CAB401

Improving Performance of a Digital Music Analysis App
by Parallelizing its FFT Component in CUDA

Stephen Hannam n2731061

October 27, 2018

Abstract

The following report details the profiling and optimization of a Digital Music Analysis application written in C# provided from the CAB401 Blackboard, semester 2, 2018. It was readily apparent that the key bottle-neck of the application was a naive implementation of the Tukey-Cooley Recursive FFT algorithm at a fixed 2048 points of resolution. This FFT implementation was used inside a wrapper to stride through a wave files audio data to produce from it a spectrogram. This spectrogram was itself represented directly in the application, but also used further downstream for other decompositions, for instance, into musical notes.

The input and outputs of this FFT module were single precision, but all the operations taking place within it were double precision. To obtain a best sequential benchmark FFTW with a C# wrapper obtained from an open source github repository was used. Efforts were made to validate that the FFTW module was as simple as possible, and not using parallelization itself, but this could not be conclusively confirmed. It is known that the pinned memory version of a FFTW function was used.

The optimization undertaken here was to parallelize the entire spectrogram generation process in CUDA, and run it from a GPU which would be fed a single array of all the wave data. Additionally, the design calls for an asynchronous kernel launch at the very beginning of the application for the GPU to set-up certain essential arrays of data that do not rely upon the wave data specifically.

The parallelization achieved took full advantage of bit-wise operations by adhering to an FFT based on inputs of lengths of powers of 2. Further to this, it restricts itself more by preserving the hard-coded 2048 point nature of the transform. Though this potentially loses an advantage in generality (this advantage not existing anyway in the original application), it does allow this parallel implementation to take advantage of the fact that the GPU used can launch a maximum of 1024 threads per block. Though typically not recommended, issues pertaining to this were addressed, and the functional correctness of the parallelized implementation was verified. Though its precision is clearly different from that of the original application.

Contents

1 Introduction

A spectorgraphic representation of a signal is one that shows the frequency decomposition of the signal at any given moment in time; putting time on one axis and frequency on the other. The heaviest work-load was found to be in the production of this spectrographic result. Furthermore, at the heart of producing this result, is the Fast Fourier Transform.

NB: The desired spectrogram is for the magnitude (complex modulus) only. Phase information is discarded.

2 Identifying Bottlenecks

It was discovered that the FFT module and the wrapper around it were the primary points requiring parallelization by running the application in Visual Studio's (2017) CPU profiler. The full results of these are provided towards the end of the report.

2.1 Fast Fourier Transform

This report will not go into great detail about FFT's deeper theoretical underpinnings, such as orthogonality of sinusoidal basis functions, and continuous versus discretized forms. However, a brief overview of its theoretical basis will be outlined here.

The Fourier Transform assumes a time domain signal/function is comprised of a sum of variously scaled sinusoids at various frequencies and phase offsets (a linear combination of sinusoidal basis functions). Any of these parameters may undergo discretization and/or quantization when moving to a discrete representation but the underlying theory remains. The ability to represent any time domain function as a linear combination of sinusoidal basis functions is not actually complete, as argued by Lagrange and later proven by Gibbs, who showed that attempts to fully reconstruct ideal square waves in such a way will always result in a certain degree of discrepancy (overshoot) at the vertical edges; referred to as Gibbs overshoot [1].

It is still immensely useful however to approximate a time domain signal as if it were an ideal linear combination of sinusoidal basis functions. It is even often useful do this with signals which are not necessarily sinusoidal, or even periodic. In this application however it is performed so as to obtain a representation at any given point in time of the proportions of a set of frequencies present in an audio track; a physical domain that is necessarily frequency based. It can then compile these frequency decompositions along the time axis, to provide a spectrogram.

The Fast Fourier Transform was theorized at least a century before its modern realisation, which is commonly regarded as originating with Tukey and Cooley in 1965. It achieved an $N \log(N)$ algorithm for producing the decomposition in a discrete domain by factorizing a large matrix into a sparse, diagonal set of progressively smaller submatrices, giving a matrix transformation in which most multiplications were by 0, and so therefore computationally ignorable.

The important consequence of this is that the original Tukey-Cooley algorithm only operates on square matrices whose dimension is a power of 2. There are N-point FFT algorithms that do not require this, but they were not included in this parallelization, as shall to discussed further, FFT is not the final goal of this application, merely a means to an end.

Finally, Tukey-Cooleys algorithm comes in two forms. An iterative one and a recursive one. The existing FFT module in the application is the recursive variant.

The most important idea here is: These two variant are essentially two different approaches one may take to realise a basic permutation at the heart of the transform, called the Butterfly Permutation or Butterfly Diagram.

2.2 The Butterfly Permutation

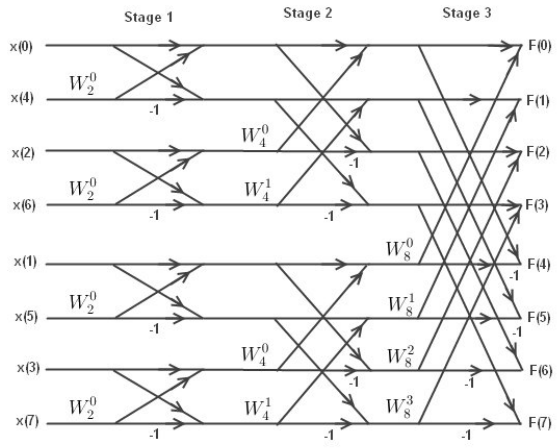

Figure 1: A fully expanded butterfly diagram for a 8-point FFT [2]

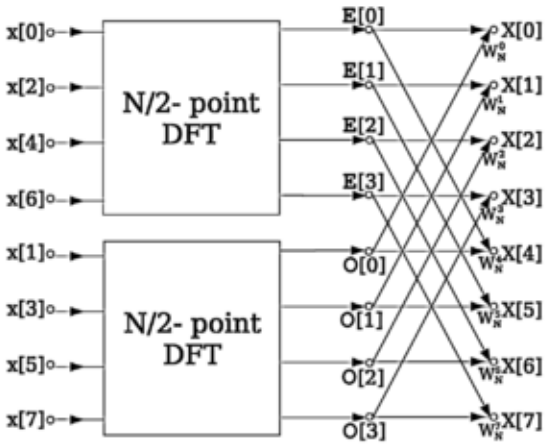

Figure 2: A general recursive form for a 2^n-point FFT butterfly diagram, comprised of lower recursive levels of itself [3]. E and O are Even and Odd.

Note the odd-even pattern of the swaps. The more general form can be understood from the second image on the right. Below are pseudo-code (cited) for both variants of the Tukey-Cooley Algorithm.

RECURSIVE FFT
max,avg,min n $log(n)$

recursiveFFT(A, toplevel = true)

1. n = A.length; // n is a power of 2
2. **if** (n is 1) **then return** A
3. **for** j = 0 **to** n/2 - 1 **do**
4. A0[j] = A[2 * j]
5. A1[j] = A[2 * j + 1]
6. Y0 = *recursiveFFT*(A0, false)
7. Y1 = *recursiveFFT*(A1, false)
8. $\omega = 1$; ω_n = ("Inverse") ? $e^{-2\pi i/n}$: $e^{2\pi i/n}$
9. **for** k = 0 **to** n/2 - 1 **do**
10. Y[k] = Y0[k] + ω * Y1[k]
11. Y[k + n/2] = Y0[k] - ω * Y1[k]
12. $\omega = \omega * \omega_n$
13. **if** ("Inverse" **and** toplevel) **then**
14. **for** j = 0 **to** n - 1 **do** A[j] = A[j] / n
15. **return** A

Figure 3: Tukey-Cooley Recursive [4]

ITERATIVE FFT
max,avg,min n $log(n)$

iterativeFFT(A, X)

1. n = X.length; p = lg(n)
2. **for** j = 0 **to** n - 1 **do**
3. X[j] = A[*reverseBits*(j)]
4. **for** s = 1 **to** p **do**
5. m = 0x1 << s
6. ω_m = ("Inverse") ? $e^{-2\pi i/m}$: $e^{2\pi i/m}$
7. **for** k = 0 **to** n - 1 **step** m **do**
8. $\omega = 1$
9. **for** j = 0 **to** m / 2 - 1 **do**
10. t = ω * X[k + j + m/2]; $\omega = \omega * \omega_m$
11. u = X[k + j]
12. X[k + j] = u + t
13. X[k + j + m/2] = u - t
14. **if** ("Inverse") **then**
15. **for** j = 0 **to** n - 1 **do** X[j] = X[j] / n

Figure 4: Tukey-Cooley Iterative [5]

The rationale for switching to the iterative algorithm for the parallelization is presented later.

Firstly, note the bit-reversal step. Take the index position (j) of each array element (A[j]), converting to binary, reverse the bits, giving the new index position of that element in an array (X) that will be operated upon. This is the step that achieves the Butterfly Permutation, in a single step. This permutation is accomplished progressively at each level of recursion in the recursive algorithm.

NB: line 6 of the Iterative Algorithm is confusing. It should be replaced with $\omega_m = e^{-2\pi i/m}$. Where i is the imaginary number.

Secondly, the complex rotation (ω) scanning multiplicatively in the j-loop resets back to unity every time after exiting. In this parallelization these values would be asynchronously pre-calculated while the app is reading in the wave data.

Thirdly, the term handled as t, referred to as a 'radius', and u referred to as a 'center', together describe radix-2 symmetric folds in the operative array (X) that transform each other through roots of unity. $(u \pm t)$.

Finally, in both cases the end result is normalized. In the above, by dividing by the number of elements in the array. But this could also be achieved by dividing by the maximum complex modulus, as is done in the original application being modified.

2.3 Recursive FFT and a Wrapper Around It

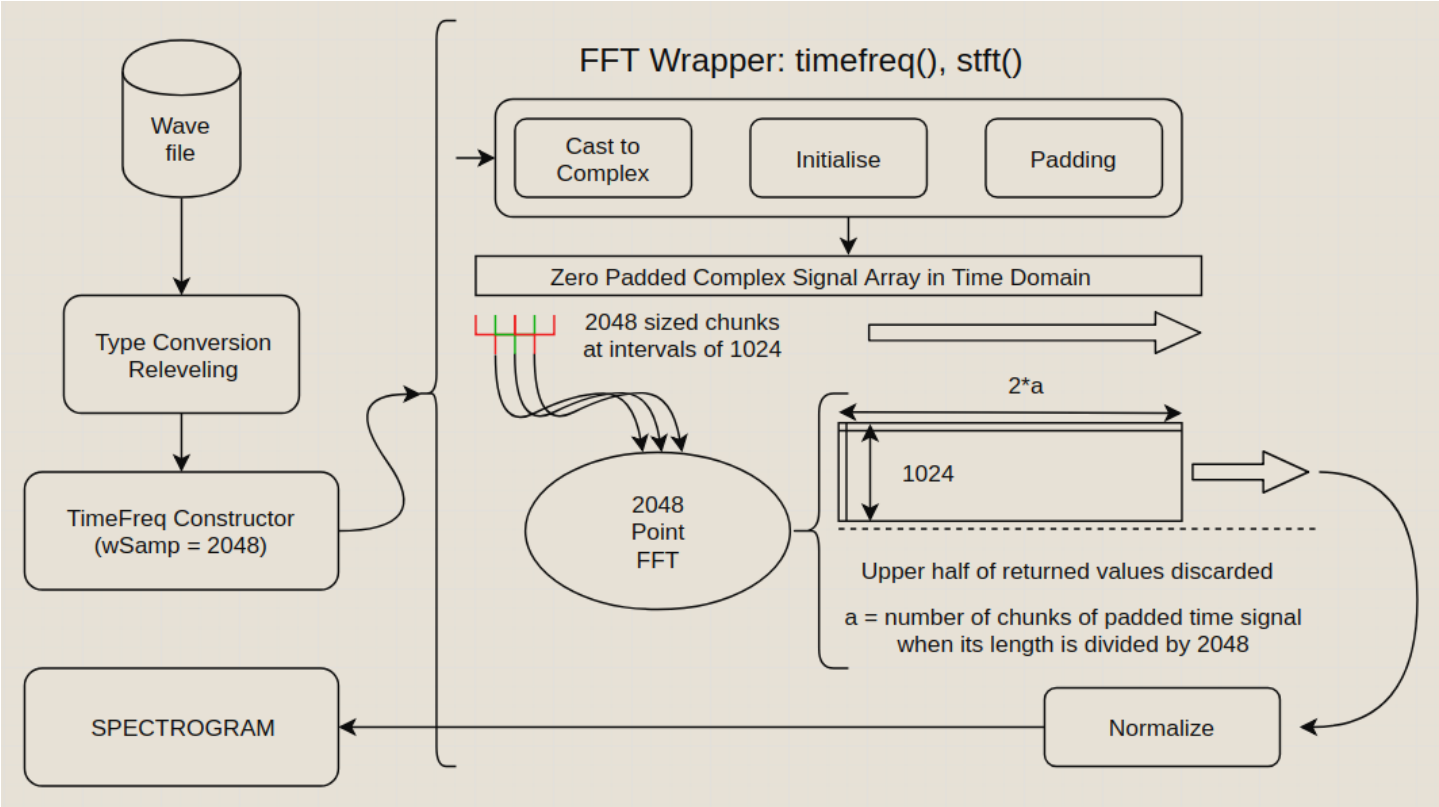

Figure 5: Graphical overview of how the application generates the spectrogram. Everything on the right side of the large single brace (FFT Wrapper) was reproduced in CUDA and run a GPU. NB: the FFT referred to here is a direct .NET implementation of ***Tukey-Cooley's recursive algorithm****. Referred to in descriptions of the core kernel.*

1. **Read in Wavefile**: in .NET the binaryReader is used. Header information is read off and parameters appropriately assigned.
2. **Type Conversion and Re-levelling**: the audio information is taken in byte-wise, as unsigned 8 bit numbers. These are then converted to floats ranging between -1 and 1 via re-levelling (subtract 128 from each value and then divide by 128).
3. **TimeFreq Constructor**: this is the constructor for the spectrogram object, 2048 is the width of FFT transform taken, and it is hard coded. The FFT was left at this hard-coded number for parallelization, and is optimized for it. This could be considered one of its limitations.
4. **Cast to Complex**: the time domain signal is cast to a Complex class. In .NET this class is a wrapper around two doubles holding the real and imaginary components, along with overloads for complex versions of standard arithmetic operations.
5. **Initialize and Padding**: the imaginary components of the now complex time domain signal are set to zero, and another set of all zero complex elements is padded onto the end of the array to ensure that its length is a multiple of 2048. This is because the recursive FFT implemented may only operate on chunks of signal that are a power of 2 in length.
6. **2048 Point FFT**: 2048 wide chunks of the padded complex time domain array are taken at intervals of 1024 and fed in to the FFT module. After every return from the FFT the maximum value of the output array's complex modulus values is recording for normalization at the end.
7. **Populating the Spectrogram Array and Normalization**: the spectrogram is a 2-D array. Its columns are populated 'vertically' with the lower half of the 2048 wide array output from the FFT, and due to the overlap in the chunks fed in, the second dimension of this array must be twice as long as the number of 2048 wide chunks that comprise the padded complex time domain array. All elements are then normalized by dividing by the maximum complex modulus evaluated in the previous stages.

2.4 FFTW("Fastest FT in the West") and cuFFT("CUDA FFT")

FFT has been polished and perfected and parallelized to a high level for decades now. Two notable mentions for open source solutions to speeding up FFT are FFTW and cuFFT. Under non-academic circumstances the only sane solution would be to use these and move on. However, FFT is where this application spends the majority of its time and it will be shown that there is not much scope for parallelization within this application outside of the (i) FFT module and (ii) the wrapper around it.

Thus, the not sane choice, is the sanest, and the FFT was taken as the primary focus for improving the performance of this application. The wrapper around the FFT was conflated with the parallelization of the core FFT module, as will be explained further in the report.

For the purposes of benchmarking against a 'best' sequential version, FFTW in a pinned memory configuration was used.

3 Evaluating Parallelisation Opportunities

As has been asserted, the primary goal of this application is not to perform generalized FFT. It is to decompose an audio stream into a frequency decomposition within typical musical scales. As such, power of 2 restricted FFT is maintained, and further than this, the hard-coded parameter of 2048 for the FFT resolution is also kept.

Through some cursory research, it would seem that much of the difficulty provided high performance solutions for FFT is concentrated in the area of high performance for generalized and flexible FFT requirements. Not being required here should allow for a high potential of optimization on the 2048 point Tukey-Cooley FFT.

4 Switching Algorithms for the FFT Modules

Both the recursive and iterative forms may be parallelized in CUDA, as CUDA 9.0 does provide for dynamic parallelism (recursive kernel launches). However, the limited capacity of the control logic available to a Streaming Multiprocessor, and the smaller number of registers available to individual threads, requires that data passed between levels of recursive be done through global memory. Something which should be minimized in GPU programming, in order to achieve high compute efficiency/occupancy.

It was due to this restriction on memory access between recursive levels that the iterative algorithm was chosen for a CUDA implementation.

5 Overview of Parallel Algorithm for 2048-DFT in CUDA

There are three kernels involved:

1. **calcTwiddles**: should be launched as early as possible from host application and run asynchronously, since the base twiddle values do not need any of the audio data in order to be calculated for a 2048 point DFT. The host should synchronize to the device prior to launching the core kernel.
2. **core**: this is the kernel that strides through the wave data, performing the FFT, and then storing the results into an output array which must be reshaped (in this report only host side reshaping was performed, but device side reshaping or 2-D arrays are possible) prior to compilation of the spectrogram. The output array is a flattened form of the 2-D spectrogram that is output by the FFT wrapper. **This core runs the outermost loop of the iterative algorithm (11 times), with the inner two loops being completely unrolled.**
3. **finalNormalization**: prior to copying device memory back to the host, a final launch of this kernel performs normalization by dividing by the maximum complex modulus, which was evaluated by calls to an atomic max finding function called at the end of every warp through the core kernel.

To reiterate the iterative algorithm in a more succinct description:

1. Butterfly Permutation on input array by Bit-reversal
2. Start s loop, from 1 to $p = \log_2(2048)$
3. Calculation of the base twiddle-factor (ω_m) at given s
4. Start k loop, from 0 to last input element using steps of size $m = 2^s$
5. Init twiddle (ω) to unity (1)
6. Start j loop, from 0 to $\frac{m}{2} - 1$
7. Find radius
8. Multiplicatively scan twiddle (ω) once with base twiddle-factor (ω_m)
9. READ first center @ index k + j, and READ second center @ k + j + $\frac{m}{2}$
10. WRITE first center to first center + radius
11. WRITE second center to second center + radius
12. after all loops exited, perform normalization

5.1 Bit-reversal

The first loop is easily unrolled provided one also has an efficient means of performing bit-reversal on the values of the array indices. The bit-reversal process is axiomatically free from the problem of collision. Whilst there are degenerate cases (when a binary form of a number is a palindrome), it is part of the normal bit-reversal process. In CUDA 9.0 there is an in-built function (**__brev()**) which does atomic, in-place bit-reversal.

However, the registers are 32-bit, and so the bit-reversal also assumes a 32 bit number, including the leading zeros. Thus, the results of this bit-reversal must be divided by some constant parameter to restore them to an appropriately scaled value (between 0 and 2047).

This could present a large penalty to performance for array sizes that are not powers of 2. However, happily, the array size is restricted to 2^{11}. Therefore, the necessary division can be performed by bit-shifting, which is afforded to the thread for free by the hardware in the form of bit-shifting. In this case, __brev() >> (32 − 11).

5.2 Early Asynchronous Launches to Prepare Twiddles

There are two good ways to parallelize the dependencies involved with the twiddle (ω). They could be scanned. Scanning is typically done for arithmetic addition and results in a prefix sum [6]. It can also be done for multiplication, and there is example code available on github for this.

However, the best way to use this multiplicative scan would actually be to perform it earlier and asynchronously, while the host application is busy preparing other data-sets (in particular, loading and conditioning the wave file data), so as to prepare an array of values that can be quickly accessed through GPU constant memory (which can occur at register speeds if all threads are accessing the same value).

Thus, the need to optimize the twiddle pre-calculating kernel is superseded by the hosts own bottle-neck back in the CPU. Therefore, a highly optimized scan was not deemed necessary.

Two areas herein where optimization would improve net performance would be:

1. To find a way to store the final calculated array to the device memory without first copying it back to the host. This would certainly be an almost trivial task for one well versed in CUDA, however, this author was scrambling to learn it well enough to get something working before the deadline.
2. To (as alluded to earlier) find a way to force all the threads in a warp to access the same constant memory location at the same time. They already do this for 11 elements stored an array containing integer offsets, but achieving this for the 2047 long array of needed twiddle values may not actually be possible. However, whether it is or not is not clear at this juncture.

5.3 Reducing Memory Transfer Overheads

One of the more surprising results from running the NVidia Visual Profiler, was to discover that the program was spending the vast majority of its time in memory allocation. **Not copying, but allocation.**

However, the upshot to this is that on Maxwell architecture devices and higher, using CUDA 7.5 and higher, cudaMalloc is nominally a non-blocking call from the host side. So cudaMallocs may be executed early on, while the host is busy performing other non-parallelizable tasks at the beginning.

6 A Note on Validation and Integration with the .NET Application

A final, turn-key solution, fully integrated with the existing application in its native C#, was attempted, and made some progress, was not not ready in time for this report. The primary means sought for achieving integration is via an open library available on github called managedCUDA. It is specific to .NET and C#. Some early standalone, prototype code using this library will be including in code submitted with this report. However, it is a large API, sparsely documented. The primary means of learning how to use it are examples and delving through source code.

To validate the CUDA implementation, the original application is interrupted at points, where it will write to a binary file, which the CUDA program reads, and then writes its output to another binary file, which is then read back in by the original application.

The CUDA program does not perfectly reproduce the results of the original application. This is probably due primarily to the CUDA program using single-precision complex numbers, whereas the original application uses double precision complex numbers. It is evident by inspection though that the results are comparable.

However, the original application does first receive the wave data in single precision form, and write its results to the spectrogram in single precision form. Time has not permitted the full validation of a double precision version of the CUDA program.

7 Explanation of the CUDA Implementation

Return to document location from whence reader came

7.1 Bank Conflicts

Older tutorials on CUDA programming recommended the use of float3 structures, where the third element would be unused so as to avoid bank conflicts. However, this is not a mature field, and more recent documentation revealed that for the hardware being used, some bank conflict detection and mitigation is already present, and padding should not be needed.

Without using padding, larger kernel executions were achieving occupancy rates consistently greater than 95%. Typically, the primary effect of bank conflicts are stalled warps [7], and a high occupancy rate is an indication of very few stalled warps. Therefore, padding was not used; though it is easily available by un-commenting a few lines of code.

7.2 Arithmetic Precision

The Maxwell architecture (that of the GPU used) provides atomic arithmetic operations for single precision numbers, but not double precision. Using double precision was attempted. It seemed to produce a less accurate result. A means to update a double maximum value atomically is not available, and time did not permit an extensive enough redesign. Instead casting to a float was used for finding maxima for normalization.

Screen-shots of functional validation of the CUDA implementations can be found in the appendices. The single precision CUDA produced a maximum value of ~ 24, the double precision ~ 48, but the original application returned ~ 200. Once again, time did not permit this discrepancy to be fully investigated.

Though switching to double precision did not incur an appreciable penalty to the occupancy of the kernels, it did **more than double their execution times.**

7.3 Structuring Programs for the Device Specifications

The device query on the GPU used may be found in the appendices. The most pertinent points are:

- **Maximum threads per block: 1024**
  As mentioned earlier, cumulatively the j and k loops iterate 1024 times for each s loop. This is quite fortuitous, given executing 1024 threads at once is possible with this GPU. This does conflict with general advice toward NOT using this maximum number; often 256 is recommended. This is due to two issues.

  (1) Registers available per thread, and (2) the schedulers are not as sophisticated as typical CPU schedulers. This first is addressed at the next itemized bullet-point. The second is overcome by only ever launching a kernel with some number of blocks that are a power of 2.

- **Maximum registers per block: 65535 (DWORD regs)**
  Using 1024 threads per block allows 63 registers per thread. As verified by the NVidia Visual Profiler tool, the calcTwiddles kernel uses 19 registers per thread, the core kernel uses 17, and the normaliseFinal kernel uses 12.

- **Total amount of CONSTANT memory: 65536 bytes**

  Total amount of SHARED memory per block: 49152 bytes
  Of CONSTANT memory, three arrays are allocated to hold a total of 2×2047 Complex numbers (at 8 bytes each) and 11 DWORD (4 byte) integer values, giving a total of 32796 bytes. Per block 16 KB of SHARED memory is used to store 2048 Complex numbers (per block), each requiring 8 bytes.

7.4 Analysing Loop Dependencies

It was observed that within the outermost loop, the total iterations of the k and j loops taken together, always summed to 1024. Or in the general case, to $2^{\log_2(N)-1}$. The GPU can launch a maximum of 1024 threads per block.

Though it is generally not recommended to use this maximum value (with 256 and 128 being common recommendations), it was also found that there are two principle problems with using the maximum number of threads per block, both of which were ameliorated in this implementation. This is further elaborated upon in Explanation of the CUDA Implementation.

A deeper analysis of the **index locations** being addressed in steps 9 through 11 of the succinct description was formalized via a MATLAB script; provided in the appendices.

S = 1		S = 2		S = 3		S = 4			S = 11	
K + J	K + J +M/2	K + J	K + J +M/2	K + J	K + J +M/2	K + J	K + J +M/2		K + J	K + J +M/2
0	1	0	2	0	4	0	8	...	0	1024
2	3	1	3	1	5	1	9	...	1	1025
4	5	4	6	2	6	2	10	...	2	1026
6	7	5	7	3	7	3	11	...	3	1027
8	9	8	10	8	12	4	12	...	4	1028
10	11	9	11	9	13	5	13	...	5	1029
12	13	12	14	10	14	6	14	...	6	1030
14	15	13	15	11	15	7	15	...	7	1031
16	17	16	18	16	20	16	24	...	8	1032
18	19	17	19	17	21	17	25	...	9	1033
20	21	20	22	18	22	18	26	...	10	1034
22	23	21	23	19	23	19	27	...	11	1035
24	25	24	26	24	28	20	28	...	12	1036
26	27	25	27	25	29	21	29	...	13	1037
28	29	28	30	26	30	22	30	...	14	1038
30	31	29	31	27	31	23	31	...	15	1039
32	33	32	34	32	36	32	40	...	16	1040
34	35	33	35	33	37	33	41	...	17	1041
36	37	36	38	34	38	34	42	...	18	1042
38	39	37	39	35	39	35	43	...	19	1043
40	41	40	42	40	44	36	44	...	20	1044
42	43	41	43	41	45	37	45	...	21	1045
44	45	44	46	42	46	38	46	...	22	1046
46	47	45	47	43	47	39	47	...	23	1047
48	49	48	50	48	52	48	56	...	24	1048
50	51	49	51	49	53	49	57	...	25	1049
52	53	52	54	50	54	50	58	...	26	1050
54	55	53	55	51	55	51	59	...	27	1051
56	57	56	58	56	60	52	60	...	28	1052
58	59	57	59	57	61	53	61	...	29	1053
60	61	60	62	58	62	54	62	...	30	1054
62	63	61	63	59	63	55	63	...	31	1055
...	...	...	...	...	...	...	...	...	...	...
2046	2047	2045	2047	2043	2047	2039	2047	...	1023	2047

Figure 6: An abridged overview of the MATLAB results. The expanding cone of symmetric folding becomes very clear. The zig-zag and colour filling (with unique colours) repeats to the end of columns. Referred to in descriptions of the calcTwiddles and core kernel.

Within a given S loop, there is no flow dependency between elements of the operative array. As shown in the succinct description, no reads to an index along a given row follow a write at the same position. Though writes do follow reads.

With 1024 concurrent threads, **efficiently mapping the expanding cone of symmetric folds to the index** being handled by a thread must be found. As shown later, it is once again, powers of 2 and bit-wise operations to the rescue. Though this is also a fundamental case of the aforementioned limitation of the **non-generality** of this solution.

There are flow dependencies involved in scanning the twiddle-factors, but those are asynchronously pre-calculated ahead of time; see calcTwiddles.

7.5 Memory Transfers and Workflow

Memory allocations and copies (and kernel launches) can be executed asynchronously provided a recent enough version of CUDA is being used with a GPU of a high enough compute capability (which is the case in this project).

Though an integrated solution, with CUDA modules interacting with the .NET framework of the original application was not successfully furnished in the time allowed, a good deal of progress was made towards this.

The managedCUDA library [8] was chosen amongst a number of considered possibilities (including CUDAfy and Alea CUDA available through NuGet in Visual Studio). It was chosen for its level of transparency. Its lack of documentation offset the advantage of being able to get into all the source code. This did not constitute a comprehensive trade study however, and so this was not an authoritatively informed choice.

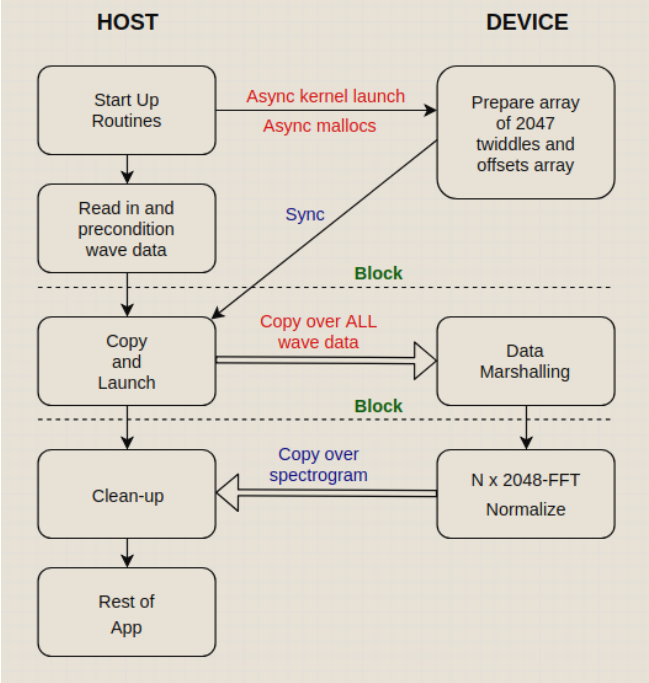

Figure 7: Proposed flow for optimized application.

7.6 Kernel Executions

7.6.1 calcTwiddles<<< 1, 11 >>>()

NB: presently the 'zerothTwiddles' (the 11 ω_m values) are calculated host-side. It would be trivial to make this a device-side task. The 'offsets' below (stored in CONSTANT memory) are $\{0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023\}$.

Recall that base-twiddles are given by $\omega_m = e^{-2\pi i/m}$ where $m = 2^s$ and the twiddles are scanned each loop as $\omega = \omega \times \omega_m$. Note the colours and the black zig-zag line in the loop analysis table. Twiddles reset back to the base-twiddle on every transition into a new colour, and scan down vertical black lines, which run the length of a single colour. The length of each scan is given by r^{k-1}, where $r = 2$ and k is the range of S. Using the formula for a geometric series $\sum_{k=1}^{n=11} ar^{k-1} = a\left(\frac{1-r^n}{1-r}\right)$ with $a = 1$, all scans of all twiddles will fill an array $\frac{1-2^{11}}{1-2} = 2047$ elements long. Results from development debugging in the appendices show the first rows of indices for both the operative array and the twiddle array being verified.

```
__global__ void
calcTwiddles(float * out_re, float * out_im, float * zerothTwiddles_re, float * zerothTwiddles_im){
  int tid = threadIdx.x;
  Complex runVal; runVal.x = 1; runVal.y = 0; //runVal.z = 0; <- bank conflict padding, not needed

  for (int i = offsets[tid]; i < 2 * offsets[tid] + 1; i++)
  {
    out_re[i] = runVal.x;
    out_im[i] = runVal.y;
    runVal = ComplexMul(runVal,ComplexCast(zerothTwiddles_re[tid],zerothTwiddles_im[tid]));
  }
}
```

A technically more efficient way to prepare the array of all twiddle factors could be to use a parallelized scan algorithm; a sequence of prefix-products. This was not deemed a productive area of optimization as this kernel should be launched asynchronously at the very start of the application; along with asynchronous memory allocations.

Finally, it may be beneficial to read wave-file header data earlier on in the host applications work-flow so as to provide the necessary parameters to issue all of the device-side memory allocation calls early on.

7.6.2 core<<< $blocksEachLaunch, threadsPerBlock, WSAMP * PREC$ >>>(), core<<< $blocksPerGrid \ \% \ blocksEachLaunch, threadsPerBlock, WSAMP * PREC$ >>>()

To prevent the scheduler from overflowing its counters, 'launch params 1' and 'launch params 2' are those appearing respectively within the triple chevrons of the launch kernels given in the above title. For smaller numbers of blocks in a launch (< 200), this is not an issue.

SHARED memory is dynamically allocated by the third term of the launch params. 'WSAMP' is 2048, 'HWSAMP' is 1024, and 'PREC' is a macro for the number of bytes contained in a single Complex number at a given level of precision; for single it is 8, and for double 16.

Below, 'blocksEachLaunch' is arbitrarily set at 32 but any power of 2 is fine. 1024 was used without issue.

```
int blocksEachLaunch = 32;
int iter = 0;
for(; iter < blocksPerGrid/blocksEachLaunch; iter++){
  core<<<... launch params 1 ...>>>(d_out, timefreq_num_els, iter, blocksEachLaunch);
}
if(blocksPerGrid % blocksEachLaunch > 0)
  core<<<... launch params 2 ...>>>(d_out, timefreq_num_els, iter, blocksEachLaunch);
cudaDeviceSynchronize();
```

The first requirement is to align the kernel within the entire array of wave data in global memory. Recall from the explication of the original sequential spectrogram generation that 2048 wide contiguous chunks at intervals of 1024 are taken, and each processed by the FFT module. The beginning of every such chunk is assigned to the 'start' variable; see the code snippet below.

Being that 1024 threads are invoked, and that two pointers within the 2048 element operative array are used, 'tidx0' and 'tidx1' are initialized for use in the Butterfly Permutation. These values are then bit-reversed atomically and bit-shifted back by IDX_SHIFT $= 32 - 11$, to return the values to a valid range. These values are then offset by 'start', and are now ready to be used in the Butterfly Permutation.

```
__global__ void
core(float * out, int size, int gridNum, int gridWidth){
  extern __shared__ Complex signalChunk[]; // twice the size of 1024 threads in block

  int tidx0, tidx1, rtidx0, rtidx1, stridx0, stridx1, idx_twid;
  Complex radius;
  // start of the block, threadIdx.x == 0
  int start = (gridNum * gridWidth + blockIdx.x) * blockDim.x;

  tidx0 = threadIdx.x;
  tidx1 = threadIdx.x + HWSAMP;
  rtidx0 = __brev(tidx0) >> IDX_SHIFT;
  rtidx1 = __brev(tidx1) >> IDX_SHIFT;
  stridx0 = rtidx0 + start;
  stridx1 = rtidx1 + start;
```

A check (arguably perfunctory) is used to ensure no addresses outside of the blocks range will be affected. The Butterfly Permutation is then performed in the same step that the blocks SHARED memory is populated with the data upon which the FFT will act. All threads must synchronize before proceeding for obvious reasons.

```
  if(tidx0 + start < size){

    signalChunk[tidx0] = ComplexCast(in_re[stridx0], in_im[stridx0]);
    signalChunk[tidx1] = ComplexCast(in_re[stridx1], in_im[stridx1]);
    __syncthreads();

    // 11 values of s as per iterative algorithm for fft (2048 point)
    for (unsigned int s = 0; s < LGSAMP; s++)
    {
```

The outer loop ('s'; which due to flow dependencies may not be parallelized) starts in the last line of the last code snippet; LGSAMP = 11. Below, 'tidx0', 'tidx1' and 'idx_twid' are reassigned values based on which iteration of the outer loop they are in; 'idx_twid' pulls precalculated twiddle factors from the array in CONSTANT memory.

Recall the index positions from the loop analysis formed an expanding cone of symmetric folds. It is clear 'tidx1' (mapping to $J + K + M/2$) will always be a distance 'offsets[s] + 1' from 'tidx0' (mapping to $J + K$).

Given that in CUDA the threads know their own index number (0 to 1023), the $J + K$ value for any given iteration of 's' can be calculated by masking off the negation of the 'offsets' and then adding the result to itself. Another view of this, is that the $J + K$ values can be seen to take the value of their own row position in the loop analysis table plus the modulus of that value with the negation of an offset value which correspondingly appears on the other side of the fold in $K + J + M/2$ for a given 's'; giving 'tidx1'.

```
25    tidx0 = threadIdx.x + (threadIdx.x & (~offsets[s]));
26    tidx1 = tidx0 + offsets[s] + 1;
27    idx_twid = offsets[s] + (threadIdx.x & offsets[s]);
28
29    // A[j + k + m/2]*w
30    radius = ComplexMul(signalChunk[tidx1],ComplexCast(twiddles_re[idx_twid],twiddles_im[idx_twid]));
31    // A[j + k]         = A[j + k] + A[j + k + m/2]*w  ==  center + radius
32    signalChunk[tidx0] = ComplexAdd(signalChunk[tidx0],radius);
33    // A[j + k + m/2] = A[j + k] - A[j + k + m/2]*w  ==  center - radius
34    signalChunk[tidx1] = ComplexSub(signalChunk[tidx0],radius);
35    __syncthreads();
```

Following this, due to flow dependency in loop 's', all threads must synchronize at the end of the loop.

Now that the heavy lifting is complete, 'tidx0' is reassigned to store to the 'out' array (of the same length as the entire array of wave data in global memory), the complex modulus of a corresponding value of the FFTs result from the first half of the SHARED memory 'signalChunk'. Recall from the explication of the original sequential spectrogram generation, that the upper half of the FFTs entire output is discarded.

```
36      }
37      tidx0 = threadIdx.x;
38      out[tidx0 + start] = ComplexNorm(signalChunk[tidx0]);
39
40      // eval for maxFFT, use AtomicMax() here
41      AtomicMax(&maxFFT, out[tidx0 + start]);
42    }
43  }
```

Finally, a function was found on github [9], which performs atomic evaluation and reassignment of a variable for updating maximum values in parallel.

7.6.3 **normaliseFinal**$<<<$ *blocksEachLaunch*, *threadsPerBlock* $>>>$(), **normaliseFinal**$<<<$ *blocksPerGrid* % *blocksEachLaunch*, *threadsPerBlock* $>>>$()

A final series of launches on 'normaliseFinal' are made in the same pattern as 'core'. Given the previous, the mechanics for this kernel should be apparent.

```
1  __global__ void
2  normaliseFinal(float * out, int size, int gridNum, int gridWidth){
3
4    int tidx = (gridNum * gridWidth + blockIdx.x) * blockDim.x + threadIdx.x;
5    if(tidx < size){
6      out[tidx] = out[tidx] / maxFFT;
7    }
8  }
```

7.7 Device Helper Functions

Below are the function signatures for the helper functions used in the above kernels.

```
//typedef float3 Complex; // padding to avoid bank conflicts - might not be needed in CUDA 9.0
typedef float2 Complex;
static __device__ void AtomicMax(float * const address, float value);
static __device__  inline Complex ComplexCast(float a, float b);
static __device__  inline Complex ComplexAdd(Complex a, Complex b);
static __device__  inline Complex ComplexSub(Complex a, Complex b);
static __device__  inline Complex ComplexMul(Complex a, Complex b);
static __device__ inline float ComplexNorm(Complex a); // sqrt(real^2 + imag^2)
```

8 Environment and Hardware Specifications

By far the most important piece of hardware to understand here is the GPU. Though PCI bus rates, bandwidths and pinned memory capabilities between the host CPU and GPU device affect the performance, GPU programming focuses on minimizing instances of memory transfer to reduce overhead and realize performance increase. The GPU the parallelized algorithm was tested on was queried with a CUDA program provided by NVidia with CUDA 9.0, in the sample folder (deviceQuery). A screen-shot of the full output from the query can be found in the appendices.

The device is an NVidia GeForce 940MX (on a Lenovo IdeaPad 510), running on a Maxwell Architecture with a compute capability of 5.0, 2GB of RAM, a maximum GPU clock of 1.24 GHz, and a maximum memory clock of 900 MHz. CUDA 9.0 was used.

The CPU on which the best sequential version was on the same machine, sporting an Intel Core i7-7500U (Kaby Lake) running at a maximum of 2.9 GHz and a typical of 2.7 GHz, with 8 GB of RAM, a 64-bit Windows 10 Pro OS.

The CUDA code was compiled on the Windows command line using:

“nvcc -gencode arch=compute_50,code=sm_50 -o kernel kernel.cu”

All other code was built and run from within Visual Studio 2017 Community Edition.

9 Profiling and Speed-Up Curve

All metrics in this section relate to performance of modules acting to produce a spectrogram for all the audio data in the file Jupiter.wav.

9.1 The Original Application

For the CPU based implementation using the unoptimized C# implementation of Tukey-Cooley’s recursive FFT algorithm, a typical time was 1636 ms. The profiling tool used was the standard performance profiler in Visual Studio 2017 Community Edition. See the appendices for some screenshot examples of this profiling.

9.2 The Application using FFTW

Using an FFTW dll, and C# wrappers found on github [10], the application was seen to perform this task in 159 ms. At this point however it can not be conclusively verified whether the FFTW call used was itself utilizing some form of parallelization. This is very possible, and so therefore should not be regarded as a benchmark for the purposes of asserting the highest performing sequential version of the code. See the appendices for some example screenshots of the profiler applied to the FFTW optimized application, and code snippets used to insert it.

9.3 The GPU Performance Profile

The best time for the GPU was 7.0435 ms, and the worst was 29.689 ms. Considerably better than the FFTW, even if the FFTW module were applying some form of parallelization.

For the core kernel, the warp execution efficiency reported by the profiler was always 99.9%, and the warp non-predicated execution efficiency always 99.6%. Example screenshots of the NVidia Visual Profiler being used, and a curve that is not an Amdahl curve, but definitely an homage to one, in the appendices.

10 Discussion of the Process

The narrative of this project, is that I left it a little too late, and then on a whim picked the Digital Music version (after toying with the idea of parallelizing a Huffman Encoder, a Viterbi Encoder or a Canny Edge Detector). Having actually completed an online MIT OCW course called Intro Linear Algebra run by Gilbert Strang in which I actually learned a great deal about the math behind FFT, I could appreciate that FFT is...well, linear algebra. And GPUs are really good at...linear algebra.

I then very quickly learned the first important thing about CUDA. It's really f**king hard! I was immensely relieved when I finally got a valid result about three days before the assignment was due. I'd love to regale the marker with more, but honestly it's a blur, and I have so many other things to attend to.

Having said that, I am super glad I chose to expose myself to it. In CAB403 I got a grip on POSIX Threads, I've completed a couple of good sized projects with FPGAs (after my interest was piqued by completing part 1 of Nand2Tetris [11] on Coursera, something I implore all serious students of computer science/engineering to do), and now I've had a taste of GPU programming. I like having that breadth of experience.

11 Conclusion and Reflection

Over the original unoptimized application, the best speedup obtained on the GPU may be put in the vicinity of 23,000%. Compared to FFTW, this speedup may be put in the vicinity of 1000%.

There is one fly in the ointment so to speak. Impressive as these numbers are, the GPU version did not properly reproduce the original spectrogram output. From the comparison shown in Validating the CUDA Outputs: Screenshots, the algorithm on the GPU does work. This result does validate the correctness of the parallelized algorithm in its basic underlying structure, but NOT in its precision and/or resolution.

Whether or not this would also impair the applications ability to correctly decompose the signal into musical notes (as it does in offsetDetection()) was not able to be investigated in the time allowed. However, before this parallelized solution could be deployed, either the output of the CUDA program would need to better reproduce the original spectrogram, or the currently achieved precision/resolution would need to be properly validated for all of the downstream functionality in the application that depend on it.

References

[1] E. W. Weisstein. (2018) Gibbs phenomenon. [Online]. Available: http://mathworld.wolfram.com/GibbsPhenomenon.html

[2] various. (2014) An 8 input butterfly. [Online]. Available: http://www.alwayslearn.com/DFT%20and%20FFT%20Tutorial/DFTandFFT_FFT_Butterfly_8_Input.html

[3] ——. (2017) Butterfly diagram. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/c/cb/DIT-FFT-butterfly.png/300px-DIT-FFT-butterfly.png

[4] E. of Nothing. (2018) Recursive fft. [Online]. Available: https://equilibriumofnothing.files.wordpress.com/2013/10/matrix_recursivefft.png

[5] ——. (2018) Iterative fft. [Online]. Available: https://equilibriumofnothing.files.wordpress.com/2013/10/matrix_iterativefft.png

[6] D. Walker. (2013) Parallel scans and prefix sums. [Online]. Available: http://www.cs.princeton.edu/courses/archive/fall13/cos326/lec/23-parallel-scan.pdf

[7] N. documentation. (2015) Issue efficiency. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm#IssueStallReasons

[8] M. Kunz. (2018) managedcuda. [Online]. Available: https://github.com/kunzmi/managedCuda

[9] N. D. Blog. (2015) Parallel forall. [Online]. Available: https://github.com/parallel-forall/code-samples/blob/master/posts/cuda-aware-mpi-example/src/Device.cu

[10] T. Szalay. (2017) Fftwsharp. [Online]. Available: https://github.com/tszalay/FFTWSharp

[11] N. Nisan and S. Schocken. (2018) nand2tetris. [Online]. Available: https://www.nand2tetris.org

Appendix A Example Screenshots of Profiling Unoptimized Application

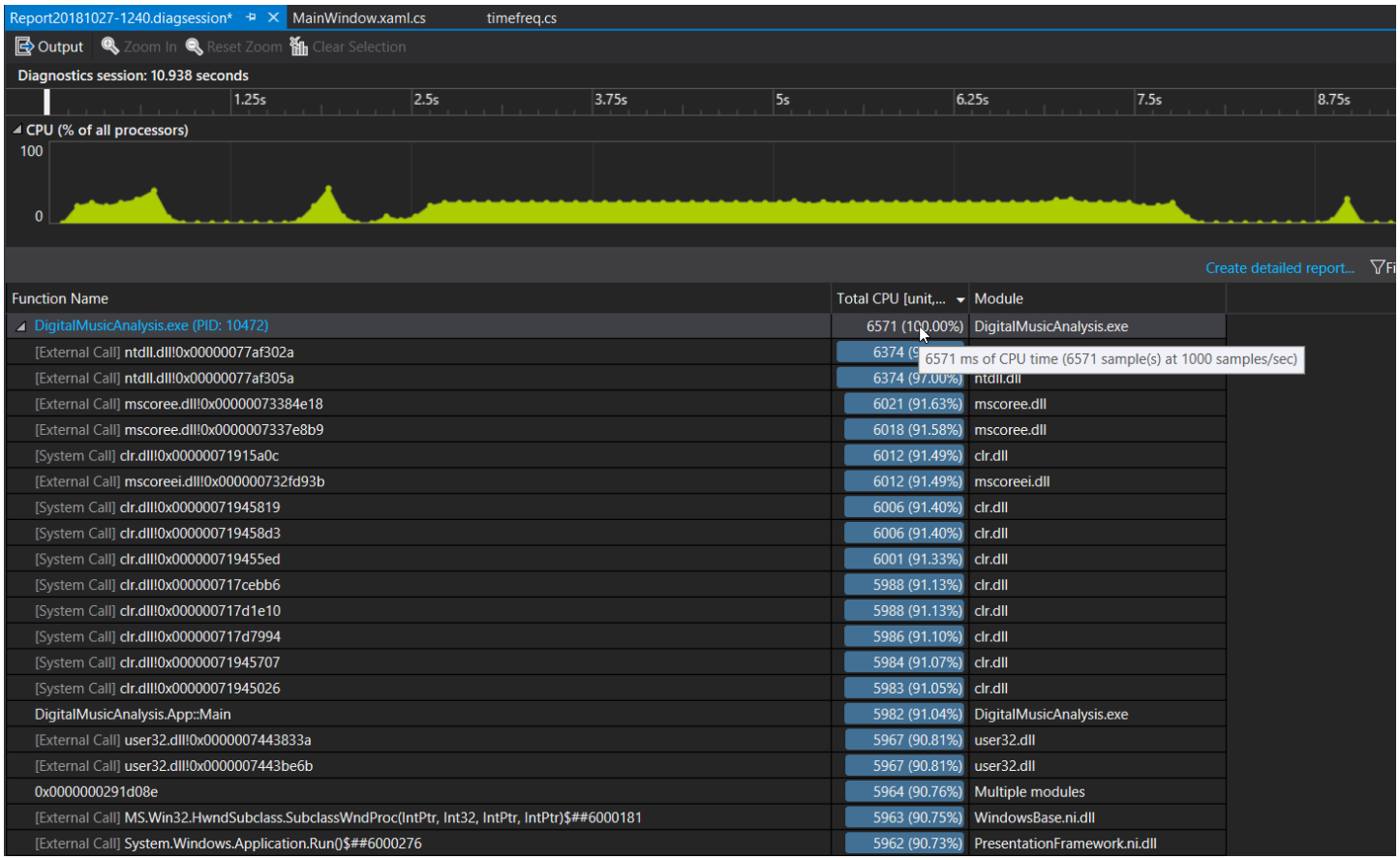

Figure 8: .

DigitalMusicAnalysis.MainWindow::onsetDetection	2580 (39.26%)	DigitalMusicAnalysis.exe
DigitalMusicAnalysis.MainWindow::freqDomain	2529 (38.49%)	DigitalMusicAnalysis.exe
DigitalMusicAnalysis.timefreq::.ctor	2508 (38.17%)	DigitalMusicAnalysis.exe
DigitalMusicAnalysis.timefreq::stft	2480 (37.74%)	DigitalMusicAnalysis.exe
DigitalMusicAnalysis.timefreq::fft	2335 (35.53%)	DigitalMusicAnalysis.exe
DigitalMusicAnalysis.MainWindow::fft	1634 (24.87%)	DigitalMusicAnalysis.exe

Figure 9: Total CPU time spent by percentage in these functions and their calls.

Appendix B Example Screenshots of Profiling FFTW Optimized Application

Figure 10: 159 ms of Total CPU time spent in call to FFTW.

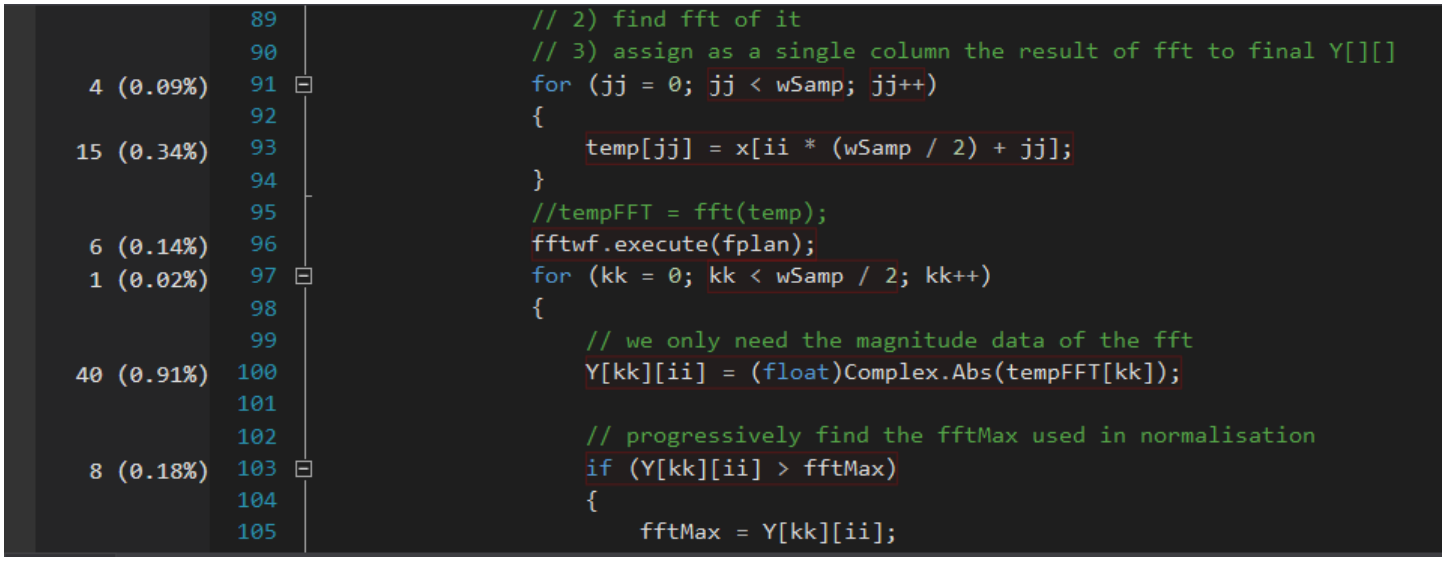

Figure 11: Profiler showing greatly reduced time spent calculating the FFT.

Code modification to timefreq.cs used to substitute in FFTW. Note: depends on dlls for FFTW and a library for C# based FFTW wrappers, found on github [10].

```
using System.Runtime.InteropServices;
using FFTWSharp;

namespace DigitalMusicAnalysis
{
  public class timefreq
  {
    ...
    float[][] stft(Complex[] x, int wSamp)
    {
      ...
      IntPtr fplan;
      GCHandle hdin, hdout;
      hdin = GCHandle.Alloc(temp, GCHandleType.Pinned);
      hdout = GCHandle.Alloc(tempFFT, GCHandleType.Pinned);
      fplan = fftw.dft_1d(wSamp, hdin.AddrOfPinnedObject(), hdout.AddrOfPinnedObject(), ...
    fftw_direction.Forward, fftw_flags.Measure);
      ...
      //tempFFT = fft(temp);
      fftwf.execute(fplan);
      ...
      fftwf.destroy_plan(fplan);
      hdin.Free();
      hdout.Free();
      return Y;
    }
  }
}
```

Appendix C Validating the CUDA Outputs: Screenshots

Return to document location from whence reader came

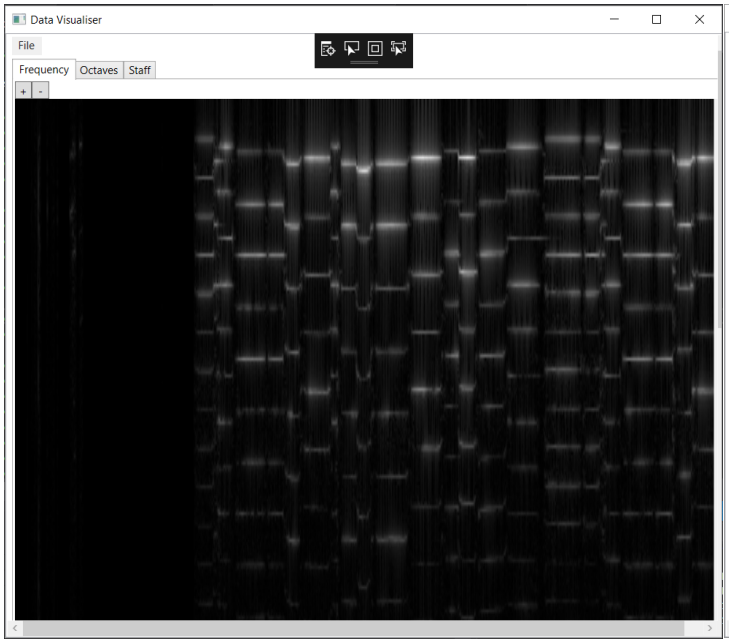

Figure 12: Original spectrogram output.

Figure 13: Single precision CUDA output.

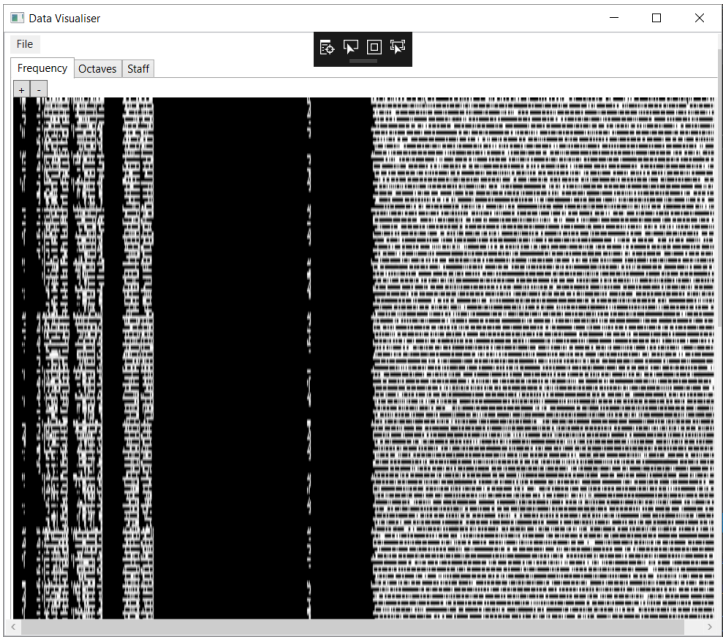

Figure 14: Double precision CUDA output. Possibly invalid due to inability to properly normalize it without further work on atomically updating double precision maximum values, or abrogating the need to do so atomically.

Appendix D Validating the CUDA Outputs: Code Snippets

The application reads in the audio data, relevels it and stores it as an array of floats. This array of floats (data[]) is also then written to a binary file which will feed the CUDA program. The CUDA result was fed back in via the modifications shown in the second segment of code.

```
namespace DigitalMusicAnalysis
{
  public class wavefile
  {
    ...
    public wavefile(FileStream file)
    {
      ...
      string filename = @"C:\CAB401data\wave_out";
      FileStream outFile = File.Create(filename);
      BinaryWriter binWrite = new BinaryWriter(outFile);

      ... original code reads wave header
      ... determines necessary params, then stores to data[] array

      for (int i = 0; i < numSamples; i++)
      {
          wave[i] = ((float)data[i] - 128) / 128; // relevel and store

          binWrite.Write(wave[i]); // catching in binary file, send to CUDA program
      }

      binWrite.Flush();
      binWrite.Close();
      binWrite.Dispose();

    }
  }
}
```

```
namespace DigitalMusicAnalysis
{
  public partial class MainWindow : Window
  {
    ...
    private void freqDomain()
    {
      ...
      stftRep = new timefreq(waveIn.wave, 2048); // downstream modules will fail without this

      int wSamp = 2048;
      filename = @"C:\CAB401data\timefreq_done";
      FileStream file = new FileStream(filename, FileMode.Open, FileAccess.Read);
      BinaryReader binRead = new BinaryReader(file);
      long length = new System.IO.FileInfo(filename).Length;
      float[] data = new float[length / 4];
      for (int idx = 0; idx < length / 4; idx++)
          data[idx] = binRead.ReadSingle();
      int A = wSamp / 2;
      int B = (int)(length / (4 * A));
      /* for timefreq_done */
      pixelArray = new float[A * B];
      for (int ii = 0; ii < A; ii++){
          for (int jj = 0; jj < B; jj++)
              pixelArray[ii * B + jj] = data[jj * A + ii]; // transpose
      }

    /* what the application was using for pixelArray data */
    //pixelArray = new float[stftRep.timeFreqData[0].Length * stftRep.wSamp / 2];
    //for (int jj = 0; jj < stftRep.wSamp / 2; jj++){
    //    for (int ii = 0; ii < stftRep.timeFreqData[0].Length; ii++)
    //        pixelArray[jj * stftRep.timeFreqData[0].Length + ii] = stftRep.timeFreqData[jj][ii];
    //
    //}
    }
  }
}
```

Appendix E Example of NVidia Visual Profiler Use

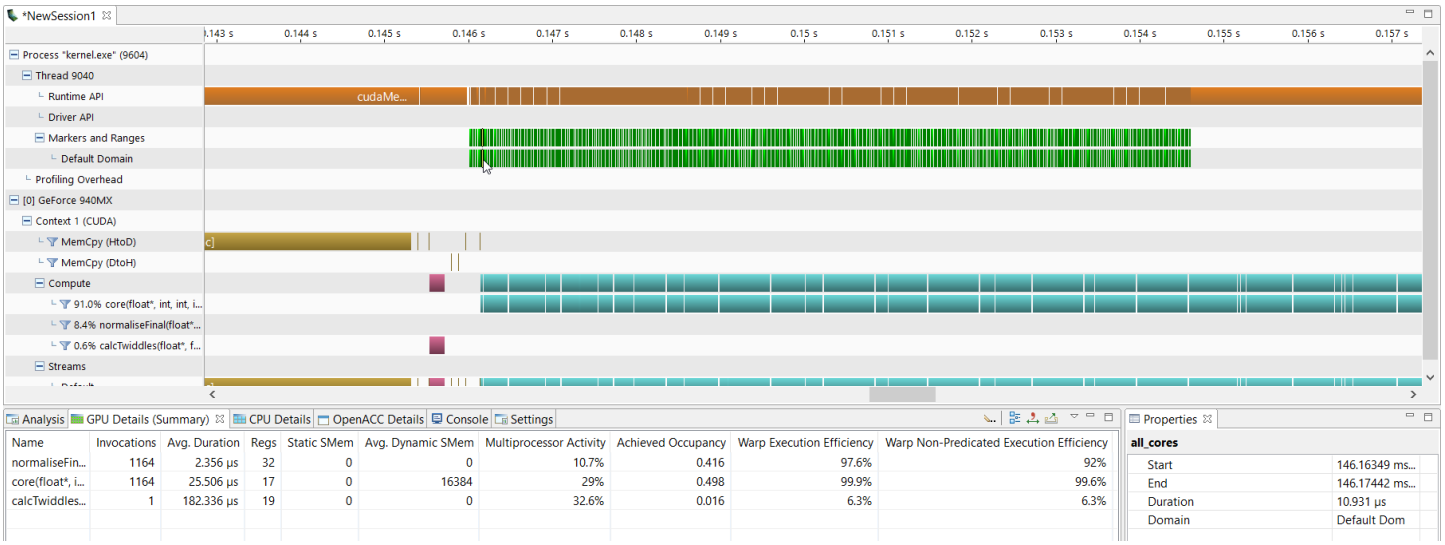

Figure 15: The metrics used: Avg Duration, Multiprocessor Activity, Achieved Occupancy, Warp Execution Efficiency.

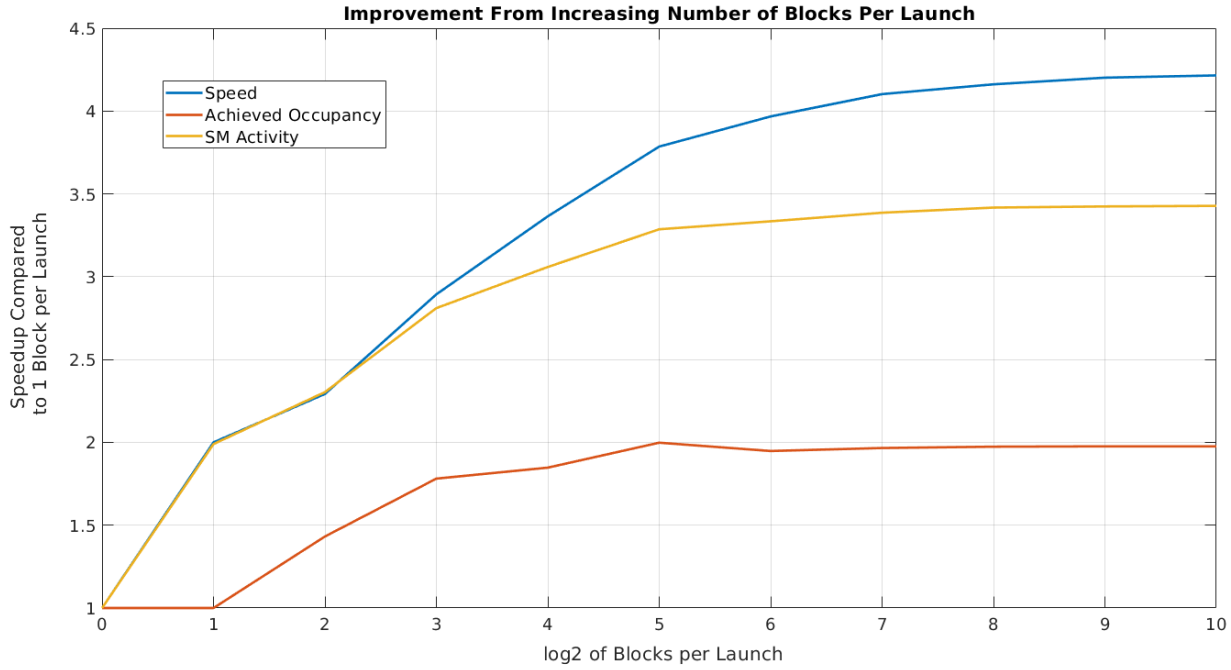

Figure 16: Not quite an Amdahl Curve. NB: 1024 Threads per block is a necessarily fixed quantity for this particular design. This is a limitation, in so far as it is non-general.

Appendix F Results of deviceQuery

Return to document location from whence reader came

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce 940MX"
  CUDA Driver Version / Runtime Version          9.0 / 9.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1242 MHz (1.24 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe (process 11916) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 17: Specifications for the GPU used in this project.

Appendix G Results of Testing and Verifying CUDA Index Masking

Return to document location from whence reader came

```
C:\Users\kostoglotov\Documents\CAB401>nvcc -gencode arch=compute_50,code=sm_50 -o test_idx test_idx_masking.cu
test_idx_masking.cu
   Creating library test_idx.lib and object test_idx.exp

C:\Users\kostoglotov\Documents\CAB401>test_idx
0    1    0    0    2    1    0    4    3    0    8    7    0    16   15   0    32   31   0    64   63   0    128  127  0    256  255  0    512  511  0    1024 1023
2    3    0    1    3    2    1    5    4    1    9    8    1    17   16   1    33   32   1    65   64   1    129  128  1    257  256  1    513  512  1    1025 1024
4    5    0    4    6    1    2    6    5    2    10   9    2    18   17   2    34   33   2    66   65   2    130  129  2    258  257  2    514  513  2    1026 1025
6    7    0    5    7    2    3    7    6    3    11   10   3    19   18   3    35   34   3    67   66   3    131  130  3    259  258  3    515  514  3    1027 1026
8    9    0    8    10   1    8    12   3    4    12   11   4    20   19   4    36   35   4    68   67   4    132  131  4    260  259  4    516  515  4    1028 1027
10   11   0    9    11   2    9    13   4    5    13   12   5    21   20   5    37   36   5    69   68   5    133  132  5    261  260  5    517  516  5    1029 1028
12   13   0    12   14   1    10   14   5    6    14   13   6    22   21   6    38   37   6    70   69   6    134  133  6    262  261  6    518  517  6    1030 1029
14   15   0    13   15   2    11   15   6    7    15   14   7    23   22   7    39   38   7    71   70   7    135  134  7    263  262  7    519  518  7    1031 1030
16   17   0    16   18   1    16   20   3    16   24   7    8    24   23   8    40   39   8    72   71   8    136  135  8    264  263  8    520  519  8    1032 1031
18   19   0    17   19   2    17   21   4    17   25   8    9    25   24   9    41   40   9    73   72   9    137  136  9    265  264  9    521  520  9    1033 1032
20   21   0    20   22   1    18   22   5    18   26   9    10   26   25   10   42   41   10   74   73   10   138  137  10   266  265  10   522  521  10   1034 1033
22   23   0    21   23   2    19   23   6    19   27   10   11   27   26   11   43   42   11   75   74   11   139  138  11   267  266  11   523  522  11   1035 1034
24   25   0    24   26   1    24   28   3    20   28   11   12   28   27   12   44   43   12   76   75   12   140  139  12   268  267  12   524  523  12   1036 1035
26   27   0    25   27   2    25   29   4    21   29   12   13   29   28   13   45   44   13   77   76   13   141  140  13   269  268  13   525  524  13   1037 1036
28   29   0    28   30   1    26   30   5    22   30   13   14   30   29   14   46   45   14   78   77   14   142  141  14   270  269  14   526  525  14   1038 1037
30   31   0    29   31   2    27   31   6    23   31   14   15   31   30   15   47   46   15   79   78   15   143  142  15   271  270  15   527  526  15   1039 1038
32   33   0    32   34   1    32   36   3    32   40   7    32   48   15   16   48   47   16   80   79   16   144  143  16   272  271  16   528  527  16   1040 1039
34   35   0    33   35   2    33   37   4    33   41   8    33   49   16   17   49   48   17   81   80   17   145  144  17   273  272  17   529  528  17   1041 1040
36   37   0    36   38   1    34   38   5    34   42   9    34   50   17   18   50   49   18   82   81   18   146  145  18   274  273  18   530  529  18   1042 1041
38   39   0    37   39   2    35   39   6    35   43   10   35   51   18   19   51   50   19   83   82   19   147  146  19   275  274  19   531  530  19   1043 1042
40   41   0    40   42   1    40   44   3    36   44   11   36   52   19   20   52   51   20   84   83   20   148  147  20   276  275  20   532  531  20   1044 1043
42   43   0    41   43   2    41   45   4    37   45   12   37   53   20   21   53   52   21   85   84   21   149  148  21   277  276  21   533  532  21   1045 1044
44   45   0    44   46   1    42   46   5    38   46   13   38   54   21   22   54   53   22   86   85   22   150  149  22   278  277  22   534  533  22   1046 1045
46   47   0    45   47   2    43   47   6    39   47   14   39   55   22   23   55   54   23   87   86   23   151  150  23   279  278  23   535  534  23   1047 1046
48   49   0    48   50   1    48   52   3    48   56   7    40   56   23   24   56   55   24   88   87   24   152  151  24   280  279  24   536  535  24   1048 1047
50   51   0    49   51   2    49   53   4    49   57   8    41   57   24   25   57   56   25   89   88   25   153  152  25   281  280  25   537  536  25   1049 1048
52   53   0    52   54   1    50   54   5    50   58   9    42   58   25   26   58   57   26   90   89   26   154  153  26   282  281  26   538  537  26   1050 1049
54   55   0    53   55   2    51   55   6    51   59   10   43   59   26   27   59   58   27   91   90   27   155  154  27   283  282  27   539  538  27   1051 1050
56   57   0    56   58   1    56   60   3    52   60   11   44   60   27   28   60   59   28   92   91   28   156  155  28   284  283  28   540  539  28   1052 1051
58   59   0    57   59   2    57   61   4    53   61   12   45   61   28   29   61   60   29   93   92   29   157  156  29   285  284  29   541  540  29   1053 1052
60   61   0    60   62   1    58   62   5    54   62   13   46   62   29   30   62   61   30   94   93   30   158  157  30   286  285  30   542  541  30   1054 1053
62   63   0    61   63   2    59   63   6    55   63   14   47   63   30   31   63   62   31   95   94   31   159  158  31   287  286  31   543  542  31   1055 1054
64   65   0    64   66   1    64   68   3    64   72   7    64   80   15   64   96   31   32   96   95   32   160  159  32   288  287  32   544  543  32   1056 1055
66   67   0    65   67   2    65   69   4    65   73   8    65   81   16   65   97   32   33   97   96   33   161  160  33   289  288  33   545  544  33   1057 1056
68   69   0    68   70   1    66   70   5    66   74   9    66   82   17   66   98   33   34   98   97   34   162  161  34   290  289  34   546  545  34   1058 1057
70   71   0    69   71   2    67   71   6    67   75   10   67   83   18   67   99   34   35   99   98   35   163  162  35   291  290  35   547  546  35   1059 1058
72   73   0    72   74   1    72   76   3    68   76   11   68   84   19   68   100  35   36   100  99   36   164  163  36   292  291  36   548  547  36   1060 1059
74   75   0    73   75   2    73   77   4    69   77   12   69   85   20   69   101  36   37   101  100  37   165  164  37   293  292  37   549  548  37   1061 1060
76   77   0    76   78   1    74   78   5    70   78   13   70   86   21   70   102  37   38   102  101  38   166  165  38   294  293  38   550  549  38   1062 1061
78   79   0    77   79   2    75   79   6    71   79   14   71   87   22   71   103  38   39   103  102  39   167  166  39   295  294  39   551  550  39   1063 1062
80   81   0    80   82   1    80   84   3    80   88   7    72   88   23   72   104  39   40   104  103  40   168  167  40   296  295  40   552  551  40   1064 1063
82   83   0    81   83   2    81   85   4    81   89   8    73   89   24   73   105  40   41   105  104  41   169  168  41   297  296  41   553  552  41   1065 1064
84   85   0    84   86   1    82   86   5    82   90   9    74   90   25   74   106  41   42   106  105  42   170  169  42   298  297  42   554  553  42   1066 1065
86   87   0    85   87   2    83   87   6    83   91   10   75   91   26   75   107  42   43   107  106  43   171  170  43   299  298  43   555  554  43   1067 1066
88   89   0    88   90   1    88   92   3    84   92   11   76   92   27   76   108  43   44   108  107  44   172  171  44   300  299  44   556  555  44   1068 1067
```

Figure 18: Each lot of 3 columns is a successive value of S. In each lot, the first column corresponds to 'tidx0', the second to 'tidx1', and the third to 'idx_twid'

Appendix H MATLAB Scripts Used

Return to document location from whence reader came

```
clear, lgsamp = 11; n = 2048;
for s = 1:lgsamp
    iter = 0;
    m = 2^s;
    for k = 0:m:n-1
        for j = 0:(m/2 - 1)
            iter = iter + 1;
        end
    end
disp([num2str(s),': ',num2str(iter)])
end

idx = zeros(iter,2*s); twiddles = ones(iter,s);
new_twiddles = ones(2047,1); cnt = 1;

for s = 1:lgsamp
    iter = 0;
    m = 2^s;
    twm = exp(-2*pi*1i/m);
    new_twiddles(cnt) = 1;
    for k = 0:m:n-1
        twiddles(iter+1,s) = 1;
        for j = 0:(m/2 - 1)
            idx(iter+1,2*s-1) = j+k;
            idx(iter+1,2*s) = j+k+m/2;
            twiddles(iter+2,s) = twiddles(iter+1,s)*twm;
            if k == 0
                new_twiddles(cnt+1) = new_twiddles(cnt)*twm;
                cnt = cnt + 1;
            end
        iter = iter + 1;
        end
    end
end
```

```
a = 2.^(0:10);
invoc = [1164, 582, 291, 146, 73, 37, 19, 10, 5, 3, 2];
avgt_us = [25.506, 25.506, 44.51, 70.276, 120.879, 211.968, 393.858, 723.735, 1426.92, ...
    2355.46, 3521.77];
occ = [0.498, 0.498, 0.713, 0.887, 0.92, 0.995, 0.97, 0.979, 0.983, 0.984, 0.984];
smact = [0.29, 0.577, 0.668, 0.815, 0.887, 0.953, 0.967, 0.982, 0.991, 0.993, 0.994];
time_us = invoc.*avgt_us/1164;

% launches, blocks/launch -> blocks
% time/launch * launches -> time
% time/launch * launches/block
figure('Units','Normalized','OuterPosition',[0 0 1 1])
plot(log2(a), time_us(1)./time_us)
hold on
plot(log2(a), occ./occ(1))
plot(log2(a), smact./smact(1))
title('Improvement From Increasing Number of Blocks Per Launch')
xlabel('log2 of Blocks per Launch')
ylabel('Speedup Compared to 1 Block per Launch')
legend('Speed','Achieved Occupancy','SM Activity')
```