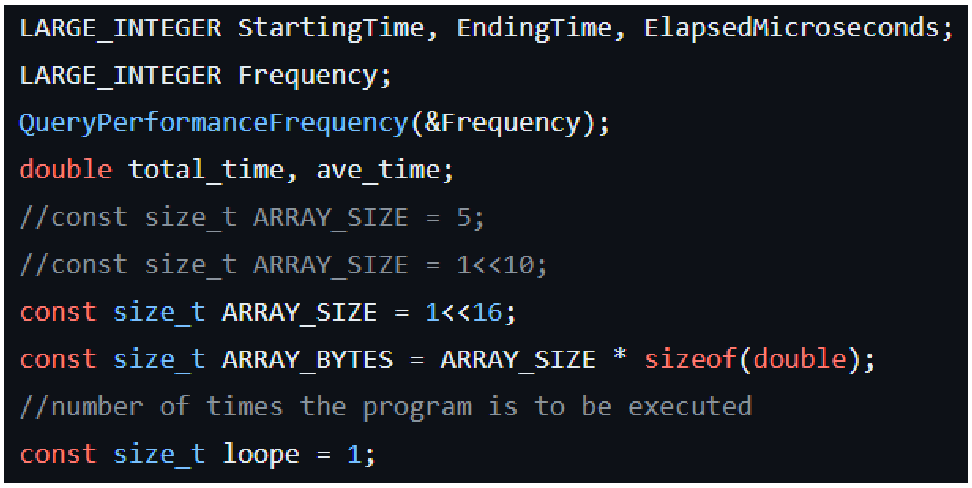
Chan, Ethan Lester
Dolon, John Michael
Lu, Andre Giancarlo
Teng, Adriel Shanlley

Speeding Up DFT and IDFT using CUDA
Integrating Project: Final Update Document

In Milestone 2, we were able to create the DFT and IDFT logic in C programming. In this final project update, we will discuss the last part of the project: the implementation of DFT and IDFT in CUDA. We also implemented DFT & IDFT in Matlab to compare the speed between Matlab and CUDA. Lastly, we tried using the DFT & IDFT built-in functions in Matlab, and see if it is faster than CUDA. In this document, code snippets of the implementation will be shown along with some explanations of the codes.

A. CUDA Initialization

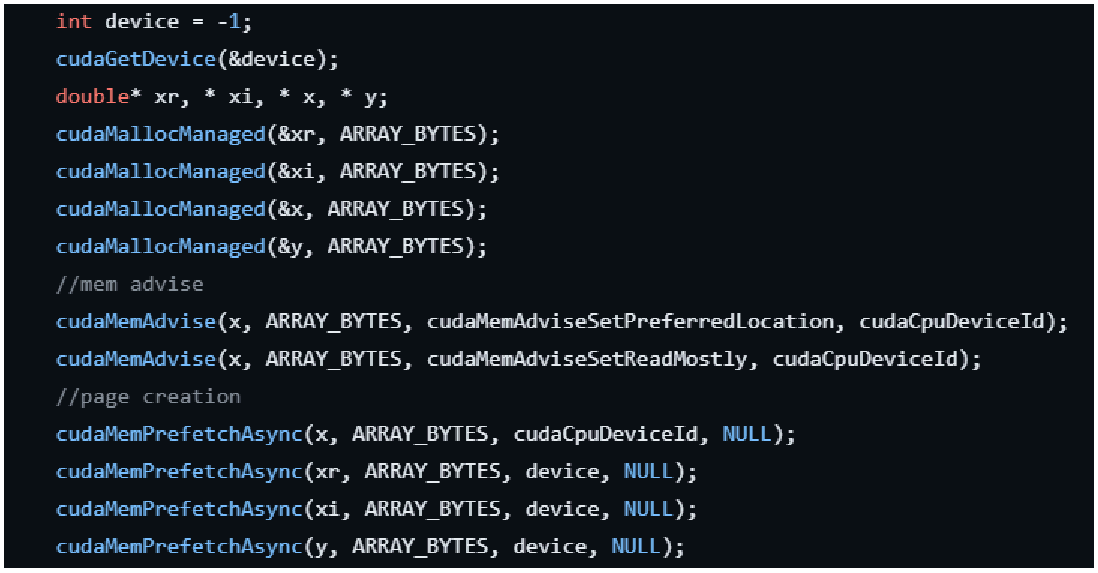
To start, the CUDA implementation is as follows:



```
LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
LARGE_INTEGER Frequency;
QueryPerformanceFrequency(&Frequency);
double total_time, ave_time;
//const size_t ARRAY_SIZE = 5;
//const size_t ARRAY_SIZE = 1<<10;
const size_t ARRAY_SIZE = 1<<16;
const size_t ARRAY_BYTES = ARRAY_SIZE * sizeof(double);
//number of times the program is to be executed
const size_t loope = 1;
```

Code Snippet #1: Initialization

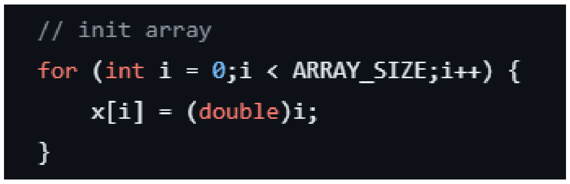
We first initialized the variables *StartingTime*, *EndingTime*, and *ElapsedMicroseconds* to know the time elapsed of the DFT function. We also initialized *Frequency* to get the frequency of the performance counter through *QueryPerformanceFrequency()*. Then, we initialized *total_time* and *ave_time* to get the total and average times of the CUDA program. Lastly, we set *ARRAY_SIZE* to what we want (ex: 2^{16}, 2^{20}, etc.), set *ARRAY_BYTES* to scale with *ARRAY_SIZE*, and set *loope* to how many times we want to loop the program (ex: 1, 2, 10, etc.)



```
    int device = -1;
    cudaGetDevice(&device);
    double* xr, * xi, * x, * y;
    cudaMallocManaged(&xr, ARRAY_BYTES);
    cudaMallocManaged(&xi, ARRAY_BYTES);
    cudaMallocManaged(&x, ARRAY_BYTES);
    cudaMallocManaged(&y, ARRAY_BYTES);
    //mem advise
    cudaMemAdvise(x, ARRAY_BYTES, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);
    cudaMemAdvise(x, ARRAY_BYTES, cudaMemAdviseSetReadMostly, cudaCpuDeviceId);
    //page creation
    cudaMemPrefetchAsync(x, ARRAY_BYTES, cudaCpuDeviceId, NULL);
    cudaMemPrefetchAsync(xr, ARRAY_BYTES, device, NULL);
    cudaMemPrefetchAsync(xi, ARRAY_BYTES, device, NULL);
    cudaMemPrefetchAsync(y, ARRAY_BYTES, device, NULL);
```

Code Snippet #2: Memory Management, MemAdvise, & Page Creation for Newly-Created Arrays

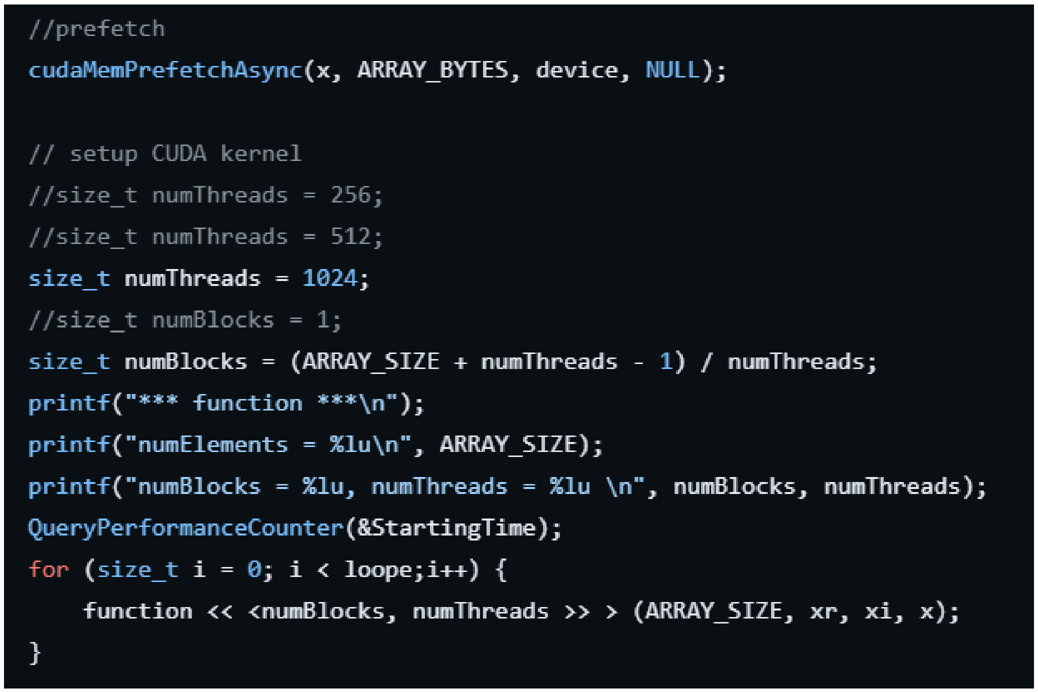
After initialization, we can use *cudaGetDevice()* to get the device (GPU) to be used in the program. Then, we set a few new variables: *xr*, *xi*, *x*, and *y*. *xr* will hold all of the real components of the DFT results, *xi* will hold all of the imaginary components of the DFT results, *x* will hold the original signal values, and *y* will hold the results of the IDFT function. We set the memory allocated by these variables using *cudaMallocMananged* and *ARRAY_BYTES*, followed by some MemAdvise for *x*, then page creation for *xr*, *xi*, *x*, and *y*.



```
    // init array
    for (int i = 0;i < ARRAY_SIZE;i++) {
        x[i] = (double)i;
    }
```

Code Snippet #3: Array Initialization

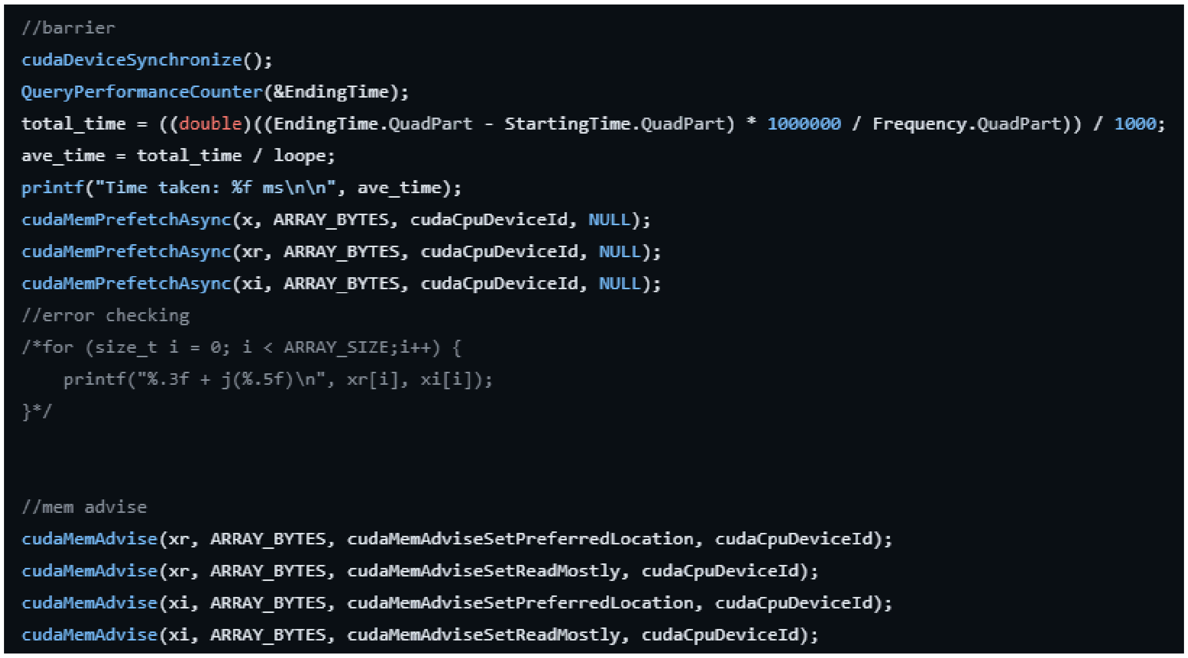
Then, we start initializing the values of *x*. The values will simply be index *i*, so when *i* = 0, *x* = 0; when *i* = 1, *x* = 1, etc.



```
//prefetch
cudaMemPrefetchAsync(x, ARRAY_BYTES, device, NULL);

// setup CUDA kernel
//size_t numThreads = 256;
//size_t numThreads = 512;
size_t numThreads = 1024;
//size_t numBlocks = 1;
size_t numBlocks = (ARRAY_SIZE + numThreads - 1) / numThreads;
printf("*** function ***\n");
printf("numElements = %lu\n", ARRAY_SIZE);
printf("numBlocks = %lu, numThreads = %lu \n", numBlocks, numThreads);
QueryPerformanceCounter(&StartingTime);
for (size_t i = 0; i < loope;i++) {
    function << <numBlocks, numThreads >> > (ARRAY_SIZE, xr, xi, x);
}
```

Code Snippet #4: CUDA Kernel Setup & Some Prefetching for DFT

Next, we prefetch *x*, then setup the CUDA kernel. We set our *numThreads* to 1024 for maximum efficiency. We scale *numBlocks* with *numThreads*. Then, we display the number of elements (*ARRAY_SIZE*), the number of blocks (*numBlocks*), and the number of threads (*numThreads*). Then, we can start timing the program time and execute the DFT function.

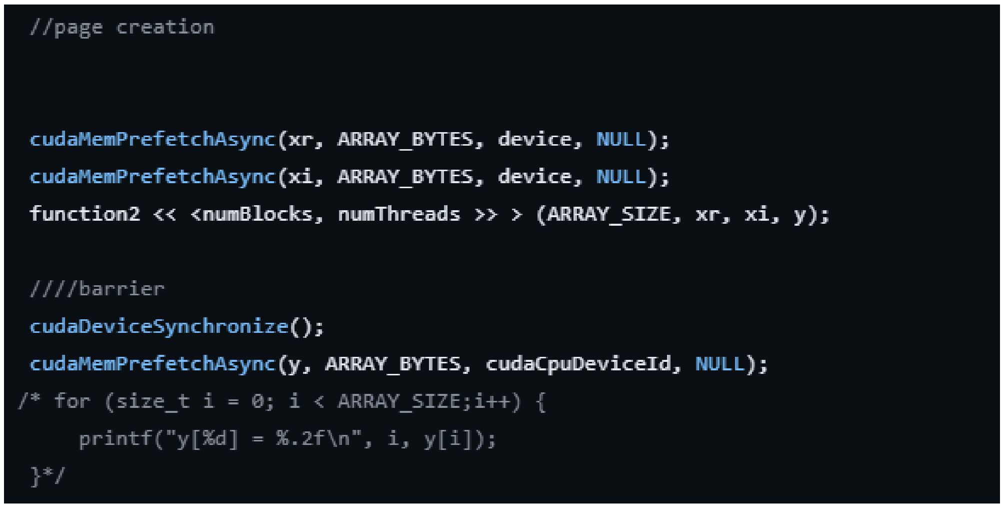


```
//barrier
cudaDeviceSynchronize();
QueryPerformanceCounter(&EndingTime);
total_time = ((double)((EndingTime.QuadPart - StartingTime.QuadPart) * 1000000 / Frequency.QuadPart)) / 1000;
ave_time = total_time / loope;
printf("Time taken: %f ms\n\n", ave_time);
cudaMemPrefetchAsync(x, ARRAY_BYTES, cudaCpuDeviceId, NULL);
cudaMemPrefetchAsync(xr, ARRAY_BYTES, cudaCpuDeviceId, NULL);
cudaMemPrefetchAsync(xi, ARRAY_BYTES, cudaCpuDeviceId, NULL);
//error checking
/*for (size_t i = 0; i < ARRAY_SIZE;i++) {
    printf("%.3f + j(%.5f)\n", xr[i], xi[i]);
}*/


//mem advise
cudaMemAdvise(xr, ARRAY_BYTES, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);
cudaMemAdvise(xr, ARRAY_BYTES, cudaMemAdviseSetReadMostly, cudaCpuDeviceId);
cudaMemAdvise(xi, ARRAY_BYTES, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);
cudaMemAdvise(xi, ARRAY_BYTES, cudaMemAdviseSetReadMostly, cudaCpuDeviceId);
```

Code Snippet #5: Setting up Average DFT Function Time & IDFT

After the DFT function, we get the ending time of the DFT calculations. This will be important, as we will be comparing the execution times of DFT in CUDA and Matlab. Then, we compute the total time by subtracting *EndingTime* and *StartingTime*. We also compute the average time by dividing *total_time* by the number of loops (*loope*), and print the average time.

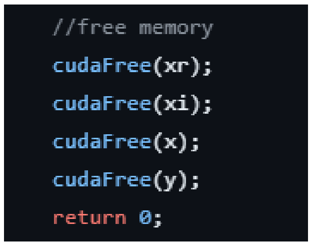
Then, we start setting up the IDFT function. We prefetch *x*, *xr*, and *xi*. Then, we MemAdvise *xr* and *xi*, as these arrays will be used for the IDFT function.



```
 //page creation



 cudaMemPrefetchAsync(xr, ARRAY_BYTES, device, NULL);
 cudaMemPrefetchAsync(xi, ARRAY_BYTES, device, NULL);
 function2 << <numBlocks, numThreads >> > (ARRAY_SIZE, xr, xi, y);

 ////barrier
 cudaDeviceSynchronize();
 cudaMemPrefetchAsync(y, ARRAY_BYTES, cudaCpuDeviceId, NULL);
/* for (size_t i = 0; i < ARRAY_SIZE;i++) {
     printf("y[%d] = %.2f\n", i, y[i]);
 }*/
```

Code Snippet #6: Page Creation for IDFT

Then, we do some page creation for *xr* and *xi*, then execute the IDFT function. Notice that we don't time the IDFT function. This is because the IDFT is mainly there for error checking, to see if the DFT function is accurate or not.

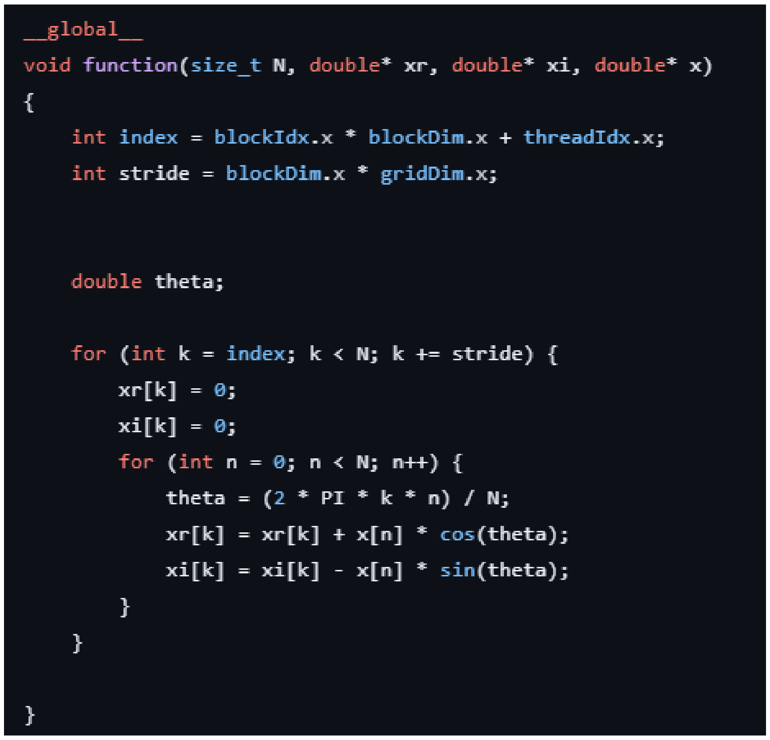


```
    //free memory
    cudaFree(xr);
    cudaFree(xi);
    cudaFree(x);
    cudaFree(y);
    return 0;
```

Code Snippet #7: Free Memory

Lastly, we free up the memory allocated by *xr*, *xi*, *x*, and *y*. This finishes the CUDA program. For the error-checking portion of the code, see part F of the document.

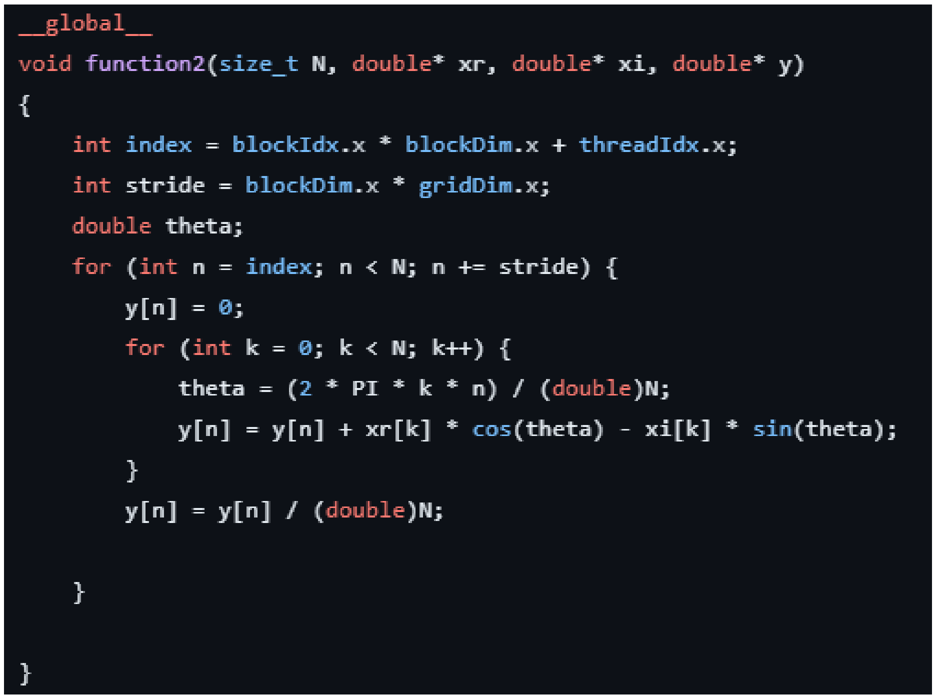
B. Discrete Fourier Transform (DFT) & Inverse Discrete Fourier Transform (IDFT) in CUDA



```
__global__
void function(size_t N, double* xr, double* xi, double* x)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;


    double theta;

    for (int k = index; k < N; k += stride) {
        xr[k] = 0;
        xi[k] = 0;
        for (int n = 0; n < N; n++) {
            theta = (2 * PI * k * n) / N;
            xr[k] = xr[k] + x[n] * cos(theta);
            xi[k] = xi[k] - x[n] * sin(theta);
        }
    }

}
```

Code Snippet #8: DFT Function

Next, we will be discussing the DFT function implementation. The DFT function accepts 4 parameters: *N* (the array size), *xr* (the array to store the real component of the DFT result), *xi* (the array to store the imaginary component of the DFT result), and *x* (the original array/signal). Recall that in milestone #2, we implemented the logic for DFT and IDFT in C. For CUDA, we simply copied the logic from the C code and added a grid-stride loop to it.

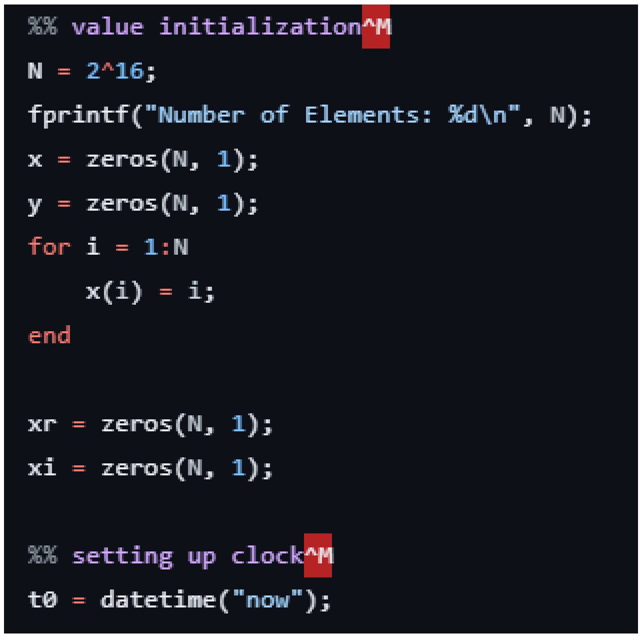


```
__global__
void function2(size_t N, double* xr, double* xi, double* y)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    double theta;
    for (int n = index; n < N; n += stride) {
        y[n] = 0;
        for (int k = 0; k < N; k++) {
            theta = (2 * PI * k * n) / (double)N;
            y[n] = y[n] + xr[k] * cos(theta) - xi[k] * sin(theta);
        }
        y[n] = y[n] / (double)N;

    }

}
```

Code Snippet #9: IDFT Function

Another code snippet is for the IDFT function. For this CUDA implementation, we copied the logic from the C implementation during milestone #2 and added a grid-stride loop.

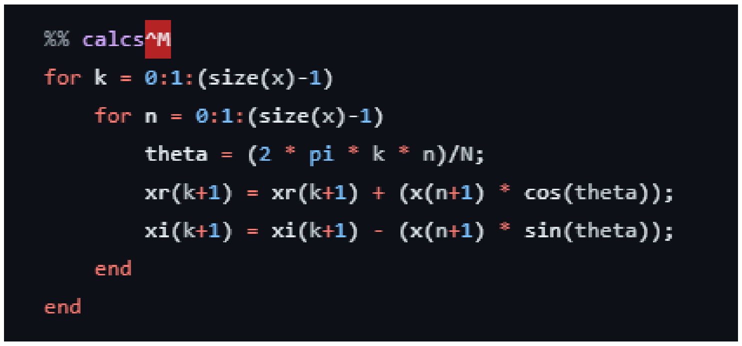
C. Implementation of DFT and IDFT in Matlab



```
%% value initialization^M
N = 2^16;
fprintf("Number of Elements: %d\n", N);
x = zeros(N, 1);
y = zeros(N, 1);
for i = 1:N
    x(i) = i;
end

xr = zeros(N, 1);
xi = zeros(N, 1);

%% setting up clock^M
t0 = datetime("now");
```

Code Snippet #10: Initialization

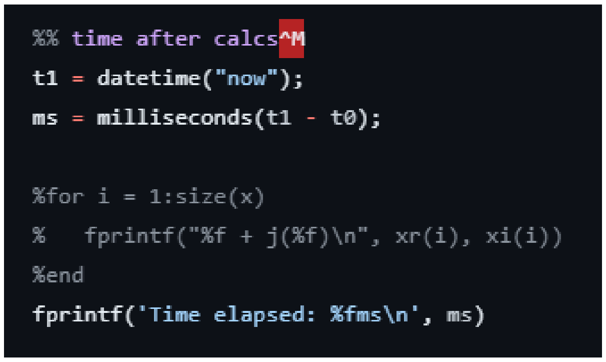
Next, we will discuss the Matlab implementation of DFT and IDFT. First, we initialize the array size *N* to what we want. Then, we initialize *x*, our original array/signal, and *y*, the array that will store the results of the IDFT function, with *N* number of zeros each. Then, like in the CUDA implementation, we initialized the values of *x* equal to index *i*. So when *i* = 0, *x* = 0; when *i* = 1, *x* = 1, etc. Then, we initialize *xr* and *xi* with *N* number of zeros. After, we setup the timer for the DFT calculations by placing the time before the calculations start at *t0*.



```
%% calcs^M
for k = 0:1:(size(x)-1)
    for n = 0:1:(size(x)-1)
        theta = (2 * pi * k * n)/N;
        xr(k+1) = xr(k+1) + (x(n+1) * cos(theta));
        xi(k+1) = xi(k+1) - (x(n+1) * sin(theta));
    end
end
```

Code Snippet #11: DFT Calculations

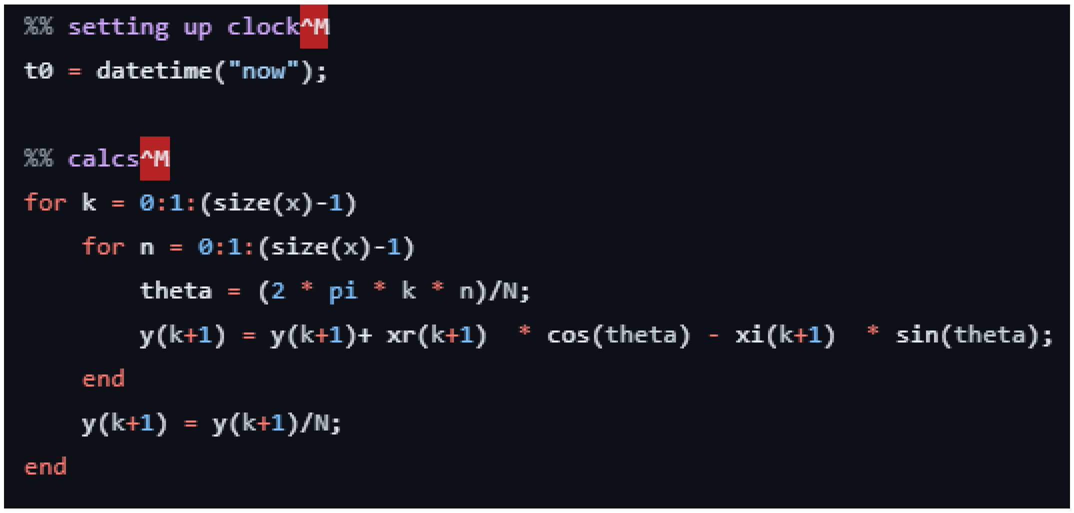
Then, the DFT calculations start. Note that we used the same logic as what we implemented in CUDA for a "fair" speed comparison.



```
%% time after calcs^M
t1 = datetime("now");
ms = milliseconds(t1 - t0);

%for i = 1:size(x)
%   fprintf("%f + j(%f)\n", xr(i), xi(i))
%end
fprintf('Time elapsed: %fms\n', ms)
```

Code Snippet #12: Printing out Elapsed Time of DFT Calculations

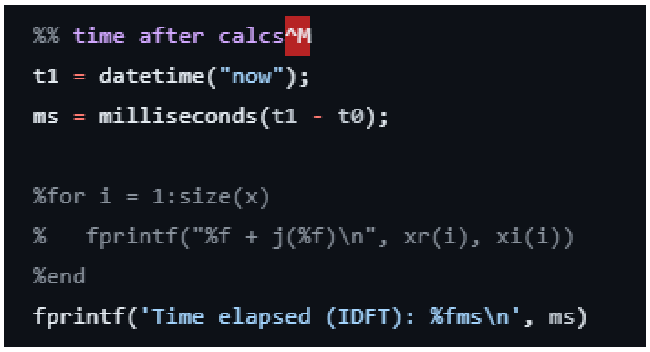
Once the DFT calculations are finished, we take the current time, store it in *t1*, and subtract *t1* and *t0* to get the elapsed time of the DFT calculations. We set this to milliseconds for more accurate time values. Then, we print the time elapsed in milliseconds.



```
%% setting up clock^M
t0 = datetime("now");

%% calcs^M
for k = 0:1:(size(x)-1)
    for n = 0:1:(size(x)-1)
        theta = (2 * pi * k * n)/N;
        y(k+1) = y(k+1)+ xr(k+1)  * cos(theta) - xi(k+1)  * sin(theta);
    end
    y(k+1) = y(k+1)/N;
end
```

Code Snippet #13: Time Setup for IDFT & IDFT Calculations

Next, we setup for IDFT. We start with resetting *t0* to the current time, then start calculating for the IDFT with *xr* and *xi*. Note that like with the DFT implementation, we used the same logic as what we implemented in CUDA for a “fair” speed comparison.



```
%% time after calcs^M
t1 = datetime("now");
ms = milliseconds(t1 - t0);

%for i = 1:size(x)
%   fprintf("%f + j(%f)\n", xr(i), xi(i))
%end
fprintf('Time elapsed (IDFT): %fms\n', ms)
```

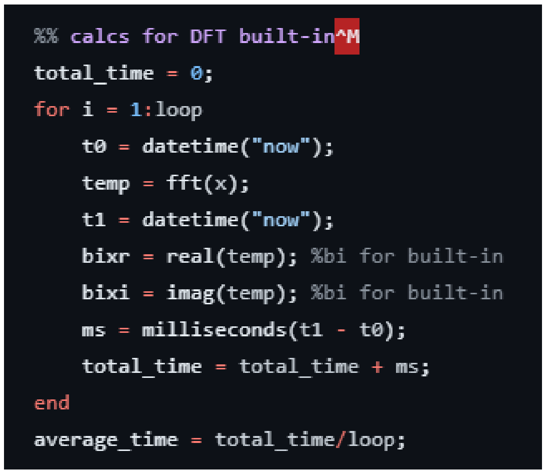
Code Snippet #14: Printing out Elapsed Time of IDFT Calculations

Once the IDFT calculations are finished, we get the current time *t1* then subtract it with *t0* to get the time elapsed of the IDFT calculations. Finally, we print out the time elapsed.

Not included in the code snippets, we added a *loop* variable, which loops the DFT and IDFT calculations a number of times. We also added the variable *total_time* and *average_time* to calculate for the average execution times of the DFT and IDFT functions.

D. The Matlab Built-In DFT & IDFT Functions

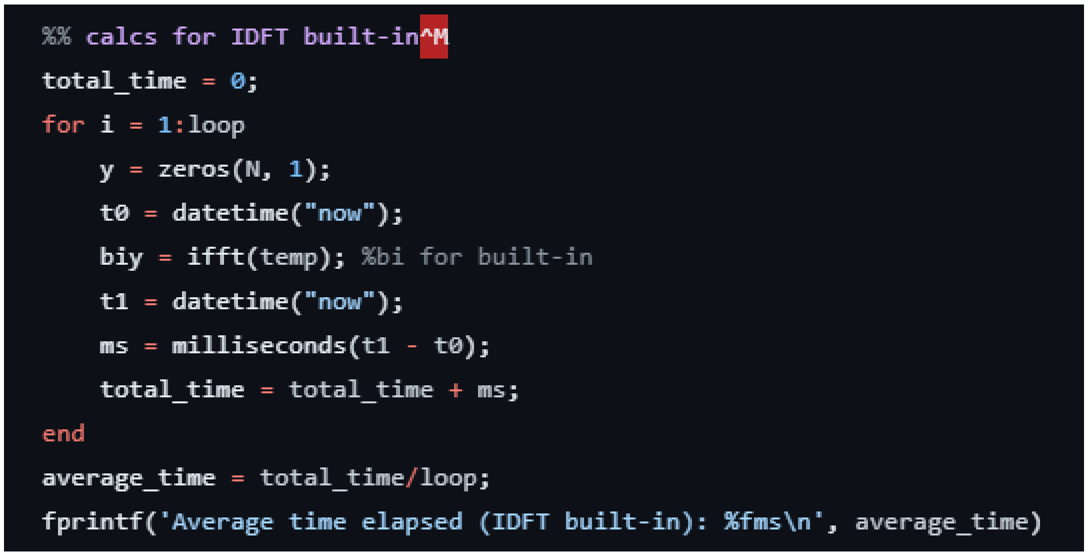
Other than our implementation of DFT and IDFT in CUDA, we found out that Matlab has built-in DFT and IDFT functions. For another test, we tried implementing these functions and compared it to the execution speeds of CUDA and our Matlab implementation. For the implementation of these functions, the initialized values are the same. *N* is still the number of elements, *x* is still the original array and is initialized through incrementing the loop index, and we still implement *loop*, *total_time*, *average_time*, *t0*, and *t1*. However, the DFT and IDFT logic is completely different.



```
%% calcs for DFT built-in^M
total_time = 0;
for i = 1:loop
    t0 = datetime("now");
    temp = fft(x);
    t1 = datetime("now");
    bixr = real(temp); %bi for built-in
    bixi = imag(temp); %bi for built-in
    ms = milliseconds(t1 - t0);
    total_time = total_time + ms;
end
average_time = total_time/loop;
```

Code Snippet #15: Built-in DFT Function Logic

Here, we simply use the *fft*() function to automatically compute the DFT of *x*. All we had to do was time the duration through *t0*, *t1*, *total_time*, and *average_time*. Other than that, for correctness checking, we split the output result, which is currently in *temp*, and stored the real component in *bixr* and the imaginary component in *bixi*.



```
%% calcs for IDFT built-in^M
total_time = 0;
for i = 1:loop
    y = zeros(N, 1);
    t0 = datetime("now");
    biy = ifft(temp); %bi for built-in
    t1 = datetime("now");
    ms = milliseconds(t1 - t0);
    total_time = total_time + ms;
end
average_time = total_time/loop;
fprintf('Average time elapsed (IDFT built-in): %fms\n', average_time)
```

Code Snippet #16: Built-in IDFT Function Logic

Then, for the built-in IDFT function, we simply use *ifft*() on *temp*, which automatically computes for the IDFT of *temp*.

E. Comparison of execution times (in ms, average of 5 loops):

For DFT:

DFT	**2^{10}**	**2^{12}**	**2^{14}**	**2^{16}**	**2^{17}**	**2^{20}**
Matlab	76.4536	1,006.3424	16,627.7718	272,753.2934	1,090,047.402	N/A
CUDA	17.8998	64.9558	244.1552	1,845.5358	5,528.1438	341,590.822
Matlab Built-in	0.344	0.4414	0.8766	3.3722	6.1582	18.9568

DFT	**NORMALIZE TO MATLAB**			**NORMALIZE TO CUDA**			**NORMALIZE TO MATLAB BUILT-IN**		
Execution time ratio (ETR)	Matlab	CUDA	Matlab built-in	Matlab	CUDA	Matlab built-in	Matlab	CUDA	Matlab built-in
ETR (2^{10})	1	0.23413	0.00450	4.27120	1	0.01922	222.24884	52.03430	1
ETR (2^{12})	1	0.06455	0.00044	15.49273	1	0.00680	2279.88763	147.15859	1
ETR (2^{14})	1	0.01468	0.00005	68.10329	1	0.00359	18968.48255	278.52521	1
ETR (2^{16})	1	0.00677	0.00001	147.79084	1	0.00183	80882.89348	547.27946	1
ETR (2^{17})	1	0.00507	0.00001	197.18145	1	0.00111	177007.47004	897.68825	1
GEOMETRIC MEAN	1	0.02379	0.00009	42.04089	1	0.00394	10659.25276	253.54492	1

For IDFT:

IDFT	2^{10}	2^{12}	2^{14}	2^{16}	2^{17}	2^{20}
Matlab	70.7116	1,005.0498	16,120.1196	269,506.2776	1,066,850.306	N/A
CUDA	17.743	60.4738	242.0722	1,881.8608	5,660.5748	347,683.428
Matlab Built-in	0.322	0.461	0.923	3.59	6.5218	15.8768

IDFT	**NORMALIZE TO MATLAB**			**NORMALIZE TO CUDA**			**NORMALIZE TO MATLAB BUILT-IN**		
Execution time ratio (ETR)	Matlab	CUDA	Matlab built-in	Matlab	CUDA	Matlab built-in	Matlab	CUDA	Matlab built-in
ETR (2^{10})	1	0.25092	0.00455	3.98532	1	0.01815	219.60124	55.10248	1
ETR (2^{12})	1	0.06017	0.00046	16.61959	1	0.00762	2180.15141	131.17961	1
ETR (2^{14})	1	0.01502	0.00006	66.59220	1	0.00381	17464.91831	262.26674	1
ETR (2^{16})	1	0.00698	0.00001	143.21265	1	0.00191	75071.38652	524.19521	1
ETR (2^{17})	1	0.00531	0.00001	188.47031	1	0.00115	163582.18682	867.94670	1
GEOMETRIC MEAN	1	0.024258	0.000099	41.223639	1	0.004100593	10053.09278	243.8672	1

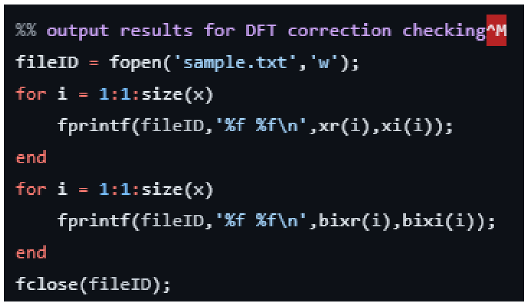
Before our analysis of execution times, we want to note that for the Matlab category, 2^{20} is left as N/A. This is because of the exponentially increasing execution times of the program. Trying to run 2^{20} elements using our Matlab implementation would take more than an hour per DFT or IDFT iteration. If we were to do an average execution time test of 5 loops, the test would take more than 10 hours to accomplish. Therefore, as a group, we decided to omit this portion of the testing, as the results are already clear enough. This also means that the correctness checking for 2^{20} for this implementation will not be included.

As we can see, the execution times of both DFT and IDFT in Matlab are much slower compared to the execution times of our implementation using CUDA. In particular, based on the computed geometric means for DFT, we can see that CUDA is 42.04 times faster than Matlab from 2^10 elements to 2^17 elements. Meanwhile, our geometric mean values for IDFT are quite similar to what we got for DFT since the result here is that the CUDA implementation is 41.22 times faster than Matlab from 2^10 elements to 2^17 elements. These results show how much better the performance of our CUDA implementation is compared to our Matlab implementation.

However, when it comes to comparing the execution times of our CUDA implementation and the Matlab built-in functions, we noticed that the Matlab built-in functions are much faster than CUDA, despite all the prefetching, memadvise, and page creations. This is because the Matlab DFT & IDFT built-in functions are fully optimized already for their purpose. In particular, we can see that based on the computed geometric means for DFT, Matlab's built-in implementation is around 253.54 times faster than our CUDA implementation, and is also 10659.25 times faster than our own Matlab implementation. The results are similar for IDFT since the computed geometric means show that Matlab's built-in implementation is 243.87 times faster than our CUDA implementation, and is also 10053.09 times faster than our own Matlab implementation. These results show that Matlab's built-in implementation of DFT and IDFT are incredibly optimized and are significantly faster than our implementations which already utilized parallelization.

F. Correctness Checking

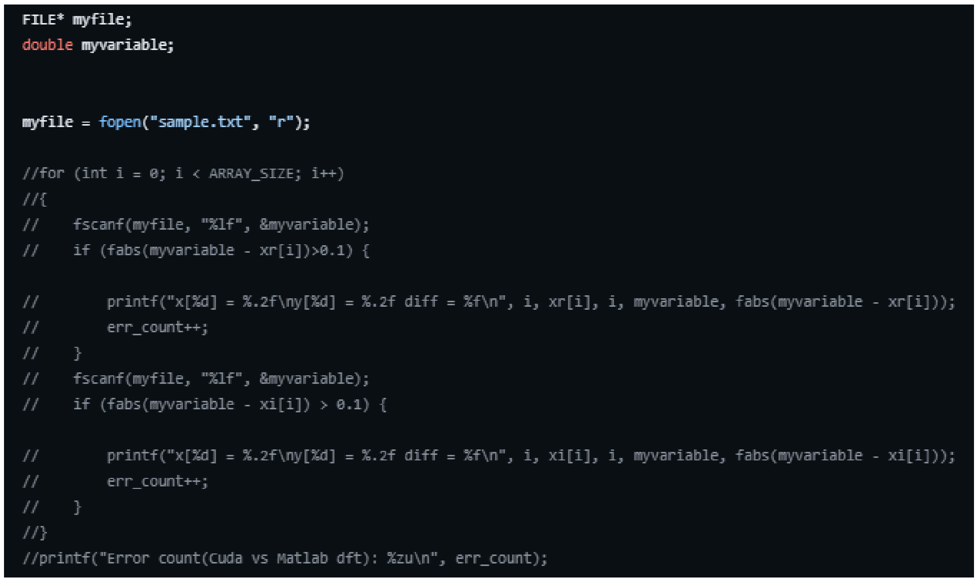
For correctness checking, we output the results from our Matlab implementation and Matlab built-in functions. The results are stored in text files, which is then compared to the results in CUDA. We error-checked CUDA vs our Matlab DFT implementation and CUDA vs Matlab built-in DFT function. We error-checked for 2^{10}, 2^{12}, 2^{14}, 2^{16}, $2^{17}$and 2^{20}. We also error-checked CUDA IDFT, where we compared x and y, as applying DFT then IDFT to an array/signal will result in the original array/signal. If the CUDA vs Matlab built-in DFT function results in 0 errors, then our implementation in CUDA is correct. If this criterion is met, if CUDA vs our Matlab DFT implementation also results in 0 errors, then our Matlab implementation of DFT is also correct.



```
%% output results for DFT correction checking^M
fileID = fopen('sample.txt','w');
for i = 1:1:size(x)
    fprintf(fileID,'%f %f\n',xr(i),xi(i));
end
for i = 1:1:size(x)
    fprintf(fileID,'%f %f\n',bixr(i),bixi(i));
end
fclose(fileID);
```

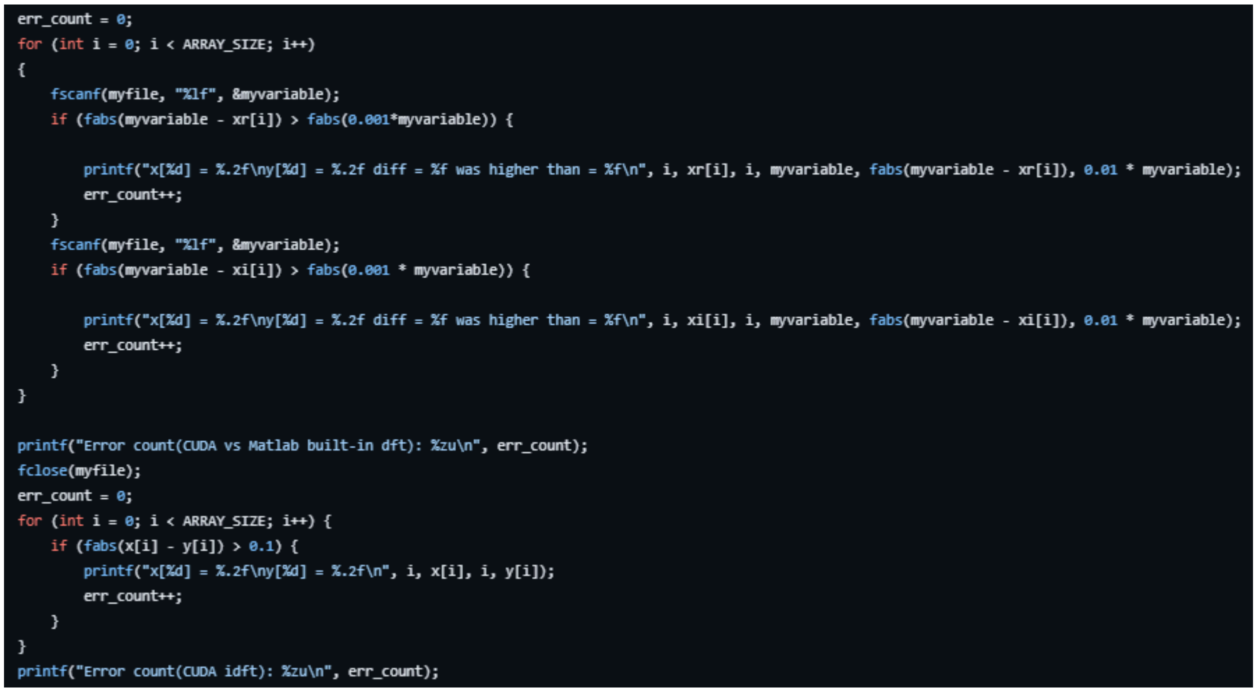
Code Snippet #17: File Writing for Error Checking

To start, we take the results from our DFT implementation in Matlab and the results from the built-in DFT function. We write the values in a file “sample.txt”.



```
FILE* myfile;
double myvariable;


myfile = fopen("sample.txt", "r");

//for (int i = 0; i < ARRAY_SIZE; i++)
//{
//    fscanf(myfile, "%lf", &myvariable);
//    if (fabs(myvariable - xr[i])>0.1) {

//        printf("x[%d] = %.2f\ny[%d] = %.2f diff = %f\n", i, xr[i], i, myvariable, fabs(myvariable - xr[i]));
//        err_count++;
//    }
//    fscanf(myfile, "%lf", &myvariable);
//    if (fabs(myvariable - xi[i]) > 0.1) {

//        printf("x[%d] = %.2f\ny[%d] = %.2f diff = %f\n", i, xi[i], i, myvariable, fabs(myvariable - xi[i]));
//        err_count++;
//    }
//}
//printf("Error count(Cuda vs Matlab dft): %zu\n", err_count);
```

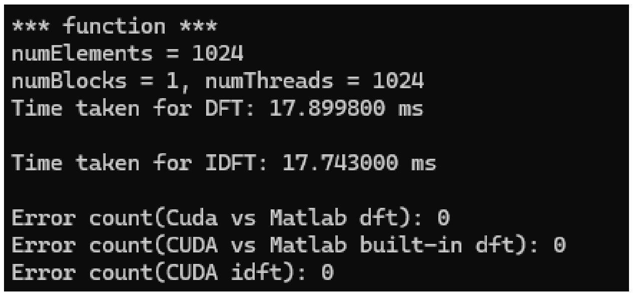
Code Snippet #17: File Reading & Error Checking for CUDA vs our Matlab Implementation (DFT)



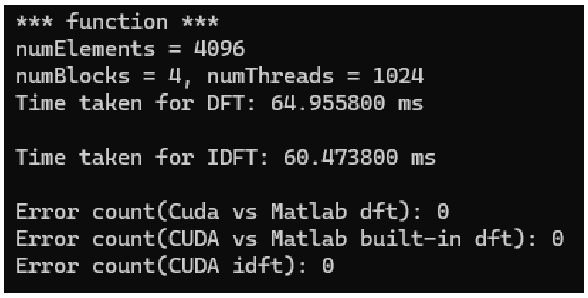
```
err_count = 0;
for (int i = 0; i < ARRAY_SIZE; i++)
{
    fscanf(myfile, "%lf", &myvariable);
    if (fabs(myvariable - xr[i]) > fabs(0.001*myvariable)) {

        printf("x[%d] = %.2f\ny[%d] = %.2f diff = %f was higher than = %f\n", i, xr[i], i, myvariable, fabs(myvariable - xr[i]), 0.01 * myvariable);
        err_count++;
    }
    fscanf(myfile, "%lf", &myvariable);
    if (fabs(myvariable - xi[i]) > fabs(0.001 * myvariable)) {

        printf("x[%d] = %.2f\ny[%d] = %.2f diff = %f was higher than = %f\n", i, xi[i], i, myvariable, fabs(myvariable - xi[i]), 0.01 * myvariable);
        err_count++;
    }
}

printf("Error count(CUDA vs Matlab built-in dft): %zu\n", err_count);
fclose(myfile);
err_count = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
    if (fabs(x[i] - y[i]) > 0.1) {
        printf("x[%d] = %.2f\ny[%d] = %.2f\n", i, x[i], i, y[i]);
        err_count++;
    }
}
printf("Error count(CUDA idft): %zu\n", err_count);
```

Code Snippet #18: More of the Error Checking Code

For how we implemented the error checking, first we take the "sample.txt" file and open it in CUDA. We scan the file and check each value in the DFT implementations. We then check each value inside the file and check if they match with the values in CUDA. If the difference between 2 values is more than 0.1%, then the error count will increase.

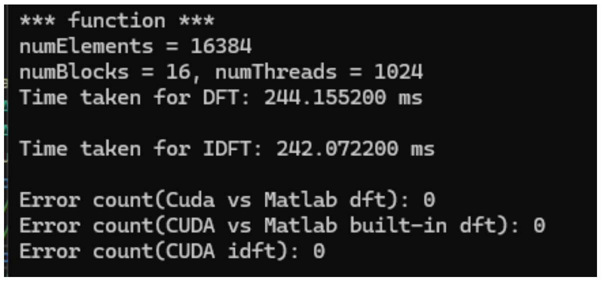


```
*** function ***
numElements = 1024
numBlocks = 1, numThreads = 1024
Time taken for DFT: 17.899800 ms

Time taken for IDFT: 17.743000 ms

Error count(Cuda vs Matlab dft): 0
Error count(CUDA vs Matlab built-in dft): 0
Error count(CUDA idft): 0
```

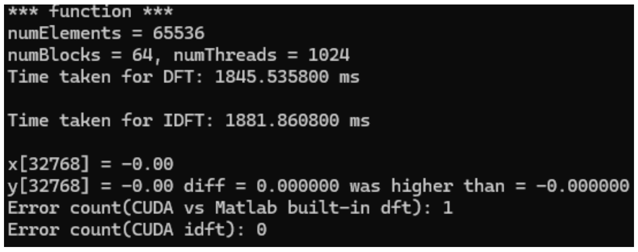


```
*** function ***
numElements = 4096
numBlocks = 4, numThreads = 1024
Time taken for DFT: 64.955800 ms

Time taken for IDFT: 60.473800 ms

Error count(Cuda vs Matlab dft): 0
Error count(CUDA vs Matlab built-in dft): 0
Error count(CUDA idft): 0
```

Figures 1 & 2: Error Checking at 2^{10} and 2^{12} elements

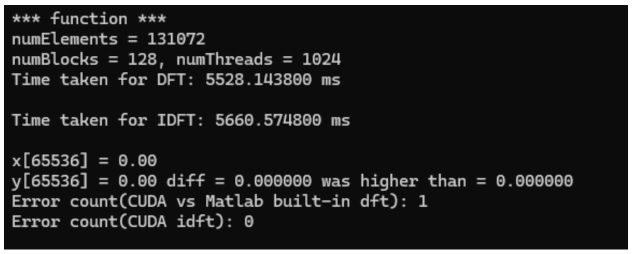


```
*** function ***
numElements = 16384
numBlocks = 16, numThreads = 1024
Time taken for DFT: 244.155200 ms

Time taken for IDFT: 242.072200 ms

Error count(Cuda vs Matlab dft): 0
Error count(CUDA vs Matlab built-in dft): 0
Error count(CUDA idft): 0
```

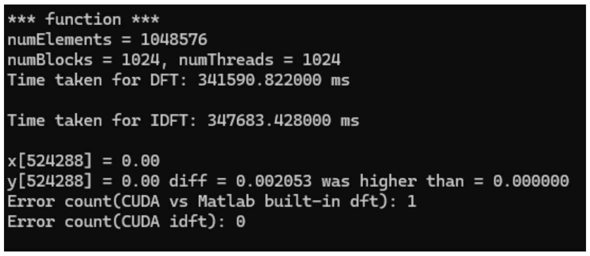


```
*** function ***
numElements = 65536
numBlocks = 64, numThreads = 1024
Time taken for DFT: 1845.535800 ms

Time taken for IDFT: 1881.860800 ms

x[32768] = -0.00
y[32768] = -0.00 diff = 0.000000 was higher than = -0.000000
Error count(CUDA vs Matlab built-in dft): 1
Error count(CUDA idft): 0
```

Figures 3 & 4: Error Checking at 2^{14} and 2^{16} elements



```
*** function ***
numElements = 131072
numBlocks = 128, numThreads = 1024
Time taken for DFT: 5528.143800 ms

Time taken for IDFT: 5660.574800 ms

x[65536] = 0.00
y[65536] = 0.00 diff = 0.000000 was higher than = 0.000000
Error count(CUDA vs Matlab built-in dft): 1
Error count(CUDA idft): 0
```



```
*** function ***
numElements = 1048576
numBlocks = 1024, numThreads = 1024
Time taken for DFT: 341590.822000 ms

Time taken for IDFT: 347683.428000 ms

x[524288] = 0.00
y[524288] = 0.00 diff = 0.002053 was higher than = 0.000000
Error count(CUDA vs Matlab built-in dft): 1
Error count(CUDA idft): 0
```

Figures 5 & 6: Error Checking at 2^{17} and 2^{20} elements

The comparisons between CUDA vs Matlab implementations have a 0.1% margin of error. From the figures above, the error checking for the execution times at 2^{10} to 2^{14} elements resulted in having 0 errors while the error checking for the execution times at 2^{16} to 2^{20} elements resulted in having 1 constant error in the built-in DFT portion. The output also displays the error that occurred wherein the difference between the CUDA and Matlab built-in DFT values surpassed the 0.1% threshold.

G. Conclusion

In conclusion, the execution times of our Matlab implementation of DFT & IDFT is the slowest, followed by CUDA, then the Matlab built-in functions. CUDA is faster than our Matlab implementations because of parallelization, as we parallelized the entire computation process for DFT and IDFT with 1024 threads, which meant that segments of the input signal were being extracted simultaneously, and were also being used in computations simultaneously. The simultaneous computations of each segment in the signal resulted in the much faster execution time of our implementation in CUDA compared to the execution time of our Matlab implementation, since Matlab performs our implemented algorithm's computations sequentially.

However, when we compare the speeds of CUDA and the Matlab built-in functions, we can see that the Matlab built-in DFT and IDFT functions are much faster. This is because these functions are already made to be optimized for their respective jobs. This shows that CUDA is not always the fastest in some scenarios. Additionally, when it comes to correctness checking, we can conclude that our implementations in CUDA and Matlab are accurate.