# Numerical Integration

Jerome Andaya Jr.
Jeffrey Lin

## Table of Contents

## Intro

“Someone once told me not to bite off more than I could chew.
I said I’d rather choke on greatness than nibble on mediocrity”

-- Unknown

We chose to bite at the optimization of numerical integration by Gauss-Legendre quadrature. This problem involves the calculation of weights, calculation of the function you want to integrate, and a dot product of the two.

This is the function we are trying to integrate: the first Feynman Loop.

$$P(m) = \int_{-1}^{1} dp_0 \int_{-1}^{1} dp_1 \int_{-1}^{1} dp_2 \int_{-1}^{1} dp_3 \frac{1}{\left(\sin^2(\pi p_0/2) + \sin^2(\pi p_1/2) + \sin^2(\pi p_2/2) + \sin^2(\pi p_3/2) + m^2\right)^2}$$

## Algorithm

The naive algorithm is as follows.

First we must determine the coefficients of the legendre polynomial and use root finding by the Newtonian method to find its roots. These roots will be transformed and eventually become the weights. Next, we need to solve for various points of the first Feynman loop. We do this simple function evaluation. Next, we employ Gaussian Elimination on the Legendre abscissae and the following system of equations to determine the weights.

$$\int_{-1}^{1} x^0 = w_1 + w_2 + \cdots + w_N$$

$$\int_{-1}^{1} x^1 = x_1 w_1 + x_2 w_2 + \cdots + x_N w_N$$

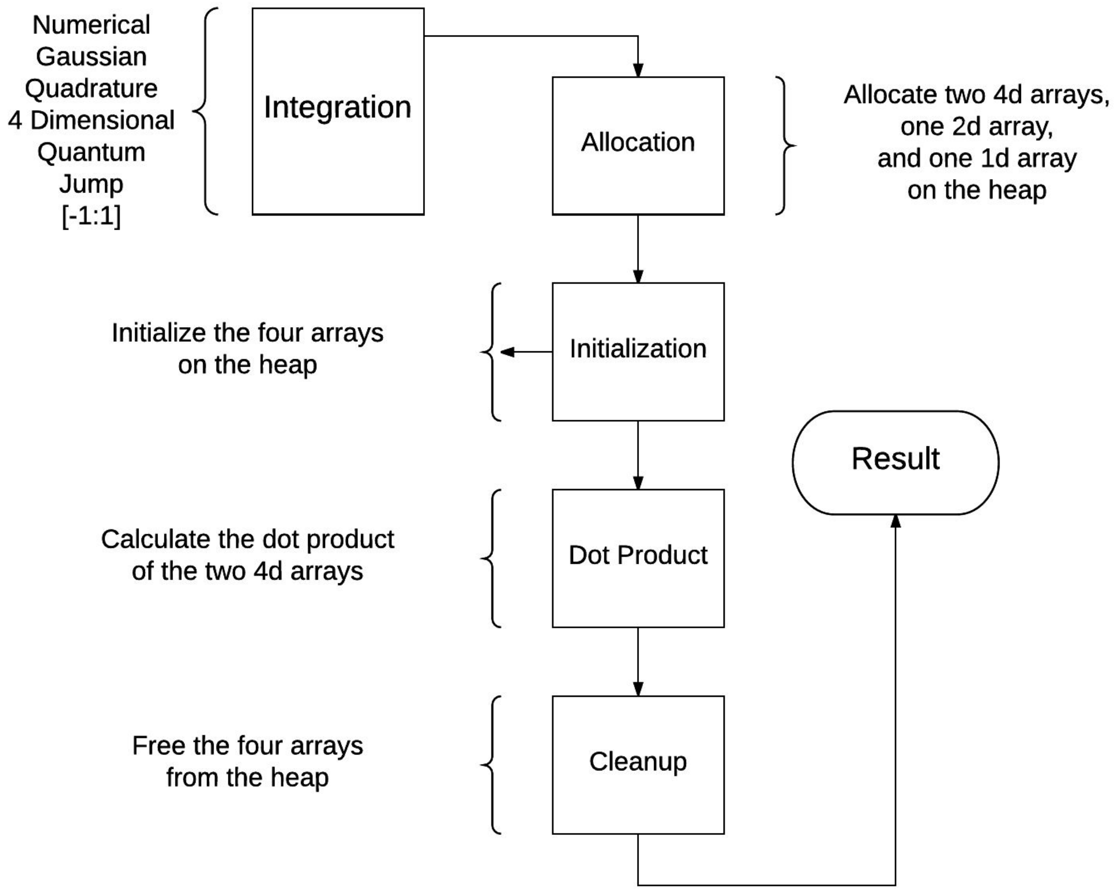
$$\vdots$$

$$\int_{-1}^{1} x^{N-1} = x_1^{N-1} w_1 + x_2^{N-1} w_2 + \cdots + x_N^{N-1} w_N$$

After the weights have been determined, we need to perform the dot product of the weights and the Legendre abscissae to find the resultant integral. However, because the is a four dimensional problem, we must extend the dot product of two n-sized one dimensional vectors to the product of two $n^4$-sized four dimensional vectors. The four dimensional weight vector contains every permutation of the product of weights, (0,0,0,0) → (n-1, n-1, n-1, n-1). The four dimensional function vector contains the function evaluation at every point (x,y,z,t) within the bounds (-1,1).

## Naive Implementation

The naive implementation of the above algorithm is as follows.


Numerical Gaussian Quadrature 4 Dimensional Quantum Jump [-1:1]
Integration
Allocation
Allocate two 4d arrays, one 2d array, and one 1d array on the heap
Initialize the four arrays on the heap
Initialization
Result
Calculate the dot product of the two 4d arrays
Dot Product
Free the four arrays from the heap
Cleanup

As the diagram suggests, the naive implementation breaks down the algorithm step by step and follows the steps set out above.

In the allocation stage, we allocate space for four very large data structures. One n-sized one dimensional vector will hold each Legendre abscissa. Another n-sized one dimensional vector will hold  the one dimensional weights after the calculations by gaussian elimination. A very large $n^4$-sized four dimensional vector holds the every permutation of the weights in four dimensions when multiplied with one another. The last $n^4$-sized four dimensional vector holds the function evaluation for every combination of points within (-1, 1)

Initialization is a catch-all term in the diagram that encompasses the determination of the Legendre abscissae, gaussian elimination and determination of the weights, as well as the function evaluation to initialize the four data structures we used.

The dot product is a very computationally taxing dot product across two four dimensional vectors. The cleanup is the time to deallocate the very large arrays we used.

Indeed the cost for each of these operations is high, and we need to find a better way than simply following the steps of the method. The reason we performed the operations in this order is because this is the way we learned it previously. The way we learned it was based on mathematical proof rather than algorithmic analysis. Essentially, this proves the correctness of the algorithm, but the speed of the algorithm was absurdly slow.

## Analysis and Optimizations

Analysis of the code led us to three major optimizations. We decided to reduce allocation space tremendously as well as the bloat of the code's loads and stores by reusing as many local variables as we could and storing to large data structures only as necessary. We also try to cut down the total number of for loops in the code by completing as much work as can be done in one for loop before exiting to concatenate the work into fewer for loops.

In our original code, we created multiple 4D data structures to store calculated values and weights that were to be multiplied during our Dot Product stage. Our values were found by plugging in calculated Legendre abscissae into the First Feynman Loop for each of the four variables. In the Feynman Loop, we would need to perform  an operation of

$$\sin^2(\pi p_1/2)$$

on each of the four variables. These four variables would correspond to Legendre abscissae. We realized that we were doing this operation on the same Legendre abscissae $N^3$ times.  To avoid doing such unnecessary work, we created an array that would store the evaluation of this operation on each abscissae. We found the same could be done for our weights as well. Each time we take a dot product, we had taken values from another 4D data structure whereas we could have easily just looked at the array we had already made when solving gaussian elimination.

Next, we decided to optimize the determination of the Legendre abscissae, hoping to improve both speed and accuracy of our values.  Our original method for calculating abscissae and weights came from lectures in another class. We would generate abscissae by using Newton's Method on an approximation of the abscissae in attempt to have it converge to an accurate enough value we could use. We found that even after 100,000 iterations of Newton's Method, some abscissae would always bounce between two values. We decided to do some more research and found a method provided by RosettaCode. Their method uses Newton-Raphson iterations to help approximate abscissae and apply a variant of Newton's Method on the value to get it to the desired precision, in our case 16 digits of precision. We compared the values from our original code and the improved algorithm and found that not only did it complete in a few iterations, it was also accurate to 16 digits compared to a lookup table we found online.

In attempt to find weights for our integration, we attempted to use our method of Gaussian Elimination once again. Our weights created from Gaussian Elimination were greater than 1, which is the limit of weights. In the same code provided by RosettaCode, they had a method for generating weights.

$$w_i = \frac{2}{\left(1 - x_i^2\right) [P'_n(x_i)]^2}$$

P'n($x_i$) refers to a derivative value calculated while generating the abscissae and $x_i$ refers to the abscissae. These weights were also accurate to 16 digits of precision on the lookup table we found.

After optimizing calculating weights and abscissae, we wanted to see if OpenMP would help speed up our code. We placed the #pragma omp parallel at every for loop and found that the fastest runtime was in the outermost loop.

Finally, we decided to optimize the function evaluations, as well as the weight evaluation by using CUDA Thrust. Thrust is a high level library that helps perform the kernel correctly by writing Thrust code. The reason we chose to use Thrust as is because of the four dimensional nature of the problem. Blocks are designed to be used on three dimensional data or smaller. We did not know the optimal block organization: indexing four dimensions of data using block index as the fourth parameter, indexing two dimensions by block index and two by thread index, or indexing the fourth parameter with thread index, and using block index for the other three dimensions.

Our rationale was that the Thrust library is optimized within itself, i.e. using Thrust operations on Thrust data structures would yield the best thread, block, and grid organizations for the task. Therefore, we used Thrust to decide this for us. The data structures used were the Thrust host_vector, device_vector, and counting_iterator. The operation used was the for_each_n function which completes the operation, but guarantees no order, and chooses the fastest order possible (Essentially what blocks do). The functor used was our own defined permute_functor. Lastly we use the Thrust reduce function to sum the results of the for_each_n(permute_functor).

The Thrust host_vector and device_vector are classes that help facilitate the functions of CUDA memory allocation and memory copy across the host and the device. This helps us move three things into and out of the device. We use host_vector to copy the values obtained by our sinusoidal calculations as well as our weight values. Afterwards, we memory copy these values from the host to the device by creating a device_vector containing the same values. Finally, we create a device_vector of size $n^4$ initialized to all zeroes.

Next we use the for_each_n function from the Thrust library. This function acts as our CUDA Kernel in a sense, since it manages what happens to the data structures we used and therefore the blocks and threads that operate on the data structures. Let's go through it.

```
thrust::for_each_n(
    thrust::device,
    thrust::counting_iterator<int>(0),
    (n*n*n*n),
    permuteFunctor(d_sinVals, d_gQ_Soln, d_sumVals));


sum = thrust::reduce(d_sumVals.begin(), d_sumVals.end(), (int) 0, thrust::plus<int>());
```

```
struct permuteFunctor
{
    const double* sinVals_f;
    const double* gQ_Soln_f;
    double* sumVec_f;
    int n;

    permuteFunctor(thrust::device_vector<double> const& sinV,
                   thrust::device_vector<double> const& gQV,
                   thrust::device_vector<double> & sumV)
    {
        sinVals_f = thrust::raw_pointer_cast(sinV.data());
        gQ_Soln_f = thrust::raw_pointer_cast(gQV.data());
        sumVec_f = thrust::raw_pointer_cast(sumV.data());
        n = sinV.size();
    }



    __device__
    void operator()(int x)
    {
        int nn = (n*n);
        int nnn = (n*n*n);

        int l = (x % (n));
        int k = (x % (nn)  / (n));
        int j = (x % (nnn) / (nn));
        int i = (x         / (nnn));

        double sumSin = sinVals_f[i] + sinVals_f[j] + sinVals_f[k] + sinVals_f[l] + M2;
        double prodGQ = gQ_Soln_f[i] * gQ_Soln_f[j] * gQ_Soln_f[k] * gQ_Soln_f[l];

        sumVec_f[(i*(nnn)) + (j*(nn)) + (k*(n)) + l] = (1.0/(sumSin*sumSin)*prodGQ);

    }
};
```

The first code snippet shows the “kernel calls”. The counting_iterator acts as an $n^4$ sized array with its index as the contents, but when passed into a functor gives the single value. What’s more is that it does not take up space like an array, but the value is generated on the fly, reducing space and time. The iterator comes into the functor as the value x. By calling the permute functor struct within the for_each_n with arguments to the three device_vectors outlined above, We are able to create raw pointers to the data in the device_vectors and use them within my functor.

Within the actual functor operator, we use the iterator as indices to my three device_vectors. Our iterator value is in base 10, but instead, we interpret the value as base n with four base-n digits. Each digit is a representation of the vector indices (i, j, k, l). By doing some modulo and division magic, we obtain each digit of the base-n number, and use them as indices to the vectors. Once each thread finds its corresponding (i, j, k, l) indices, the corresponding sinusoidal calculation and weight are extracted. They are multiplied and the result is stored in the corresponding element of the result vector. Lastly, back in the host code, the Thrust reduce function is called to reduce all the vector elements into its sum.

## Data

| Array Length | Improved Algo(ms) | OpenMP(ms) | GPU(ms) |
|---|---|---|---|
| 25 | 47.2051524 | 43.2131017 | 174.7024933 |
| 50 | 300.4422824 | 279.514343 | 183.9050198 |
| 75 | 1392.411498 | 1294.931103 | 206.8363697 |
| 100 | 4278.951602 | 4036.609501 | 267.8543725 |
| 125 | 10418.17427 | 9762.643722 | 393.0992675 |
| 150 | 21577.06628 | 20203.19885 | 631.9053304 |

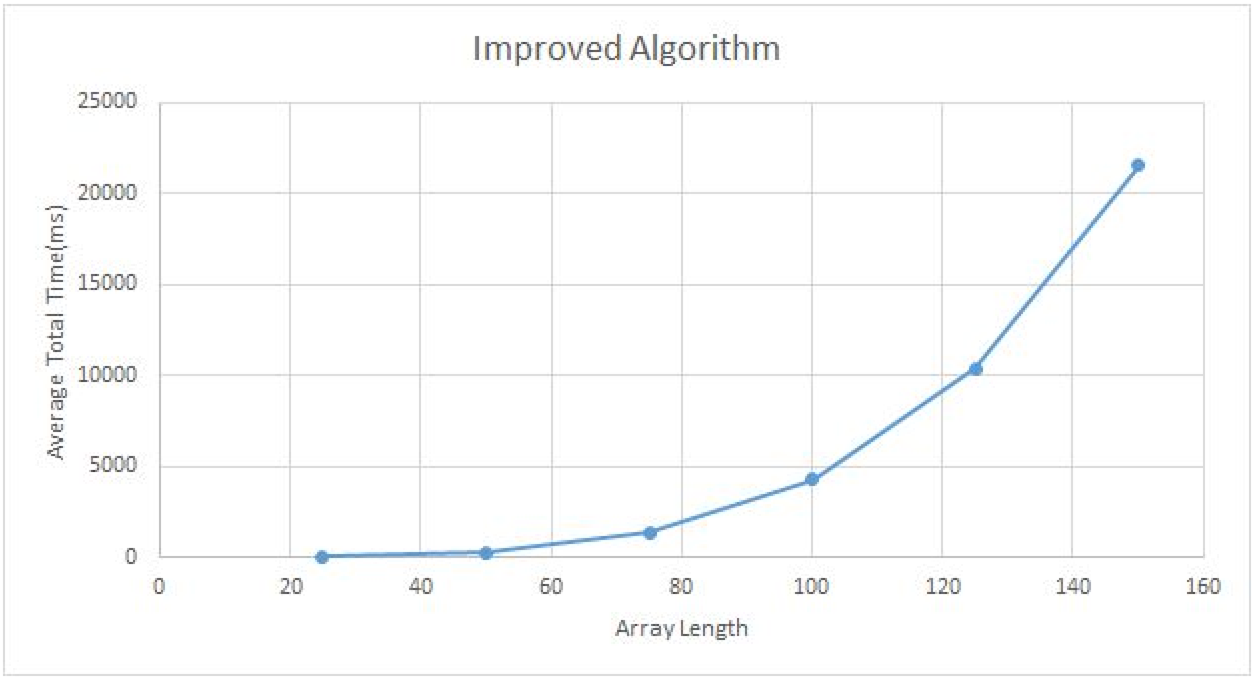
Looking at the data, we can see that even though GPU’s lose out to both the improved algorithm and the OpenMP for smaller Array Lengths, as lengths increase steadily, GPUs begin to blow the non threaded code out of the water. This is probably due to the overhead we have in transferring the data over from the host to the device and then back again, as well as setting up other variables like our thrust library.In our initial test for our baseline code we were able to compute Array Lengths of 100 in over 20,000ms. Our Improved Algorithm shortens this time by 4.6x and our OpenMP by 5x. Our GPU code solves this array in 268 ms which is almost 75x faster for computing $100^4$ values. As Array Lengths continue to increase, the GPU will significantly outperform its opposition. We can see that when Array Lengths double from 75 to 150, Both the improved and the OpenMP times increase over 10- fold. This is because as Array Lengths double the total number of operations actually increase 16- fold. Despite this large increase in operations, the GPU only sees an increase of over 3x when comparing the average 75 Array Length time to the average 150 Array Length time.

## Conclusion and Thoughts for the Future

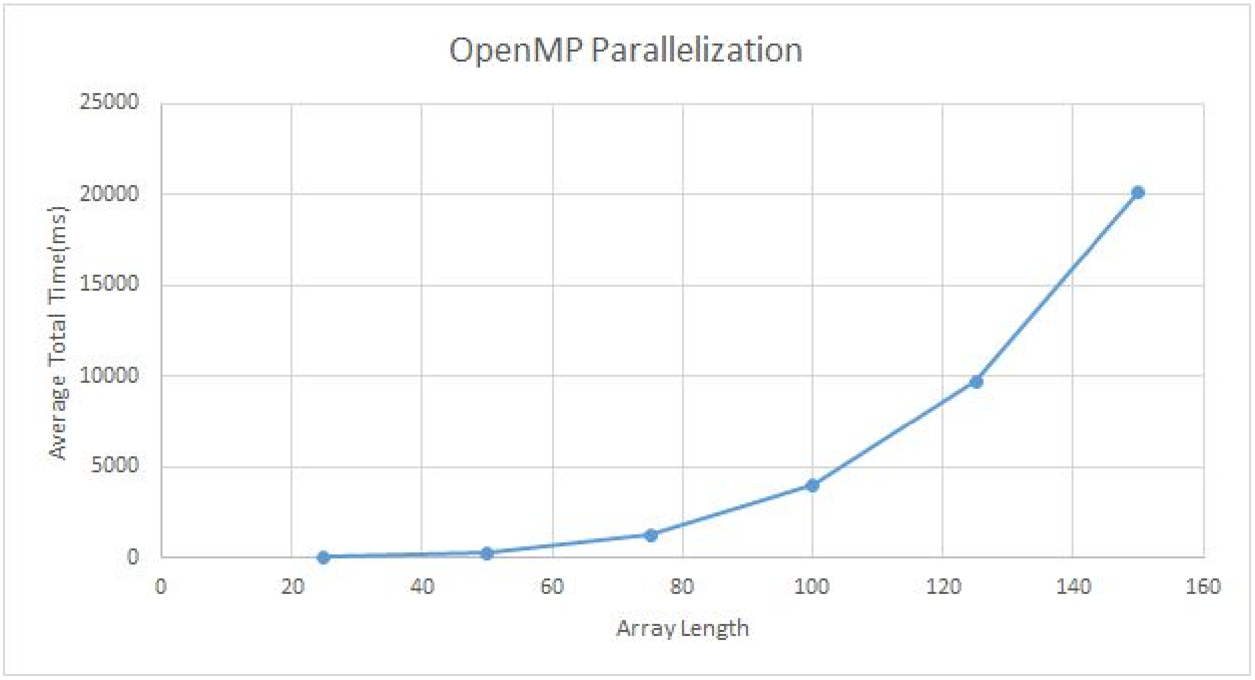
In this project, we confirmed our knowledge that GPU's significantly outperform any scalar CPU process. We also confirmed our knowledge that OMP optimizes with very little work but gets slightly better results than the unoptimized versions. All in all, the exercise of optimization was successful.

We chose to optimize an entire program instead of a subroutine. This was very difficult. In the end, we realize, we could optimize so much more. We could also do a complete code restructure to avoid function calls to unoptimized subroutines and make it more GPU friendly. We could turn those auxiliary functions into GPU code as well. Additionally we could be more stringent with our results, only moving forward when we have 16 digits (double) precision at every stage. However, with our time constraints we were not able to get 16 digits at every stage. In some stages, we resolved to keep 10-12 digits of precision, however with the nature of the problem, the error increases with every multiplication. Because of this, the actual result of the calculation is erratic. We attribute this to the lack of precision in our code, however, we also do not fully understand the function that we are trying to integrate. Because it is a quantum mechanics equation, there may be hidden consequences to not computing perfect 16 digit precision every time. If we only had time for one more optimization, this would be our focus.

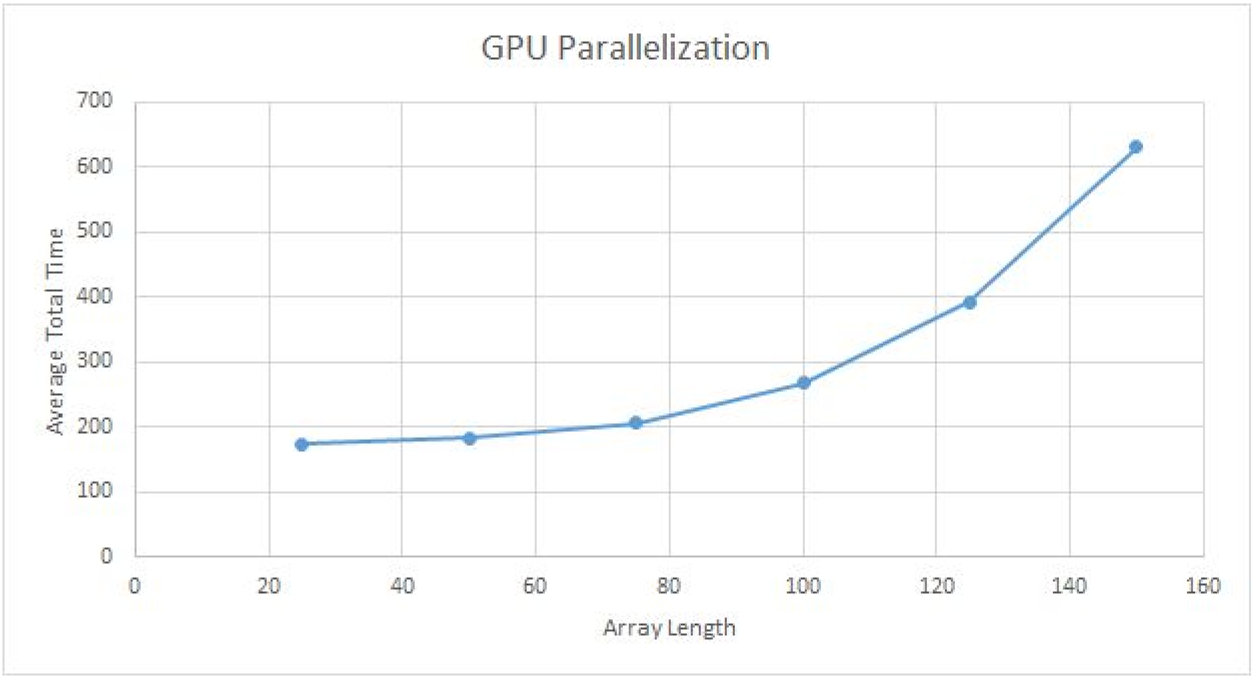
**Appendix**



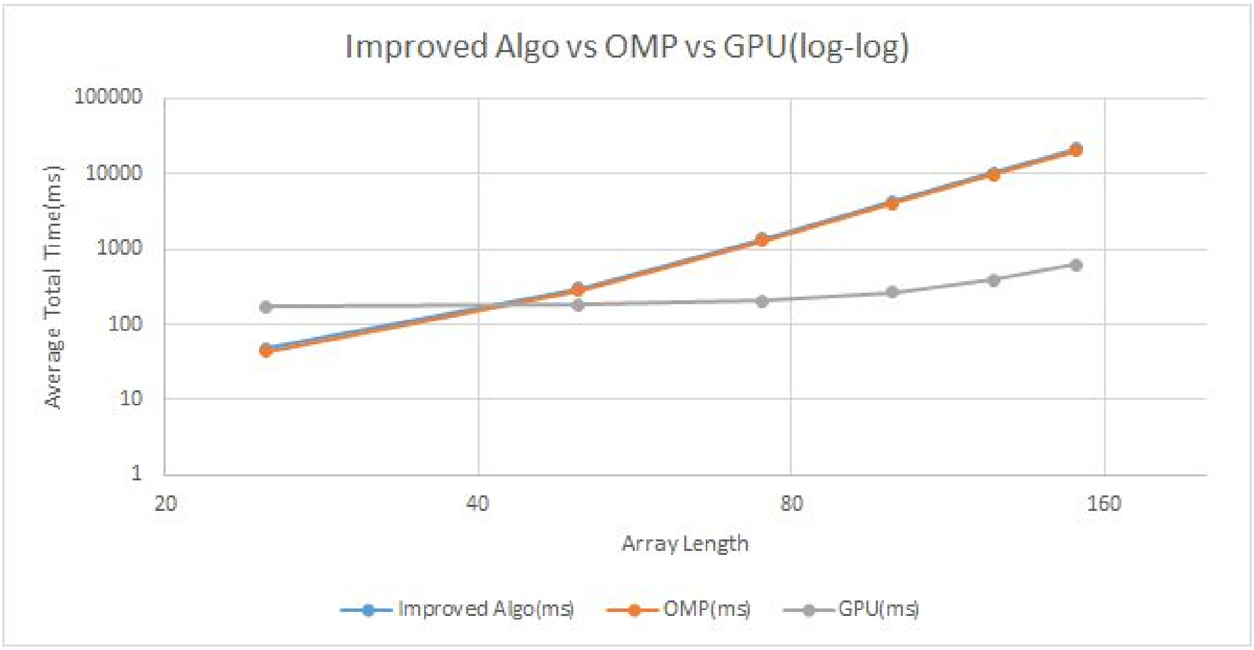
Graph 1. Optimized Algorithm (Average of 10 runs)



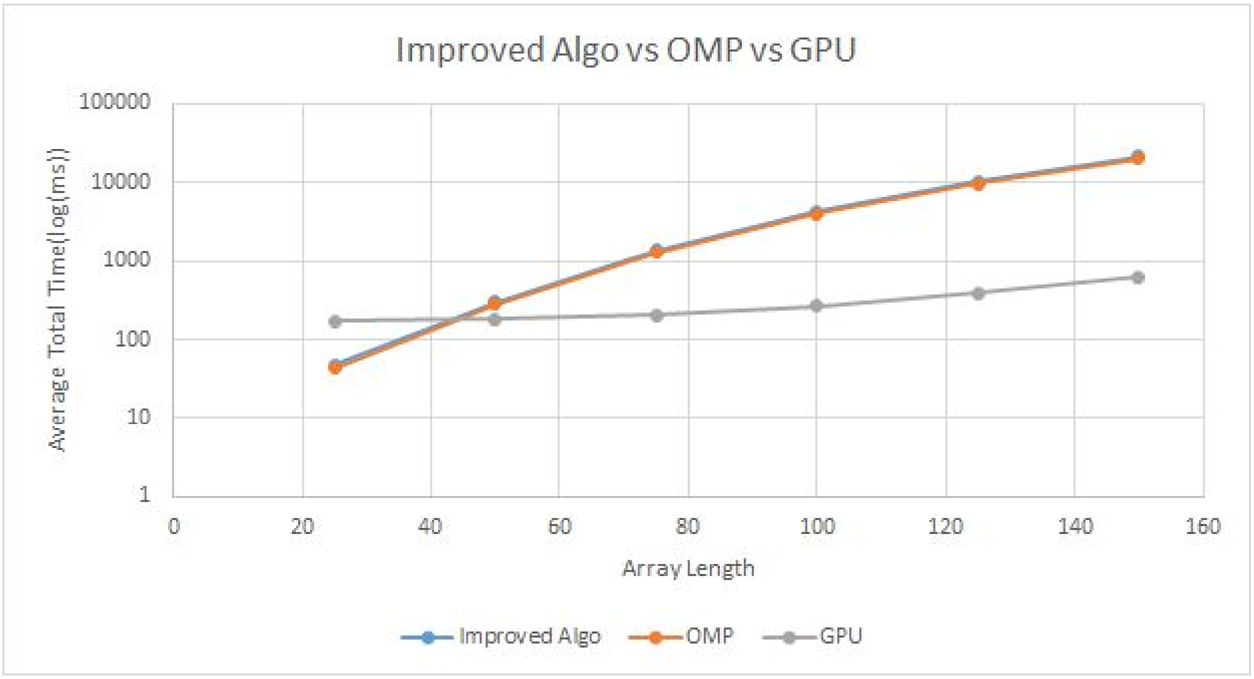
Graph 2. OpenMP Parallelized Algorithm (Average of 10 runs)



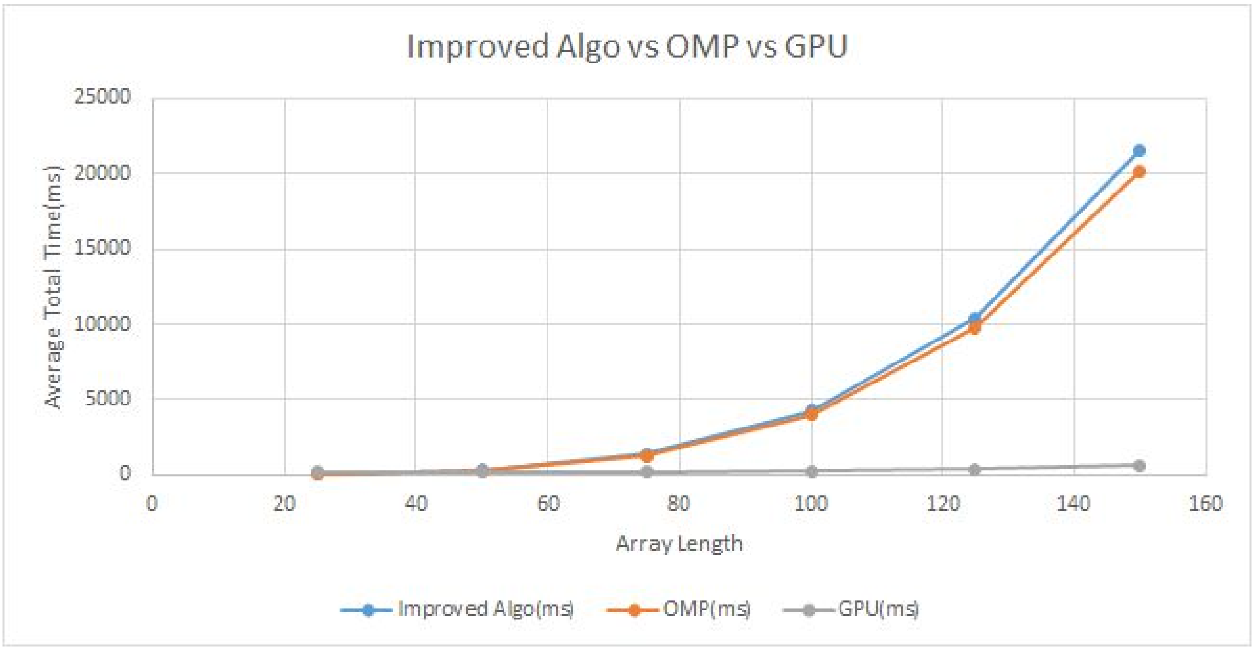
Graph 3. GPU Parallelized Algorithm (Average of 10 runs)



Graph 4. Log Log Graph of Three Optimizations



Graph 5. Log Scale of Three Optimizations



Graph 6. Linear Graph of Three Optimizations

## Sources

http://mathworld.wolfram.com/Legendre-GaussQuadrature.html

https://rosettacode.org/wiki/Numerical_integration/Gauss-Legendre_Quadrature

http://thrust.github.io/doc/modules.html