadlock.

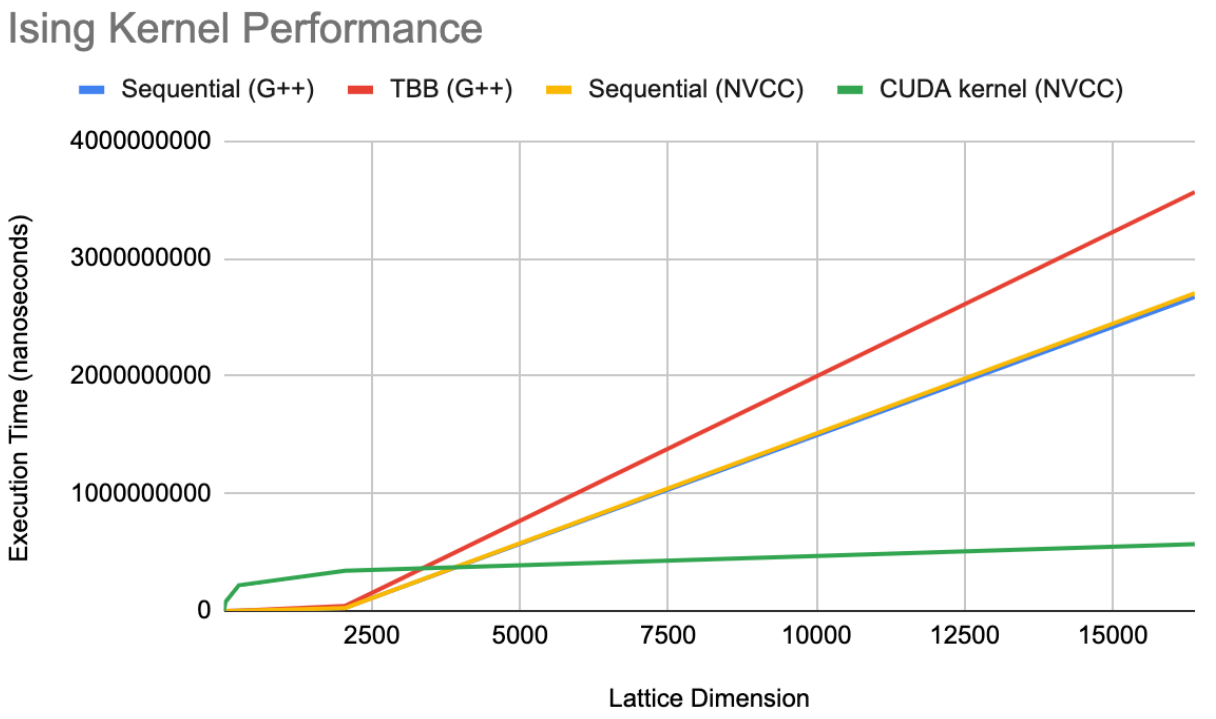
This strategy generally worked well, but failed when it came to spins on the edge of the grid, since the adjacent spins would be on the other side of the grid and therefore interfere with the order. I therefore had to come up with nine different cases which have been laid out in the attached diagram. In each case, the target spin’s status as an edge or corner case is determined, its necessary adjacent spins are identified, and then all locks are acquired from top to bottom and left to right. The grid is an oversimplification; the grids in the test cases are significantly larger than 3x3, and thus most cases fall within the middle case, with more cases than depicted also falling within the edge (but not corner) cases.

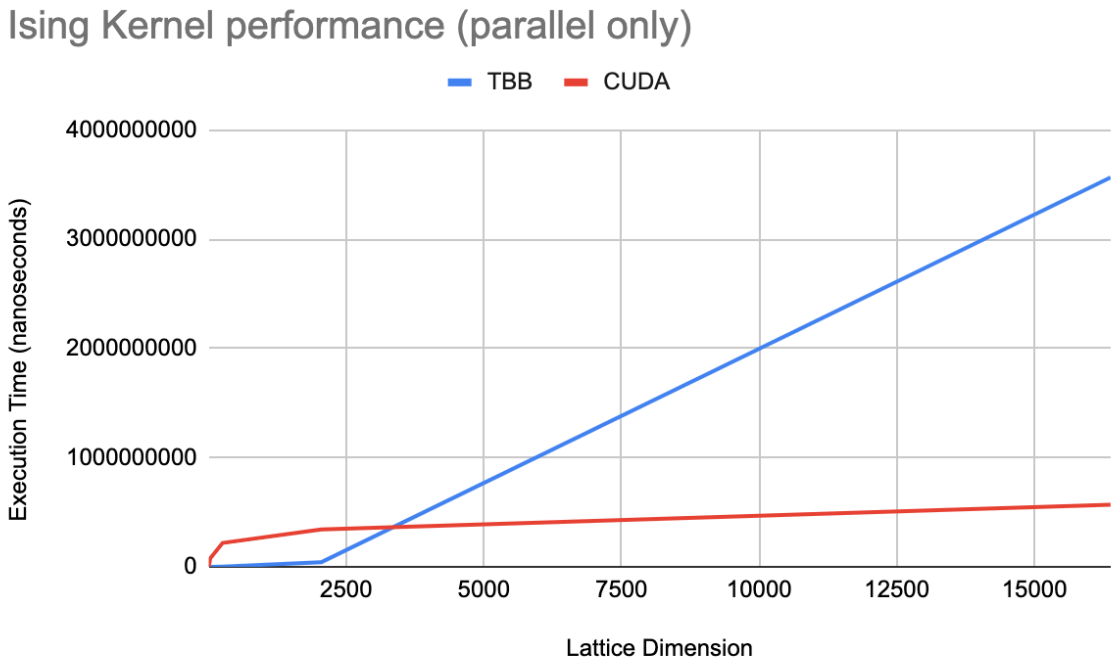


Acquiring locks in this order solved the problem of deadlock. However, mutexes come with quite a substantial amount of computational overhead, and having to wait for and acquire five of them for every operation resulted in a significant slowdown of my program. Ultimately, my TBB program was only able to run about 75% as fast as my sequential code due to the overhead, and it is the only program in this project that doesn’t result in speedup.

My CUDA code for the Ising kernel wound up more successful than the TBB code I wrote, despite following very similar instructions. The main advantages of my CUDA code are that I'm able to run them on far more threads, as well as only making use of atomic variables without mutex functionality, mainly since GPU code doesn't have functionality for mutexes. Additionally, my CUDA code has built-in backoff functionality, where a thread will give up on executing an instruction if it is in conflict long enough. These factors combined result in about 4.5x speedup.

Lattice dimension	Sequential (G++)	TBB (G++)	Sequential (NVCC)	CUDA kernel (NVCC)	TBB speedup	CUDA speedup
4	106.6	3383.8	99.2	140613.2	0.03150304392	0.00070548142
32	10320.6	27369.6	3596.4	77266665.4	0.3770826026	0.00004654529843
256	235905.4	305874.4	226621.6	217671562.2	0.7712492448	0.001041117166
2048	25022461.2	41225634.6	24565796.6	341080637.6	0.6069636391	0.07202342758
16384	2674120773	3566469771	2703680746	568243037.6	0.7497948796	4.757965461





Conclusion

From this project I learned about the various strengths and limitations of TBB parallelization on the CPU and handwritten CUDA kernels on the GPU. TBB is overall more user-friendly and easier, allowing for functionality like passing data structures into parallel functions, but gives the programmer less control over how to run the program, as it isn't possible to choose how many blocks and threads to run. It is possible to make the choice of how many threads to run on the GPU, but at the cost of having to pass in every data structure as a one-dimensional pointer with a size known prior to runtime. As a result of having more control over the program, handwritten CUDA kernels are generally faster, provided the programmer knows what they are doing. If I were to pick this project up again sometime in the future, I would improve my 2MM program to be able to make use of the GPU for matrix multiplication, as well as try again to figure out a solution to the Collatz conjecture that involves dynamic programming.