Application of GPU processing for acceleration of the non-uniform discrete Fourier transform

Kai Zhuang, and Daniel Trad

ABSTRACT

We implemented a non-uniform discrete Fourier transform(NUDFT) algorithm in C++ on both CPU and GPU. In this paper, we explore the advantages and challenges of the implementation of the CUDA API on the non-uniform discrete Fourier transform. The implementation of CUDA is interesting in this application because the non-uniform transform is very time-consuming on the CPU due to $O(n^2)$ complexity and the high amounts of memory operations that are performed due to the non-uniform natures of the transform. We implement the NUDFT in CUDA to see if the superior multi-threading performance of GPUs can offset the losses due to memory operations, to make NUDFT and in extension interpolation via NUDFT a computationally viable alternative.

INTRODUCTION

With the increased adoption of specialized processors like GPUs, previous algorithms that were too costly to implement commercially can be considered. Advancements in technology, especially in the consumer GPU market in recent years provide a very high-performance alternative to the traditional CPU-based parallel processing methods for mathematical operations. GPU processing's main draw is the massively parallel pipeline that exists on GPUs. A consumer GPU currently has 10s of thousands of threads that it can compute on, which allows for much higher throughput than traditional CPUs. Although many seismic workflows are highly parallelizable, depending on the implementation some algorithms may see limited benefits from some parallelization methods. While originally used only for graphics processing, over the years advancements in GPU technology have made general parallel computing on the GPU more and more appealing, known as general-purpose GPU computing or GPGPU. As the graphics processing unit (GPU) is traditionally used for computer graphics their architecture is purpose-built differently from a CPU's. One approach is to think of a CPU as "multi-serial" processor where each thread can do independent (serial) jobs unrelated to each other at very high speeds. The GPU is then a parallel processor where each operation executed by its grouped threads must be the same but it can process significantly more operations in parallel than a CPU can. For purposes of general signal processing, the fast Fourier transform is often used over the discrete Fourier transform as it performs a similar job while being significantly faster for larger data. The discrete Fourier transform however has the advantage of being able to be performed on irregularly sampled data which is regularly encountered in real-world data, generally, the irregularly sampled data would either be resampled or binned to a regular interval for FFT but will cause errors when binned data are not similar or where information will be lost due to binning. The application of a DFT operator allows for the binning process to be bypassed entirely and allows for work with exact sampling, with the trade-off being computational expense.

CUDA

In 2007 the graphics card engineering company, Nvidia, introduced the Compute Unified Device Architecture API, otherwise known as CUDA. CUDA allows programmers to utilize the GPU rendering engine to be used for general-purpose parallel computing (Nickolls et al., 2008). This allows for users to access the very high throughput GPU threads for use in parallel processing. A GPU differs from a CPU in many ways: generally, a GPU is made with groups of processors, called streaming multiprocessors (SMs), which each having an allocation of lightweight processing threads. Each SM has an allocation of L1 cache and shared memory to which its threads can access. This general architecture can be seen in Figure 1. The SMs are sub-divided into groups called warps, with each warp containing a set of usually 32 threads. This is important because, in each execution, all threads in a warp must do the same operation. This is different from threads in a CPU, which can perform different operations for each thread. This layout allows for very fast calculations due to the sheer number of very lightweight threads. Many computational algorithms like convolutions and matrix multiplications can benefit from this pattern.

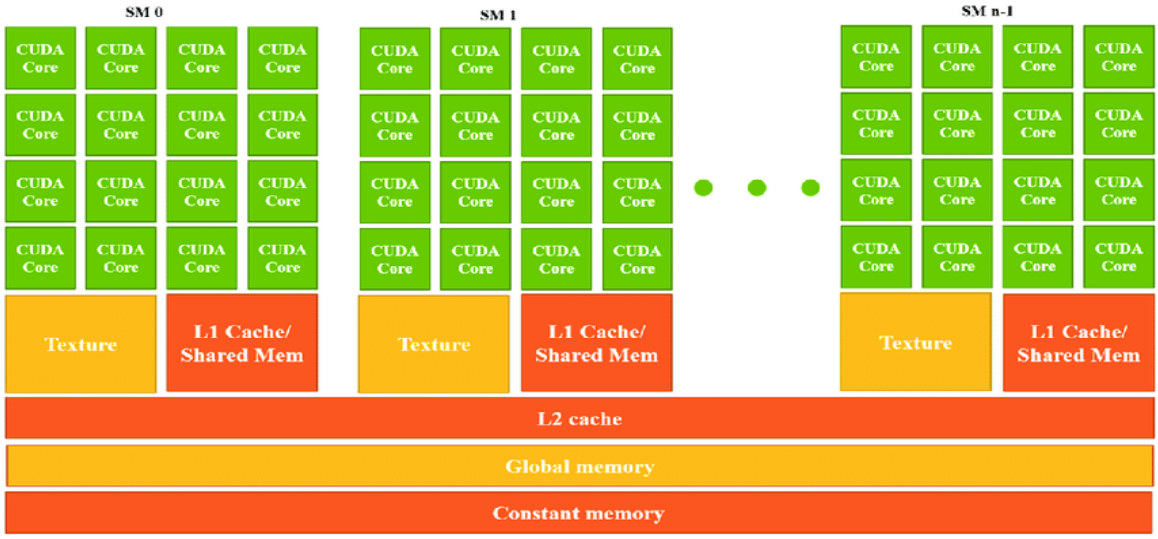

FIG. 1. General layout of a GPU (Wang and Kemao, 2017).

CUDA is built on top of the C/C++ language and requires Nvidia's proprietary compiler called NVCC in addition to a standard C/C++ compiler. CUDA introduces many new qualifiers, data types, and API calls for processing on the GPU. With CUDA programming in C/C++, a set of new qualifiers for functions determines whether the codes will run on the device (GPU) or the host (CPU). Functions that run on the GPU are called kernel functions. Usually, they are invoked by the CPU and run on the GPU, but can also be invoked from the GPU itself. Coding for CUDA is unique because we need to explicitly define which type of memory each operation will store information, and use a to transfer the information required to perform each operation since the high-speed "shared" memory is small and does not communicate directly across different parts of the GPU. These transfers are usually slow, so they have to be done in such a way that they do not outweigh performance gains. To get the most efficiency out of the program, CUDA requires an understanding of the low-level GPU architecture. Some important architectural details that impact the performance of a program include proper allocation of both global and shared memory, where shared

memory can only be accessed by threads inside a block but have significantly lower access times. Figure 2 show the relative speeds from accessing information from many parts of the system.

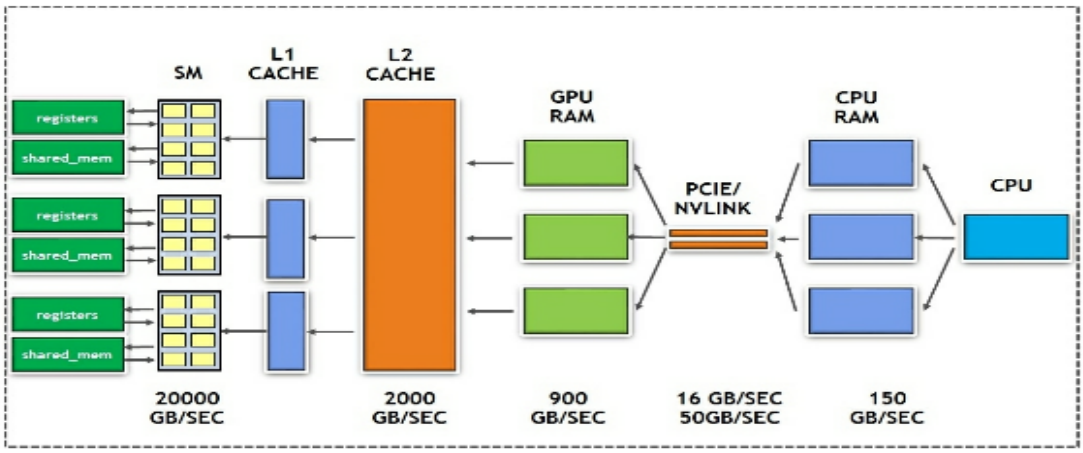

FIG. 2. Transfer memory bandwidths in different memory sections. (Han and Sharma, 2019).

Thread wraps are another factor we need to keep in mind when we program for CUDA as all threads under a warp execute the same instruction. Other concepts that can make a large difference in efficiency are newer features recently added to GPUs, like tensor cores, which are much faster than previous methods for matrix calculations on specifically sized matrices. The programmer must also keep in mind not to excessively transfer data from the system memory to the GPU memory due to limitations in PCI-E protocol signaling. This is due to compatibility for older x32 CPUs allowing only the exposure of 256MB of GPU ram at a time to the CPU through the PCI-E Base Address Register (BAR) for memory access. With proper coding a GPU program can significantly outperform a CPU program in parallel tasks.

Non-uniform discrete Fourier transform

The non-uniform discrete Fourier transform (NUDFT) is important in the field of signal processing as in many applications the signal recorded does not exist in regular intervals across some dimensions. In many applications of the NUDFT, the signal is interpolated into regular sampling then run through an FFT algorithm. For our applications, because we are seeking to interpolate with NUDFT, resampling/interpolating to regular sampling is in contention with our objective. The standard DFT equation can be seen in equation 1.

$$H(\omega) = \sum_{k=0}^{N-1} h_k e^{-i\omega k \Delta t} \tag{1}$$

Where $H(\omega)$ is the Fourier coefficient, h_k is the signal at the k^{th} sample ω is the wavenumber and N is the number of frequencies. For the purposes of signal processing,

the fast Fourier transform (FFT) is generally used, this transform takes advantage of the regularity due to its periodicity, reducing computational complexity to $O(nlog(n))$. This is different from the direct evaluation of the discrete Fourier transform with complex $O(n^2)$ of which NUDFT shares. Processing using NUDFT is also costly in that because the samples are not evenly spaced NUDFT takes more time to calculate due to the extra calculations needed to set up the unevenly spaced matrix. This problem is similar to the computer science problem of sparse matrix-vector multiplication where sparse matrices must account for the non-uniform way in which data is stored. In 5D interpolation, binned FFT interpolation is commonly used for its speed, in this application seismic traces are binned together to form a regularly spaced dataset. The regular spacing then allows for the use of common FFT algorithms, allowing for a relatively fast 5D interpolation algorithm. 5D interpolation using the FFT operator however causes an issue at long offsets due to the binning of the data where values at different offsets are binned together causing loss of information in those areas (Trad, 2016). By implementing 5D interpolation using a NUDFT operator then bypasses the need to bin traces for a uniform grid, however, this adds a significant processing requirement by significantly increasing the computational complexity of the problem from $O(nlog(n))$ to $O(n^2)$.

RESULTS

We ran the test algorithms for DFT on a range of different hardware to show the performance of the algorithms in different conditions, a laptop, and two high-end workstations. The laptop utilizes an intel i7 8750H 6-core CPU with an Nvidia RTX 2060 MAX-Q GPU, the first workstation is equipped with an AMD Threadripper 3960x 24-core CPU, with an Nvidia RTX 3080, the second workstation has an intel i5 8600 8-core CPU equipped with an Nvidia RTX 1080 GPU. The test was run with 25 shots 297 receivers and 3600 time samples. These results can be seen in table 1.

Processor (API)	Time for 25 shots (s)
i5 8600 6-core (openMP)	1.5996
i7 8750H 6-core (openMP)	1.752
TR 3960x 24-core (openMP)	0.8539
GTX 1080 (CUDA)	0.0447
GTX 1080 (CUDA THRUST)	276.43
RTX 2060 MAX-Q (CUDA)	0.0563
RTX 2060 MAX-Q (CUDA THRUST)	304.43
RTX 3080 (CUDA)	0.0379

Table 1. Table of average runtimes for DFT implementation.

It should be noted that although the RTX 2060 series GPUs are newer they have a lower CUDA core count ($\approx 75\%$) with respect to the GTX 1080 which results in the difference in computational times seen in the table. The results recorded are only the operator kernel runtimes for this implementation as our current work is ongoing and memory copies still need to be optimized, because a significant amount of runtime is eaten up by memory transfers between GPU and CPU. As pure DFT performance is the objective of our paper

we will be mainly looking at the kernel runtimes with an updated paper later for total program runtimes. The DFT operator was implemented in three different ways, once on CPU using openMP, twice on GPU, using both a custom CUDA kernel as well as a prepackaged CUDA thrust library for matrix operations. The comparison between the CUDA and CPU times is as expected with a significant speedup seen by the CUDA GPU due to its massively higher thread count. The Thrust library results in comparison to the other two were significantly worse by multiples of a magnitude, we are as yet unsure of why performance is so poor using the Thrust library matrix operators. One possible cause of the poor performance of the thrust library is that the inner product function does not compute a correct inner-product for complex vectors, thus a custom functor was made and added as an argument to the function. Upon profiling the code to diagnose the issue, the results show a significant amount of runtime was used in a thrust copy operation which is not used anywhere else in the code. These memory operations are what most likely caused the significant increase in processing time.

CONCLUSION

The use of the discrete Fourier transform has generally been overlooked in favor of the much faster fast Fourier transform for most signal processing uses, however in certain applications the FFT cannot be used because data is given in a format that is un-evenly sampled. Traditionally the solution is to either interpolate or bin the data before sending it to the fast Fourier transform. The introduction of GPU processing allows us to implement operators that are generally too expensive to implement on the CPU even through the use of large clusters. In our comparison, we show that when properly implemented, the discrete Fourier transform on GPU can perform much faster than its CPU counterpart. The difference results in computational time which may make it economically more acceptable for use in industry. This speedup may allow for the implementation of the DFT in programs such as 5D interpolation or curvelet transforms without the need for local binning. We also note that due to unexpected memory operations, the prepackaged Thrust toolkit matrix operators cause significant slowdowns to the calculations.

ACKNOWLEDGMENTS

We thank the sponsors of CREWES for their continued support. This work was funded by CREWES industrial sponsors and NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 543578-19.

REFERENCES

Han, J., and Sharma, B., 2019, Learn CUDA Programming: Packt Publishing.

Nickolls, J., Buck, I., Garland, M., and Skadron, K., 2008, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?: Queue, **6**, No. 2, 40–53.
URL `https://doi.org/10.1145/1365490.1365500`

Trad, D., 2016, Five-dimensional interpolation: exploring different fourier operators: CREWES Research Report, **28**.

Wang, T., and Kemao, Q., 2017, GPU Acceleration for Optical Measurement: