Fast and High Precision Signal Multiresolution Engine using Parallel Processors

E4750-2021Fall-HPSME-report

Manuel Jenkin Jerome (mj2984)
Electrical Engineering Department
Columbia University
New York, United States
mj2984@columbia.edu

Zixuan Yan (zy2501)
Electrical Engineering Department
Columbia University
New York, United States
zy2501@columbia.edu

***Abstract*—Multi-resolution analysis opens up a range of new possibilities to sample and analyse signals. It has been successfully employed in analysing the temporal nature of signals in areas of biomedical engineering, communication systems, quantum physics and music or speech analysis. However, due to computational constraints their adoption has been limited, especially in real-time systems. In this paper, we propose a algorithm for Multi-resolution Convolutions that is scalable for both small and large convolution kernel sizes, and uses parallel processing to achieve good speed up in computation. Architecture suggestions for FPGA or DSP Logic designs is also discussed.**

***Index Terms*—multi-resolution analysis, Convolution, Kernel, wavelet transform, CUDA**

I. Introduction

Multi resolution analysis refers to the transformation of a signal into it's spectral-temporal representation. This helps identify patterns in the signal that are not easily visible on superficial analysis of the raw data. This can be used for many purposes from signal classification, defect identification, and stability analysis, etc.

The most commonly used type of spectral representation is the Fourier transform. It has been employed successfully in a range of areas owing to its ease of implementation and also the existence of algorithms for fast computations (Fast Fourier transforms). However Fourier transform is not an ideal tool to analyse temporal structure of the signal, especially temporally compact ones, since the method averages them using an infinite time sinusoidal basis, hence blurring useful information. Note that, the representation is fully invertible for fourier transforms (assuming no truncation errors) and no information is lost. The issue is that the visual representation no longer represents something that can be easily deciphered due to the visual blurring.

More powerful tools exist for analysing non stationary signals. One such process involves correlating the signal with a finite time signal instead of an infinite time sinusoidal basis, and is called multi resolution analysis. This opens up a much broader range of options to define basis functions and find the most optimal choice for it.

One such category of Multi resolution functions is Continous Wavelet transforms. It involves convolution of input signal with a square-integrable function of different scale parameters. Since these functions can be made temporally compact, its can help analyse the temporal properties of the signal better. Since the choices for wavelets are many, it is possible to project many signal categories into at-least one wavelet transform where they shall be sparse. This can be used for efficient sampling and compression.

Unfortunately, for most Wavelet Transforms, especially Continuous Wavelet transforms, due to their complex representation, a Fast Fourier Transform equivalent is not available and this makes their implementation harder. And, while the open ended nature of Wavelet transforms gives options for many basis functions which can be analysed and tuned to our signal of interest, the computational limits stop it from being a reality.

Like wavelet transforms, there are many other choices of multi resolution functions that can be used to analyse and estimate different kinds of signals. But they also share the same computational limits as most wavelet transforms.

We propose a method to speed up Multi resolution computation using parallel processors. In addition, we also discuss implementation of additional functionalities, to account for deviations due to sampling amplitude errors, and sampling time errors. We also discuss functionality to add digital padding during the computation to analyse the signal properties beyond the sampling limits.

II. Literature Review

There are very few libraries available for GPU accelerated Multi-resolution transforms, and even in this small set, most of them are designed only for Discrete Wavelet Transform. This is because Discrete Wavelet Transform chooses only scales that can be split into a sequence of coefficients based on an orthogonal basis of small finite waves, which is easier to handle in computer programs, and has an equivalent of Fast Fourier transform. One example is the Fast Wavelet Transform for Discrete Wavelets by Keith G. Boyer [2].

Continuous Wavelet transforms on the other hand, and are not limited to orthonormal basis functions giving them the ability to take any arbitrary resolution scale. They generate an over-complete representation of the signal which is useful for analysing random signals in both time and amplitude. However, it comes at the complexity of computational feasibility using general methods. Therefore, there are very few reliable libraries for continuous wavelet transform. Many of them have limitations in their precision. Pywavelets [1] for example, starts to show numerical inconsistencies after a scale of 64 due to resampling of the convolution kernels for different scales (without resampling, it gets computationally more expensive for CPU based generation).

We were able to find one library on GPU accelerated computation of Continuous wavelet Transforms. In this paper [3], they make use of the fact that Time Domain Convolution is equivalent to Frequency Domain Multiplication. The Input Signal and the Wavelet function are first transformed into Fourier Frequency domain using FFT algorithm [4]. The resulting samples obtained are then multiplied. Finally an inverse FFT is done to get back the time domain signal.

However this has many compromises. First, Wavelets by design, are temporally compact and hence spread out in Fourier Frequency domain. By virtue of taking Fourier transform and computing the results, a trade off in frequency resolution and time resolution is already made, especially if done with finite window periods. For appreciably accurate results with a very compact mask, a large number of frequency bins may need to be generated. Second, each such transformation, done using computational methods have their penalty considering numerical accuracy since truncation happens in each step done in a finite bit depth machine. Also, the last stage, of taking inverse Fourier transform, may not be simple for super compact signals which would be spread out in Fourier space. Finally, the above algorithm is constrained to vanilla implementations of specific wavelets and would be hard to implement additional features onto them.

For our code, we intend to add additional functionality and also accommodate more than just wavelets. So we try to do the actual convolution, but in a smarter, and resource efficient way to ensure good speed up.

To sum up, we do not have too much reference materials for GPU acceleration of Multi-resolution Functions with arbitrary scale values. Therefore, we have developed all the algorithm designs based on original ideas and have written all kernel and PyCuda codes from scratch.

III. Core Algorithm

For our current code, we have implemented a Wavelet Transform (using Morlet Wavelet) and further references in our paper will refer to wavelet transforms. The code is flexible and can be expanded to any multi resolution convolution function as desired by the user with change in mask generation code.

A wavelet transform can be seen as multiple convolutions between input signal and different masks. We have divided the whole process into three parts. First part is mask generation, where each scale parameter will correspond to a particular mask of temporal extent defined by the scale. Assume we want to analyze the input signal in N scales, we will have N daughter functions. Each daughter function corresponds to one mask here. The second part involves pre-processing the masks generated to enable efficient use of GPU resources and improve computation throughput. This second task is required since the mask sizes can be of arbitrary lengths and can cause inefficient use of resources if not optimized well. The Last part involves convolution between the input signal and the N masks to get a total of N outputs. As it can be seen, the last task is the limiting factor in terms of computational speed.

Following sections provide detailed illustration about how this algorithm works in each stage. We organized them in following structure: section III-A is for mask generation, section III-B is for mask preparation and III-C, III-D together consist of the whole convolution process for all masks, since we apply different convolution method for different size masks.

A. Mask Generation

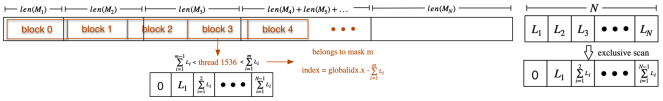

Fig. 1. Diagram of mask storage and generation. Array on left side is mask array which save all masks in 1-D form. Arrays on right side are length array and exclusive array. Below it is an example about how thread 1536 retrieves its corresponding mask scale and position in the mask.

As discussed before, the first step of wavelet transform is to generate a set of masks. The user shall define the scale parameters s for the multi-resolution function. Each scale parameter is directly related to the scale and stretch parameters for the wavelets, and hence directly affects the center frequency as well as the total number of samples in the mask. For example, for morlet wavlet, the length of each mask i is $L_i = 16s_i + 1$. Thus, when we define N masks with different scales, they would have N different lengths.

There are many possible ways to define how these N arrays are stored. One method is to save them as array of pointers and each pointer points to the start of one mask. However, pointer usage in python is not elegant and it will also introduce extra global memory access for each mask element because it will use two pointers to find each element. Another method would be to save them in a 2-D array, whose height is N and width is length of the longest mask L_N. It will make it convenient to visit every mask element. However, since the masks may be of different sizes, the memory space will be underutilized, as we would have to define array length as the largest of the masks available.

Therefore, we decide to save N masks in an 1-D array called *mask array*, concatenating one mask after another in a increasing order by its length, as shown in Fig 1. In this situation, it will be hard to differentiate one mask from another if we just look into the mask array. So we need another two arrays with N elements. First we define an array that saves

the length for each mask called *length array*. Next we do exclusive scan to length array and save the results, called *scan array*. Here we can find that each value of scan array defines the starting point of corresponding mask, which is useful following. Note that N is much smaller than the sum of all mask lengths, thus the extra memory space to save these two arrays can be omitted.

After defining how to save the masks, our code generates these masks using parallel computation. First we will define a kernel function that inputs a sampling point x, scale s and outputs a mask value m_i. For morlet wavelet, the kernel function is

$$m_i = \psi(x, s) = \frac{1}{\sqrt{s}} \exp^{-\frac{(\frac{x}{s})^2}{2}} \cos(5\frac{x}{s}) \quad (1)$$

Then, we assign one thread for each mask element as shown in Fig 1. Here, the main problem is how to let each thread know that which mask it is working on and which respective position it lies in this mask. We achieve this by comparing thread global id with values in scan array. Consider a thread with global id gid. After comparison ,it finds $\sum_{i=1}^{m-1} L_i < gid < \sum_{i=1}^{m} L_i$. It means that this thread belongs to $mask_{m-1}$ and its $x_{gid} = gid - \sum_{i=1}^{m-1} L_i$. Thus it knows its s and x, then can get its mask value by kernel function we defined before.

By designing mask storage and generation as above, we achieve some advantages compared to other naive implementation. First there is nearly no extra memory waste. Second, there are no idle threads since every thread has similar work to accomplish. Third, there will be fewer divergent blocks. In the 2-D mask array approach, divergence may happen when a block comes across a mask's boundary.

There are optional features such as Sampling Amplitude Error Compensation and Sampling Timing Error Compensation in mask generation. We will discuss them later, and the enumeration of these are also an input to the mask generate function.

B. Mask Preparation

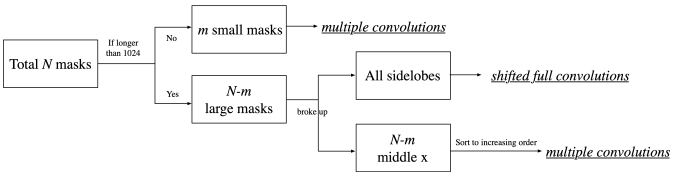

Fig. 2. Flowchart of how to divide all masks into different part and apply different convolution solutions.

Since wavelet transforms, at its core, are a set of convolution functions, the general concepts of Tiling and Shared memory can be used to speed up the computation. However, with a classical approach to shared memory and constant memory, we would encounter issues since there are more than one masks and each of varying sizes. After initial analysis, we found that the main bottleneck is constant memory size, or actually L2 cache size. For GPU Tesla T4, the L2 cache size for each block is 8KB[1]. That means if we use double datatype to save mask values, there can only be 1024 masks value which can fit in L2 cache. Masks that are longer than 1024, not fit in L2 cache thus the memory access can not be fully optimized. We use the length of 1024 as a boundary to divide to two groups - small masks and large masks.

For small masks that can fit into L2 cache, many of them can be fit into one kernel launch thus saving computational time. Thus we design a kind of convolution which we name 'Simultaneous Multiple Convolution', which will be discussed in III-C.

For large masks, we split up the kernel tasks into two parts - convolution of a small center portion where our result location is centered around the mask; and the corner portions which we call sidelobes, where the result location is shifted by some elements. Note that our split up is defined in a way that there can be more than 1 sidelobe depending upon the size of the Mask but there is always only one center portion. These sidelobes can be applied with a specific convolution kernel which we name "Shifted Convolution" and all middle left masks can be collected and applied with the same Simultaneous Multiple Convolution that we applied for small masks.

The mask prepare function does the split process as shown in Fig 2. It first divides the masks to two groups, and then split large masks to sidelobes and middle parts. Another important step is to sort all middle parts in a increasing order, which is required by the Simultaneous Multiple Convolution kernel. After the preparation, we can apply corresponding convolution methods to each masks group.

C. Simultaneous Multiple Small Mask Convolution

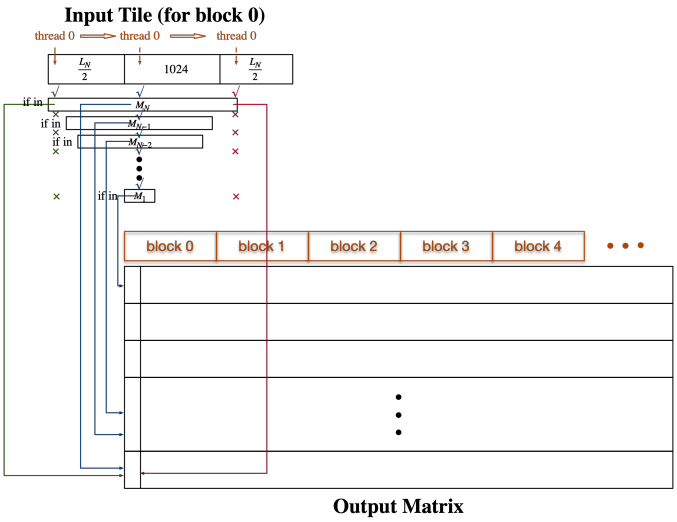

Fig. 3. Illustration for how multiple convolutions can be conducted by one kernel launch. Green, blue and red arrows show how one thread performs at leftmost, middle and rightmost position of input tile respectively.

[1]The GPU constant memory is generally larger than that of the block cache size. But we are limited by the block cache size for our algorithm, else it might cause lag due to paging adjacent areas of constant memory many times.

For small masks, multiple masks may fit into the constant memory together. Thus we proposed a kernel implementation to accomplish several convolutions at the same time instead one by one by assigning one thread for each column of output matrix. The motivation for this method is that we observed the output elements of a same column need similar input tiles. Although the padding size for input tile varies with mask lengths, the main part remains the same. If we assign different blocks for each row (i.e. a 2-D grid), there will N times extra input elements load that could have been shared. Here N is number of different scales, which also is number of all masks.

In this situation, each thread is responsible for the computation of N output elements. We achieve this by several steps. First, the whole block collaborates to load a large input tile that contains all elements needed by all masks to shared memory. The length of this input tile will be $blockDim + L_N - 1$, here L_N is the length of longest mask. Because the largest mask determines the amount of padding data required for our computation. The smaller masks require a subset of this input data.

In second stage, all threads will move from left side of the input tile to its right side, judging if current input element is needed by each output element. More specifically, each thread will begin from its thread id tid and end at $tid + blockDim$, taking $blockDim$ steps. During each step, it will compute the distance dis from this element to the center of its input tile, then compare dis to half of each mask's length $\frac{L_i}{2}$. If $dis <= \frac{L_i}{2}$, it means this input element is needed by mask i, thus do multiplication and add this partial result to corresponding output position. An important trick is that we apply nested if statement here. Since we arrange all N masks in a increasing order, if $dis > \frac{L_i}{2}$, it means dis is larger than any of the rest masks from 1 to $i-1$, then we do not need to step into the rest of if statements. In a word, each thread will compare dis with mask lengths from long to short and stop at the fist time the if statement failed.

For example, as shown in Fig 3, here illustrates the performance of thread 0 in block 0. It starts at the leftmost position of input tile, which is marked with green. Since $dis_0 <= \frac{L_N}{2}$, it will multiply current input value with first element of mask N. Then $dis_0 > \frac{L_{N-1}}{2}$, so it will stop and move to next position. When it approaches the middle of input tile, there will be more and more true if statement and the thread will do more and more multiplication and addition. When it arrives the middle point, as marked by blue arrows, all if statements will be True and it should do N multiplications and additions. After leaving middle point for rightmost point, there will be fewer and fewer true if statements. And when it arrives the rightmost position marked with red, there will be only one $dis_{blockDim} <= \frac{L_N}{2}$ again. After each thread has gone through the whole steps, the calculation of the output elements of the corresponding column is also completed, and the entire output matrix is finally obtained.

One thing to mention here is that although there are a lot of if statements, all nested if loop are done by one same thread and each thread will perform same action at one interval. Thus there will be no wrap divergence.

In conventional method, all convolutions are done separately. It would require a separate memory copy of almost the same data for each mask, and separate function calls, all of which introduce noticeable resource efficiency loss. Another important divergence in conventional method is that different convolutions with different masks will have different work burden, which leads to more and more idle threads when the total work approaches the end.

In our implementation, we have been able to fit multiple mask computation in one kernel. It will need less global memory access in total and leads to nearly no idle threads since every thread has similar work burden, which will finally result in a significant speed up. A possible weakness of this method is that it is only efficient when input is very long, otherwise there will be too few active threads. Since real world signals are usually millions or even billions of samples in length, this limitation is a non issue for real world scenario.

D. Partial Large Mask convolution

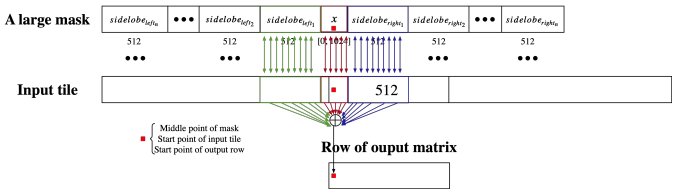

Fig. 4. Illustration for how large mask can be split into sidelobes and center mask. The whole mask can be divided into n pairs of sidelobes and one middle mask x. All parts will compute their own partial results and add temp result into output position. Only two of sidelobes' computation process is visualized but all others are similar.

For large mask whose length is larger than 1024, one mask is too large for L2 cache, so we must break it up as shown in Fig 4. We can represent the length of mask i by $L_i = 2nC + x$. Here $C = 512$ represents the length of one sidelobe, also half size of L2 cache. $x \in [0, 1024)$ means the length of center mask. n is number of pairs of left and right sidelobes. Therefore, each large mask can be easily split into n pairs of sidelobes, each pair contains one left sidelobe and one right sidelobe in symmetric position, both have 512 elements. And one center mask contains the rest section $x = L_i \% 1024$. For example, if a mask is 3201 elements long, there will be 3 pairs of sidelobes and middle part x will be 129 elements long.

In this manner, the whole convolution can be divided into $2n + 1$ parts, each part can derive its partial result by a convolution between a smaller input tile and a smaller mask array. At last, we add up all these partial results and get the final value of output. We place the rest part in the middle instead of head or tail because of a specific reason. From Fig 4 we can observe a red square that indicates the middle point of whole mask M, the middle point of the rest part mask x, the start point of input tile, the start point of output row. These four

positions are align on the same column. That means if we take middle part x out, it is actually doing a normal convolution on a smaller input tile. However, for sidelobes, they are not. We can observe that their results are shifted several elements away from the corresponding position if we do normal convolution.

Therefore, we design a slightly different convolution kernel called 'Shifted Convolution' for all sidelobes. The input of this convolution is the same as normal convolutions except an extra parameter to indicate how many elements its result should be shifted. One thing to mention is that we always process a pair of left and right sidelobes together. Since they are symmetric, we only need to provide the shift paramter for left sidelobe and the shift distance for right sidelobe will be its negative value. We can computed shift parameter for left sidelobe as $shift = \frac{L_N}{2} - \frac{C}{2} - start$. Here $start$ means the starting point for this sidelobe. For example, for the sidelobe on the leftmost of this large array, it will be $shift = \frac{L_N}{2} - 256 - 0 = \frac{L_N}{2} - 256$.

And for all middle parts x, since they are all smaller that L2 cache size, the 'Simultaneous Multiple Convolution' we defined in 3 is suitable to compute them all at one kernel launch. There is one extra step to rearrange these middle parts to increasing order by their lengths otherwise the nested if loops can not work. It is also important to record their original orders since we need to rearrange the results we get from the multiple convolution to original order, then add these two results up to get the final convolution result for large masks.

IV. Features

This section discusses both the pre-embedded and user definable features in our current implementation.

A. Speed optimizations

Numerous optimizations have been done to ensure fast computation times both for the mask preparation kernel as well as for the multi resolution convolution kernel. The current implementation is done using PyCuda on a CUDA Enabled GPU. The following discussion will be in context of CUDA feature usage, especially Mask Convolutions where a lot of CUDA specific features have been used.

For the case of mask generation, when traversing the mask scan lengths, we traverse from the last element to the first element. This will mean the samples to be generated for the largest mask will have least waiting time in this comparison stage. Since there are more elements to be computed for the largest mask, this will in turn save many cycles in many threads hence making computation faster. Only the threads computing smaller masks would spend more time in this comparison stage, and since they are very few in comparison, overall computation time is faster. Other areas of similar speedup optimizations have been discussed earlier in the core algorithm section.

As is the general case of convolution functions, the code makes use of Shared Memory for storing the input range for convolution. During the copy from Global memory to shared memory, optimizations are done to ensure good coalescing and less cycle stealing operations. The input is padded to reduce an if comparison for all threads as well.

It also makes use of broadcasting feature of shared memory in CUDA. In the code for Small Mask Multiple Convolution, each of the if comparisons involves a broadcast of the mask size to each thread. For the local copy of the temp data, it is written in a way that ensures no bank conflicts (assuming the threads are scheduled as expected).

For large masks our code generates a symmetry to ensure that there are very few comparisons, and if so all threads exit quickly leaving them open for the next block. It has also been optimized to ensure coalescing when accessing the global memory.

While the above optimizations are done with respect to CUDA, the same can be translated effectively to Vulkan and OpenCL since they have equivalent alternative to concepts of shared memory and constant memory.

B. Numerical Stability Optimizations

Numerical Stability has been of significant importance to the design of the algorithm and it has been optimized in many areas. Whenever possible, division has been changed into multiplication at the final step, to ensure less errors in the intermediate steps.

The user can input two variables for the scale parameters scale numerator and scale denominator both of which together define the scale value when the numerator is divided by the denominator. It is kept as two separate variables until the final generation stage where they are used optimally.

In addition, we use more than one running variable for the large mask convolutions. Since the convolution is running in Floating Point precision, the contrast between the addends determines how accurate the result is. If only a single variable is used there can be a significant error in Large mask Convolution when all inputs truncated in each step add up to be a significant error. For example, if a large value is added in one stage in subsequent stages the addends are very small, then in each stage the small addends will be truncated significantly.

Such small errors, if left unattended can add up to a significant amount when repeated many times. In smaller masks, and the center convolution of larger masks, the number of summing elements is small (of the order of 500 at most) and since we are using atleast FP32, the errors here will be negligible. Since we are constrained with the number of local variables for small masks, it would not be possible to add more precision there, and in any case, they don't contribute to significant deviations. Hence for the Large masks sidelobes, we use more than one running variable to store different contrasting levels the data and in the end we add these two variables to give the result. This ensures better numerical stability.

V. Optional Features

In our current code a simple implementation of the below 3 functions are implemented, adding a constant value to the time and/or output amplitude. Since it is defined inside the kernel and the kernel has access to the mask scaling

factor, a frequency domain model based compensation can be implemented inside the kernel with little effort.

A. Sampling Amplitude Error Compensation

This is a function implemented to allow user to compensate for Sampling Amplitude Errors, both for the input signal (Quantization noise during sampling) and for the mask (Truncation due to finite bits available). The compensation can be modelled as a function of scale parameters (or a function of frequencies which can be mapped to the center or average frequency of the scale). This will generate a new mask that has this compensation embedded into it.

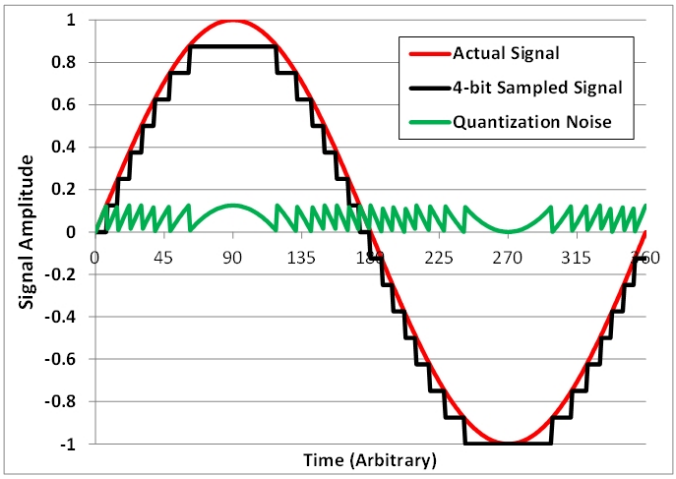

Fig. 5. Figure Showing quantization noise of a sinusoidal signal sampling

Dithering Functions are generally available in frequency domain models and can be used for this purpose. In addition, CUDA also provides different round-off modes to further customize these functions.

B. Sample Padding

This feature lets the user analyse properties of the signal beyond sampling limits, by using probabilistic models. The user can do a 2x or 4x Oversampling Digitally on top of the input signal using a user definable padding algorithm, which would then become the new signal for convolution. This can also be used to compensate and analyse aperture time errors of the input sample. Currently, this feature uses CPU to do the padding before feeding to the GPU convolution kernel where it is parallelized.

C. Sampling Timing Error Compensation

In it's present form, this function can be used to set clock sampling deviation from a frequency domain model. It will be used to distort the mask structure in the temporal domain and compensate for the sampling time errors in the input sample. Just like Amplitude Error Compensation, the Time error compensation can be inputted as a function of scale parameters. Many of the phase noise models currently are available as frequency domain representations, and can be used.

VI. Result

This section covers our results, and analysis of our implementations in terms of computational speed and numerical precision. VI-A gives a detailed description about where and how we conduct these tests. VI-B provides results and analysis about mask generation part. VI-C and VI-D provides runtime and numeric analysis of multi convolution part. Finally, VI-E presents the profiling result.

A. Test setting

Our parallel algorithm for morlet wavelet transform is tested using PyCuda implementation and run on Tesla T4 GPU on Google Cloud Platform. The Optional features of sampling error compensation and timing error compensation were not included in this comparison. Three versions of code that have been realized, all performing the same convolution task (masks generated from PyWavelets library) for comparison. First is an implementation of core algorithm we put forward in this paper. The next is a very simple GPU kernel implementation without memory optimizations. It has been timed and profiled to compare with the optimized algorithm. Further description of the simpler GPU kernel is skipped due to limited space. Finally we also have a python serial code implementation, using the PyWavelet Library.

It is important to note that since our computation is in double precision but T4 does not have dedicated double precision computation units, it would use multiple cycles to finish the computation. The results obtained might be better with GPUs that have dedicated double precision computation units.

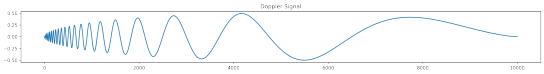

Fig. 6. Doppler Signal, a classical unstationary signal. It works as input signal for all the tests.

While our algorithm works for diverse wavelets and diverse input signals, for the convenience of stable results, we choose Doppler Signal and Morlet Wavelet as our input signal and wavelet function. Doppler Signal, as shown in Fig 6, is a non stationary signal whose frequency components shift from high to low as time increases. Such signals are very suitable for analysis by wavelet transform.

B. Mask Generation

TABLE I
SPEED UP RATIO AND RESPECTIVE DIFFERENCE OF MASK GENERATION

Number of scales	16	64	256	1024	4096
Speed up over CPU	0.36	5.26	5.23	8.69	10.02
Speed up over naive GPU	0.16	0.83	1.49	2.12	3.05
Respective Difference	0.00%	0.00%	-0.00%	0.00%	0.00%

For mask generation, we conducted five tests when there are 16,64,256,1024,4096 different scales. From Fig 7, we can observe that when there are 16 masks, our algorithm works

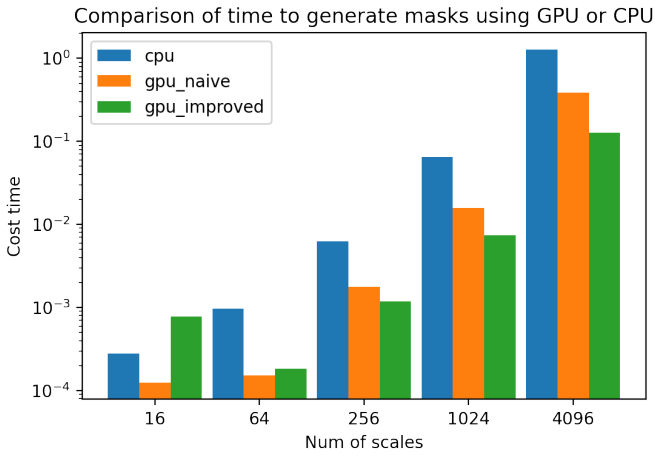

Fig. 7. Test result of mask generation for scale in [16,64,256,1024,4096] by three different implementations. The scale of y axis is plotted as log scale for better visualization.

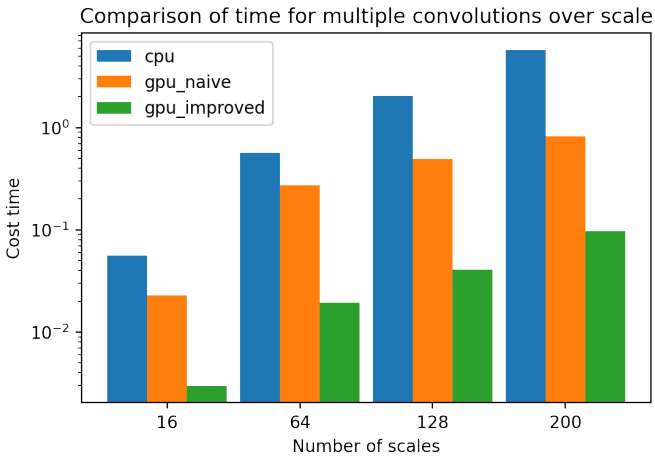

Fig. 9. Test result of multiple convolutions part for number of scales in [16, 64, 128, 200] by three different implementations, input length is fixed at 102400. The scale of y axis is plotted as log scale for better visualization.

even slower than CPU code because input length are too short for parallelism. However, when number of masks increases, our algorithm is performing progressively faster and faster than the CPU implementation. At the same time, naive GPU implementation works faster than our algorithm in 16 and 64 scales. It is because there are little space to optimize when input scales are small. With the increasing number of scales, the gap between our kernel and naive kernel is gradually widening.

From Table I, we can observe that our algorithm finishes the task 10 times as fast as the CPU mask generation code and 3 time as fast as the naive GPU kernel. This confirms that the optimization methods we have employed has been successful in mask generation process.

C. *Multiconvolution:Runtime Analysis*

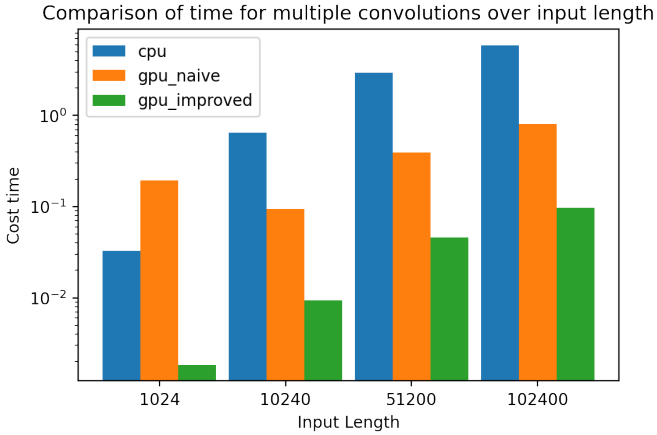

Fig. 8. Test result of multiple convolutions part for input length in [1024, 10240, 51200, 102400] by three different implementations, number of scales is fixed at 200. The scale of y axis is plotted as log scale for better visualization.

For the main part of our algorithm - multiple convolutions, we present its runtime analysis in this section. The numeric

TABLE II
SPEED UP RATIO FOR MULTIPLE CONVOLUTIONS OVER INPUT LENGTH AND NUMBER OF SCALES

	1024	**10240**	**51200**	**102400**
16	0.15	1.13	2.05	2.53
64	0.08	0.90	2.16	2.35
128	0.35	3.96	4.15	4.50
200	0.16	6.69	7.24	7.53

Rows in the table are number of scales (frequency resolution)
Columns in the table are input length
Values in the table are speed up ratio by our algorithm over CPU

analysis is presented in the next section. We conducted several tests changing input length from 1024 to 102400 and number of scales from 16 to 200. Larger input lengths and scales can also be inputted for further analysis, and it is likely to show more gains in performance. From Figure 8 and 9, we can observe that as our input length in increased, our algorithm quickly becomes faster than CPU and naive GPU kernel for all the mask sizes. Our goal of scalability in both directions has been realized.

From Table II we can see more diverse situations. Speed up is not conspicuous when input is too short or scales are too few, it is because of the way we assign threads. When input length is too short or number of scales, which is also number of masks are too small , there will be few active threads thus GPU resources are not fully exploited. Although this improved algorithm can make up to 7 times faster, it is still not as much as we expect. We have analyzed several bottlenecks for current implementation and some possible solutions as following.

First, the Compute to Global Memory Acess (CGMA) ratio of current implementation is too high. We have mathematically deduced the formula and found its CGMA ratio is up to several thousands. In this manner, bottleneck will be computational resources instead of memory transfer. There are more evidence to prove this conclusion in profiling. At the same time, we are using NVIDIA Tesla T4, which does not have FP64

computation units. However, we must use double datatype to preserve numeric precision. Therefore, computational ability of GPU will be a good place to improve. We found that Nividia Tesla V100 is also avaliable on GCP. Tesla V100 has 7.5 TFLOPS FP64 performance, which is nearly 30 times than 254.4 GFLOPS for Tesla T4 (assuming all other parameters not bottlenecked), so it is a nice substitute machine. However, we ran into some problems when we wanted to conduct more test on this new machine currently, thus can not report the performance on it. But it is still a promising way to try.

Second, our current code launches one kernel for each one of large mask. However, too many kernel calls may lose parallelism. It is possible to write a large kernel call that iteratively loads in sidelobes and does convolutions for different large masks. In this way, we can reduce a lot of time wasted on transferring data between kernel and host. We can also see more evidence for this idea in profiling.

Third, we are using double datatype to preserve numeric precision for now. If we switch all datatypes to float, it will be a significantly speed up since one double computation can be 32 times lower than one float computation in our current setup, since there is no FP64 computation units in Tesla T4. However, it will lose some numeric stability. This approach is only practical for applications that have little requirement on precision.

D. Multiconvolution:Visualization and Numeric Stability

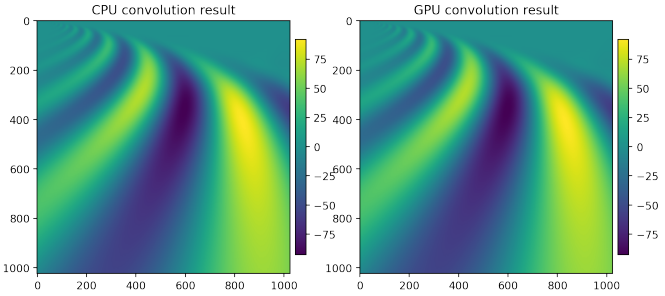

Fig. 10. Visualization of outputs of CPU and GPU wavelet transform on Doppler Signal.

TABLE III
RESPECTIVE DIFFERENCE FOR MULTIPLE CONVOLUTIONS OVER INPUT LENGTH AND NUMBER OF SCALES

	1024	**10240**	**51200**	**102400**
16	-0.00%	-0.00%	-0.01%	0.00%
64	-0.00%	0.00%	-0.02%	0.02%
128	-0.01%	-0.02%	-0.02%	0.01%
200	-0.00%	0.19%	-0.03%	-0.11%

Rows in the table are number of scales (frequency resolution)
Columns in the table are input length
Values in the table are respective difference by our algorithm over CPU, which is computed by $(gpu_{conv} - cpu_{conv})/cpu_{conv} * 100$

In this section, we visualize the output of our GPU implementation for wavelet transform and compare its numeric stability with CPU implementation. The masks prepared by PyWavelets' Morlet Wavelet is used to ensure matched masks and compare the convolution kernel accuracy.

In Fig 10, we visualize the output matrix of (N, T) as an image. Note that this wavelet transform is conducted on an 1024-elements long input Doppler Signal and with 1024 scales just for better visualization. Our actual kernel will not be enough efficient in this situation. We can see that outputs of wavelet transform on CPU and GPU are nearly the same. The output itself is also reasonable. As we discussed before, Doppler Signal has more higher frequency components in earlier timestamps and more lower frequency components in latter timestamps. Actually the output image is a perfect reflection of this character.

First we must note that yellow and black pixels both reflect high absolute output values, which means a strong frequency, and cyan pixels are zero, which means there are no frequency component here. Also scale s can be seen as inversely proportional to the frequency, thus small s means high frequency and vice versa. We can see that in earlier time (approaching 0), there are more frequency components in small scales, which means more high frequency components. While for when time increases (approaching 1024), there are more and more 0s in small scales and most of energy concentrates on large scales, which means lower frequency components. It is actually what we expected from multi-resolutional analysis that provides information about frequency and time domain at the same time.

Also, from Table III, the attention to numeric stability optimizations can be seen. There are nearly no precision loss when inputs are not extra large (top left of the table). When there is a very long input signal or many scales to process, there is a slight difference of 0.1% to 0.2%, which is totally acceptable.

If there is a need to increase numeric stability even more for higher precision application, one simple suggestion is to running more temp variables in large mask. We are now running two of them, using one of them to collect large values and one to collect small values, then add them up finally. If more temp variables are kept for different value level, the result will be more accurate. However, it will absolutely lead to more computational and memory burden since there are more memory space needed to save these variables and more additions to do.

E. Profiling

Shown in Fig 11 and 12, the code has been profiled using the Nvidia Nsight Profiler for an input size of 1.024 Million Samples and masks of scales from 1 to 200 linearly spaced.

Both the Small Mask Multiple Convolutions and Large Mask convolutions achieve significant Streaming Multiprocessor Usage, which is a good thing. The achieved active warps per SM also reaches close to the theoretical maximum. In addition the Small Mask convolutions also make good use of the available memory resources. Here we also observe that in both two kind of convolutions, we are approaching the full bottleneck of computation resources, leaving memory usage as

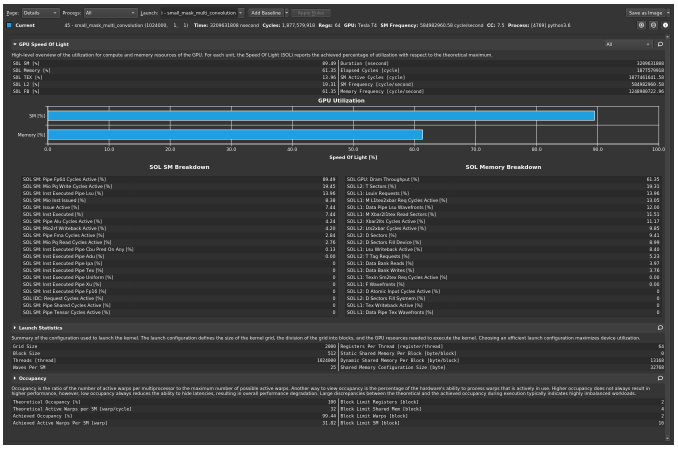

Fig. 11. Profiling result of multiple convolutions for small masks.

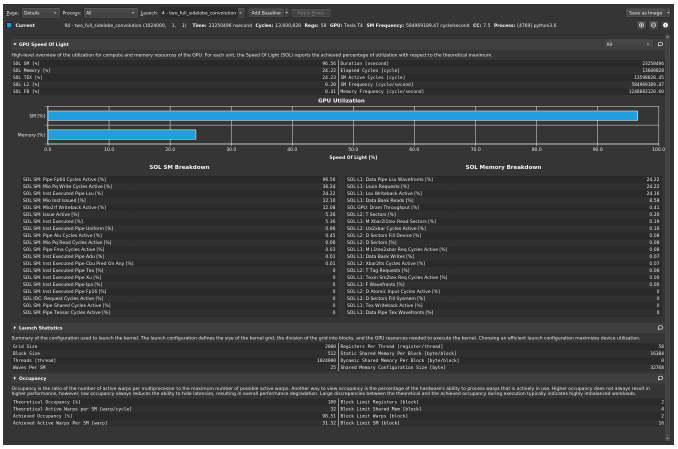

Fig. 12. Profiling result of multiple convolutions for large masks.

40% - 60%. It provides the evidence that if we switch to GPUs with higher computational ability and FP64 performance, we will get better result (since the FP32 units have to compute 32 times to generate a FP64 Result). Another thing to notice is that memory usage of large masks is much lower than small masks, it may be becuase of the multiple kernel calls of large masks. If we can combine these calls into a large kernel call, the result will also improve.

One issue is that for the Small Mask Multiple convolutions, the required number of local variables is not accomodated within the 64 Variables per thread limit and hence the shared memory for the block becomes dynamic. We predict this to be the reason why the speed up in small mask multiple convolution, while good, is not as high as the speed up achieved in Large mask convolutions. In any case, it still performs quicker than the naive implementation hence it makes up for this with the overall masks computed within a single kernel.

VII. Conclusion and Future Work

A resource efficient and scalable algorithm utilizing parallel processors has been designed and tested for performing Fast and High Precision Signal Multi-resolution. It offers considerable speedup over both CPU implementations and naive GPU implementations when the input and mask sizes are large enough. Profiling has been done to ensure resources are utilized as expected.

Another point to note is that the algorithm can be scaled in both ways even in terms of the data precision of the hardware, since we have the option of having multiple variables storing different contrast levels of the addends, before finally adding the different results together. So it can be scaled down match FP16 hardware with little effort, yet delivering a decent boost in numerical precision compared to conventional algorithms for the same hardware setup.

It has currently been implemented in PyCuda and we plan to extend it to Vulkan C++ in the future. Additional Features have been implemented as proof of concept. We make an open invitation to Engineers and Researchers from different domains for suggestions for models to be used for these additional features to analyse Phase noise, Quantization noise and Aperture Uncertainty deviations. If permitted, we plan to utilize this code for our future courses which may involve sparse representations, analysis and compression algorithms for signals.

Along with our analysis we would also like to share suggestions on Architecture for multi-resolution analysis acceleration that could be implemented in FPGA designs or DSP logic designs.

A. Architecture Suggestions

We can see that the task of multi-resolution convolution is well suited to the Parallel Computing Graphics processors used here. For an efficient FPGA design to do the same task, it can leverage a lot of the Graphics Compute pipeline with a few suggested changes to make it even better. Since we have to load up a new kernel, and hence re-initialize the shared memory every time, a noticeable amount of time and resources is underutilized. A dynamically buffer-able constant memory and thread caching, with a synchronized status register would reduce total number of memory loads needed. This would allow more mask functions to be run with one shared memory copy before moving to the other. More local variables or additional L4 cache hierarchy to store local variables can be useful if the number of small masks to be computed is more. The paging from this L4 cache to the main cache can be simple, initialized only when many masks are to be computed together, and hence it will not add to computational issues if implemented well.

REFERENCES

[1] G. Lee, R. Gommers, F. Waselewski, K. Wohlfahrt, and A. O'Leary, "Pywavelets: A python package for wavelet analysis," *Journal of Open Source Software*, vol. 4, no. 36, p. 1237, 2019.

[2] K. G. Boyer, "The fast wavelet transform(fwt)," Master's thesis, Citeseer, 1995.

[3] Z. Deng, D. Chen, Y. Hu, X. Wu, W. Peng, and X. Li, "Massively parallel non-stationary eeg data processing on gpgpu platforms with morlet continuous wavelet transform," *Journal of Internet Services and Applications*, vol. 3, no. 3, pp. 347–357, 2012.

[4] K. Moreland and E. Angel, "The fft on a gpu," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003, pp. 112–119.

INDIVIDUAL CONTRIBUTION

This project involved us collaborating and exchanging ideas together. It involved both our contribution in each stage as we iterated upon each other's work to achieve the final result. The total list of tasks have been presented in a tabular form.

Task	Manuel Jenkin Jerome (mj2984)	Zixuan Yan (zy2501)
Project Planning	50%	50%
Literature review	50%	50%
Algorithm design	50%	50%
Code generation	50%	50%
Profiling	50%	50%
Result Analysis	50%	50%
Presentations	50%	50%
Report	50%	50%