Department of Informatics, University of Zürich

BSc Thesis

A GPU-enabled Single-Point Incremental Fourier Transform

Johann Schwabe

Matrikelnummer: 17-726-724

Email: johann.schwabe@uzh.ch

July 26, 2020

supervised by Prof. Dr. M. Böhlen and M. Saad

University of Zurich[UZH]

Department of Informatics

dedicated to Science

Acknowledgements

I would like to express my gratitude to my supervisor Muhammad Saad from whom I could learn much and who guided me through this thesis.

I would like to thank Prof. Dr. Michael H. Böhlen and the Database Technology Group for making this project possible.

I would like thank Dr. C. Schwabe for thoroughly reviewing the thesis and providing valuable feedback.

Abstract

Although Fourier transforms are widely used, special implementations are still being developed for high performance applications. For use in a streaming environment, the Single Point Incremental Fourier Transform (SPIFT) was recently proposed (Saad et al. 2020). In SPIFT the main computational bottleneck is the incremental addition, where the Single Point Fourier Transform of each new datapoint is integrated into the previous result. Two key optimizations to speed up SPIFT were introduced and tested. Firstly, GPUs are used to efficiently sum the Single Point Fourier Transforms of the individual datapoints to the final result. Secondly, Single Point Fourier Transform of the different datapoints with the same shift are combined using an aggregation matrix instead of individually being integrated into the resultant image. Five different implementations combining these two optimizations were evaluated. Both optimizations are crucial and are together able to increase the throughput on tested matrix dimensions by three orders of magnitude.

Zusammenfassung

Obwohl die Fourier-Transformation weit verbreitet ist, werden für Hochleistungsanwendungen noch immer spezielle Implementierungen entwickelt. Für den Einsatz in einer Streaming-Umgebung wurde kürzlich die Single Point Incremental Fourier Transform (SPIFT) vorgeschlagen (Saad et al. 2020). Bei SPIFT liegt der Hauptengpass bei der Berechnung des letzten Schritts, bei dem die Single Point Fourier Transformation jedes neuen Datenpunkts in das Zwischenergebnis integriert wird. Zwei wichtige Optimierungen zur Beschleunigung von SPIFT wurden eingeführt und getestet. Erstens werden GPUs verwendet, um die Single Point Fourier Transformationen der einzelnen Datenpunkte effizient zum Endergebnis zu summieren. Zweitens werden mehrere gleichartige Single Point Fourier Transformationen, die aus verschiedenen Datenpunkten berechnet wurden, in einer Aggregationsmatrix zusammengefasst, anstatt einzeln in das resultierende Bild integriert zu werden. Fünf verschiedene Implementierungen, die diese beiden Optimierungen kombinieren, wurden evaluiert. Beide Optimierungen sind entscheidend und können zusammen den Durchsatz um mehrere Größenordnungen erhöhen.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In radio astronomy, sky objects such as stars or galaxies are not observed directly. Instead an array of radio telescopes capture the incoming radio waves for 9 to 12 hours. Until now, sky images are only computed once the observation is finished and all data has been collected into a visibility grid. To transform this data into an image a standard 2D Fourier like the Fast Fourier transform (FFT) is optimal.

To improve the flexibility of the observations real-time viewing of the images is required. This can only be achieved in a data streaming approach where not only the final image is calculated but intermediate images are created from the incoming flow of data. However, in a streaming approach, computing the Fourier transform using the FFT become a computationally intensive operation. Thus a specialized Fourier Transform is needed. For this purpose, the Single Point Incremental Fourier Transform (SPIFT) was proposed (Saad et al. 2020). It can compute the Single Point Fourier Transform of each datapoint in (N). This shifts the bottleneck of the processing pipeline away from the Single Point Fourier Transforms of the individual datapoints to the summation of those. But each summation still takes (N^2) operations. Here it might be possible to harness the computational power of Graphical Processing Units (GPUs) as this is a parallelizable task in which GPUs thrive.

The goal of this bachelor thesis is to measure to which degree the processing capabilities of GPUs can be used to speed up the Single Point Incremental Fourier Transform.

The key performance measurement of SPIFT in a streaming environment is its throughput. This throughput can be defined by a) the maximum speed of the data stream it can handle and b) the rate at which it can provide output. To optimize SPIFT, not only can the results from the Single Point Fourier Transform be summed efficiently by GPUs but also the number of updates needed can be reduced. This is done by combining Single Point Fourier Transforms calculated from different datapoints in an aggregation matrix instead of individually integrating them into the result image. This reduces the computational load on the algorithm and therefor significantly increases the maximum speed of the data stream it can handle.

To verify the advantages of a GPU based implementation and test the aggregation matrix, four GPU based approaches were compared to a CPU based implementation. The calculations were executed with picture resolutions between 1024 and 8192 in x and y on test datasets of 2^{20} 2^{28} datapoints. The results are compared in terms of throughput as defined by a and b and the different approaches are discussed.

2 Background

2.1 Fast Fourier Transform vs Single Point Incremental Fourier Transform

For many applications that need Fourier transformations the Fast Fourier Transform (FFT) is the best choice as its asymptotic complexity of $O(nlog(n))$ (Cooley and Tukey 1965) is the lowest with n being the size of the 1D input vector. FFT can only be used to transform one-dimensional data. A modification leads to the 2D-FFT which transforms two-dimensional data, as it is needed for telescopic data. Another minor modification leads into the inverse 2D-FFT that is needed to transform the frequency spectrum to the result image. The resulting algorithm still has a complexity of $O(n \ log(n))$ with n being the number of datapoints in the 2D input matrix.

Performing a FFT for every datapoint, as needed in a streaming enviroment, is computationally very intensive. For the specific case, where it is not need to recalculate the whole image, but only update it with the newest datapoint a specialized Fourier transform was designed: SPIFT. Most other Fourier Implementations, including 2D-FFT, require static data of fixed size to perform the transformation. This means that the time while the data is collected the algorithm is either idling and the time is wasted or completely recalculating the image for every datapoint using all previous datapoints. Not so with the SPIFT: It incrementally incorporates the datapoints as they come from the telescope. As the algorithm starts with the first datapoint the picture is building up right from the beginning and intermediate results can be extracted at any point during the execution. The algorithm finishes after the last datapoint is measured and incorporated.

2.2 SPIFT

In the implemented SPIFT, an image is calculated from a stream of datapoints. Each datapoint is a triple $hu; v; visi$: The u and v coordinates together describe the distance of the telescopes and their angle to the observed source and a complex visibility vis encodes phase and amplitude information of the measured radio wave. The stream of incoming triples is unordered. By calculating the Fourier transform of all datapoints individually and then combining them the image in form of a matrix of complex numbers is calculated.

The whole process can be simplified to the single equation: $I_t = I_{t\ 1} + vis_t \ \ L^{u_t;v_t}$. I_t is the state of the result matrix at time t, vis_t is the complex visibility of the datapoint at position u_t and v_t. $L^{u_t;v_t}$ is the twiddle factor matrix, a matrix of the same dimension as I_t consisting of the twiddle factors for the coordinates u_t and v_t. The twiddle factor at position

j, k in the twiddle factor matrix for the datapoint at u, v can be calculated in two ways: $W^{u\,j+v\,k} = e^{(i\ 2\ (u\,j+v\,k)=N)}$ or $W^{u\,j+v\,k} = cos(2\ (u\,j+v\,k)=N) + i\ sin(2\ (u\,j+v\,k)=N)$. The twiddle factors are complex numbers on the unit circle in the complex plain and there are only N distinct twiddle factors. These N twiddle factors are repeated in each twiddle factor matrix. Thus a significant reduction in runtime complexity was made by initially calculating all N twiddle factors and then only combining them into the individual twiddle factor matrices. By exploiting the periodicity of the twiddle factors within a twiddle factor matrix, not even the construction of the whole twiddle factor matrix is needed anymore. A twiddle factor matrix can be defined by a twiddle factor vector, a shift type and a shift index. From these three parameters the twiddle factor matrix can be recreated by copying the twiddle factor vector into the first row in the twiddle factor matrix, then circularly shift the twiddle factor vector by the shift index and copying this shifted vector into the second row in the twiddle factor matrix. This continues until the whole matrix is populated. This procedure for the shift type "row shift" can be easily adapted to "column shift" by filling the twiddle factor matrix column wise and shifting column wise. An example of a twiddle factor matrix is given in Table 2.1.

Table 2.1: The 8 x 8 twiddle factor matrix $L^{2;3}$. Shift type: "row shift", shift index: 3, shift vector: row 0

	0	1	2	3	4	5	6	7
0	W^0	W^3	W^6	W^1	W^4	W^7	W^2	W^5
1	W^2	W^5	W^0	W^3	W^6	W^1	W^4	W^7
2	W^4	W^7	W^2	W^5	W^0	W^3	W^6	W^1
3	W^6	W^1	W^4	W^7	W^2	W^5	W^0	W^3
4	W^0	W^3	W^6	W^1	W^4	W^7	W^2	W^5
5	W^2	W^5	W^0	W^3	W^6	W^1	W^4	W^7
6	W^4	W^7	W^2	W^5	W^0	W^3	W^6	W^1
7	W^6	W^1	W^4	W^7	W^2	W^5	W^0	W^3

The above described update of the image is broken into four steps that are performed sequentially for each datapoint. Each of these steps only depends on the previous steps for this datapoint and is independent of all other datapoints.
The four steps are:

1. Calculating the shift type
2. Calculating the shift index
3. Calculating the shift vector
4. Updating the previous image

Step 1: Shift type

Calculating the shift type is simple: it is column shift if one of these three conditions hold:

v is 0

u is odd and v is even

v is even and a power of two

Otherwise it is row shift. Algorithm 1 shows the pseudo code for calculating the shift type.

Algorithm 1: isRowShift(u_t, v_t) (Saad et al. 2020)

return !((v_t== 0) or (u_t% 2 == 1 and v_t% 2 == 0) or
(v_t% 2 == 0 and gcd(u_t,N) < gcd(v_t,N)))

Step 2: Shift index

The calculation of the shift index depends upon the shift type. For a row shift, the shift index is the smallest positive integer j where $j \quad v_k \mod$ the matrix dimension equals u_k. Similarly, for column shift the shift index is the smallest positive integer j where $j \quad u_k \mod$ the matrix dimension equals v_k. In the worst case all numbers up to the matrix dimension -1 have to be tested and thus this step has an asymptotic complexity of $O(N)$. It can be proven, that for every coordinate with $u_k \ \& \ v_k$ smaller than the matrix dimension a shift index can be found. This computation is shown in the pseudo code in Algorithm 2.

Algorithm 2: ShiftIndex(u_t, v_t, isRowSh) (Saad et al. 2020)

```
if (u_t or v_t == 0) then
    return 0;
p = -1 ;
if (isRowSh) then
    for k=0 to N do
        if u_t == k v_t % N then
            p = k ;
            break;
else
    for j=0 to N do
        if v_t == j u_t % N then
            p = j ;
            break;
return p;
```

Step 3: Shift vector

To calculate the shift vector the shift type and the shift index has to be known. The shift vector is computed by multiplying the complex visibility of the datapoint with the twiddle factor vector. For this the twiddle factor at each index position is fetched from the set of the precomputed twiddle factors and is then multiplied with the visibility of the datapoint. The correct twiddle factor for the k-th datapoint at the j-th position in the shift vector is $j \quad v_k \mod N$, if the shift type is row shift or $j \quad u_k \mod N$ for column shift. Like this the shift vector can be computed in N complex multiplications and thus the asymptotic complexity is (N). Algorithm 3 shows how this can be implemented in pseudo code.

Algorithm 3: ComputeVector(u_t, v_t, vis_t, isRowSh, N) (Saad et al. 2020)

```
for k=0 to N do
    if (isRowSh) then
        q[k] = vis_t  W^{k v_t %N}
    else
        q[k] = vis_t  W^{k u_t %N}
return q;
```

Step 4: Updating the previous image

The shift type, index vector from step the previous steps implicitly define the second part of the update equation: $vis_t \quad L^{u_t,v_t}$, the Single Point Fourier Transform. Now in this step the previous image has to be incremented by the matrix defined by the Single Point Fourier Transform. A pseudo code implementation of this is given in Algorithm 4.

Algorithm 4: IncUpdate($I_{t\ 1}$, q, isRowSh, p, N) (Saad et al. 2020)

```
if isRowSh then
    for j = 0 to N do
        startIdx = (p  j) %N;
        for k = 0 to N do
            idx = (startIdx + k) %N;
            I_t[j,k] = I_{t 1}[j,k] + q[idx];
else
    for k = 0 to N do
        idx = (p  k) %N;
        for j = 0 to rows do
            idx = (startIdx + j) %N;
            I_t[j,k] = I_{t 1}[j,k] + q[idx];
return I_t;
```

Asymptotic complexities of the four steps

The first step can be accomplished in asymptotically constant time. The second and third step have an asymptotic complexity of (N) with N being the matrix dimension. The fourth step has an asymptotic complexity of (N^2) and thus it is the main bottleneck.

As the first three steps are completely independent of the current state of the result or any other datapoint, a very efficient parallelization could be possible. Even the fourth step, updating the matrix, can be parallelized as the update consists of summing the result matrix with a matrix defined by the shift vector. As matrix summation is commutative and associative, the order of the updates is irrelevant, and the updates can be grouped into different matrices that are later combined. Therefor the algorithm to be implemented focuses on reducing the bottleneck of the fourth step by optimizing and parallelizing it.

2.3 Comparing CPU and GPU

The CPU is the centre of every computer system. It is responsible of executing the operating system and the programs the user requests. These are rather few processes. To allow for a pleasant experience the system needs to be very responsive and thus these few tasks of the CPU need to be executed fast. Modern computers have one CPU which consists of several relatively independent cores. Typical desktop processors have about 4 – 12 cores. In servers the CPU is also responsible for executing parts of the processing pipeline. The parts, that are best handled by the CPU, are those that are not dividable into multiple subproblems and thus need to be executed by a single fast core. To be able to execute several of these parts in parallel, modern servers have between one and four CPUs with each up to about 64 cores. Most modern CPUs support hyperthreading which means that each CPU core can execute 2 threads almost in parallel. While the most powerful systems can handle 512 threads in parallel, we limited our research to more widely used server capacities of around 40 cores.

In comparison modern GPUs have up to 128 streaming multiprocessors (SM). Each SM consists of 32 compute unites, called cuda cores, and thus is able to execute 32 threads in parallel. These parallelly executed threads are called a warp. GPUs switch between warps to minimize idling time resulting from slow memory access. Context switching is very fast as there are registers for several warps on chip. This means that there always are multiple warps ready for execution on a SM. Whenever one is waiting for a slow memory access, the SM can switch warp and continue executing a different warp until the requested data arrives. As all the information of this new warp is already loaded into the registers the switch has minimal overhead. There is space in the registers for up to 64 warps per SM, but this is also dependent on the number of registers needed per thread. This results in up to 4096 cuda cores in a single GPU. To increase the GPU based parallelism even further there can be multiple GPUs in a system.

A combination of both CPU and GPU is needed to efficiently execute the various different tasks of a computer from system kernel execution to image rendering. While CPUs are strong in executing a branching set of instructions, GPUs excel in parallelism. This difference is reflected in the hardware used for these compute units. CPUs and GPUs use a different number

of different compute cores with each different instruction sets and different memory.

CPU cores are optimized to reduce latency to make the system responsive. This results in very fast cores (2 – 5Ghz). To reduce the latency even more, a big part of each core is dedicated to branch prediction. Branch prediction is the process of a core taking a guess what it should do next while waiting for the decision on what it really should do. If the guess was correct, the branch is accepted, and the execution continues at the final point of this branch. Otherwise the branch is ignored, and the correct branch is calculated. As the branch prediction is done during idle time of the processor, no performance is lost but potentially time is gained. The prediction is important to execute a branching set of instructions very fast.

For GPUs branch prediction doesn't make sense as there are very few branches in GPU code. Additionally, when they execute code they are always under full load. Parts of the GPU never idle while others are working, because the tasks for the GPU are those that can be split into many subproblems and spread onto all parts of the GPU. For example, while waiting for data, GPUs can very efficiently context switch and process another thread. With these many multi thread focused cores a GPU can efficiently process big chunks of data, while CPUs excel at predicting the path of a single thread and thus executing it very fast.

2.4 GPU Thread model

As the main point of GPUs is parallel execution of threads, understanding the thread model is key. On the highest level the GPU is a single processing unit. It consists of streaming multiprocessors (SM) that can each execute a warp of 32 threads in parallel. When launching code on a GPU this code has to be within a special function called 'kernel functions'. The key difference in behaviour of kernels compared to normal functions is the automatic multithreading. When launching a kernel not only the arguments are passed in but also the parallelisation information defining on how many threads the kernel should be launched. This is done by specifying a grid of blocks, where each block contains a fixed number of threads. The number of threads in a block is defined in the launching parameter blockDim. The parameter blockDim consists of three integers (x,y,z) specifying a three dimensional matrix of threads. Each block is executed on a single SM in warps of 32 threads. These 32 threads in a warp are then executed truly parallel, meaning that each thread executes the same command at the same time. The only exception is the case when the threads within a warp branch. Then one part of the warp idles while the other threads execute the branch. This is very inefficient and has to be considered when writing kernels. Which 32 threads of a block are grouped into a warp is chosen by the SM and cannot be predicted or influenced.

A key design decision when writing GPU code is the number of threads per block. Obviously, the number should be dividable by 32 as otherwise a warp with less than 32 threads will be executed which leads to idle time in the SM. The maximum number of threads per block is 1024. When using too few threads per block the SM overhead for scheduling and switching blocks becomes significant. Less than 64 is not recommended (NVIDIA Corporation 2020). The optimal number depends on the problem and can only be found by testing. Depending on the number of threads per block, the number of blocks required to launch the total number of threads needed can be calculated. These blocks are then ordered into a three-dimensional grid

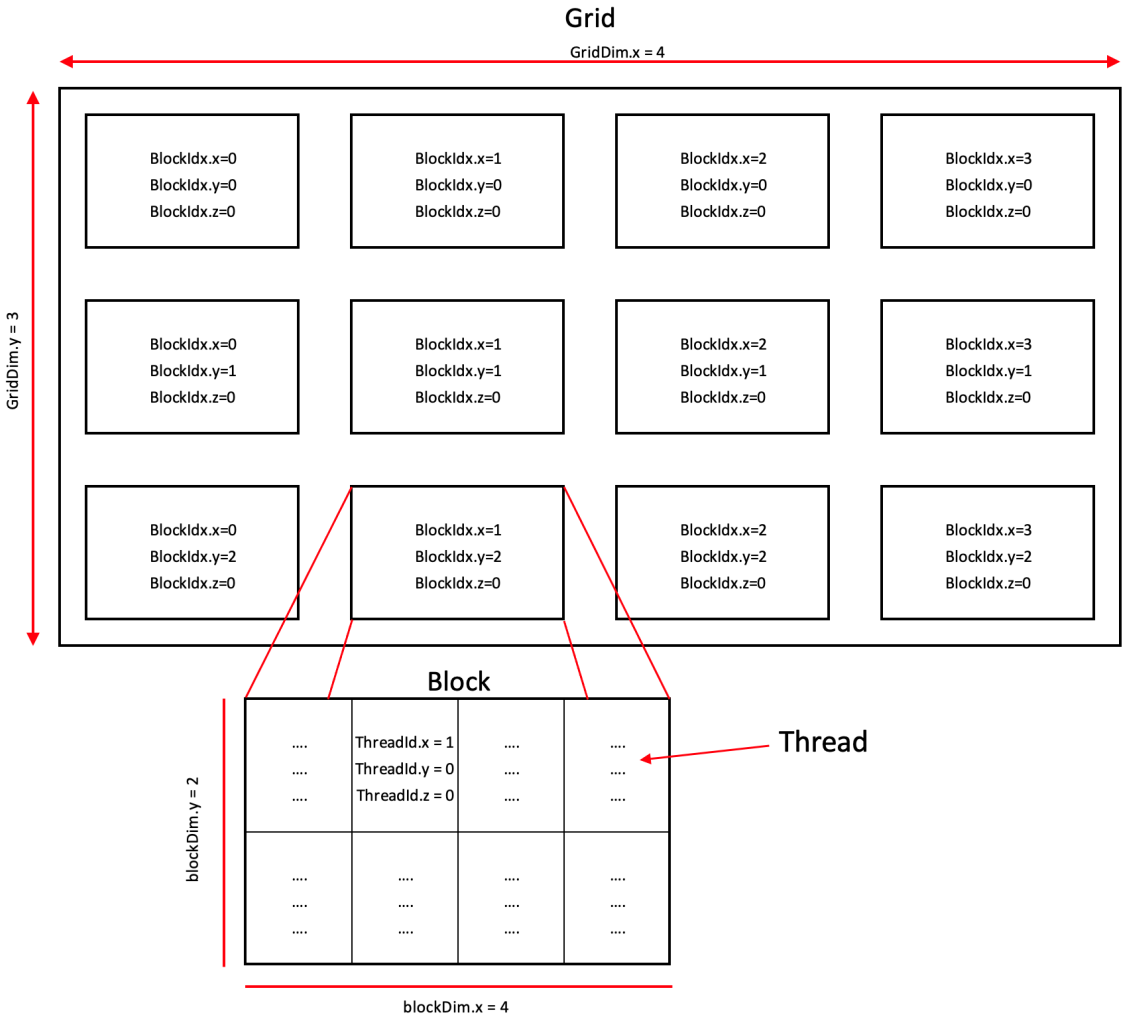

Figure 2.1: Illustrating the GPU-Thread model

defined by the launch parameter gridDim. How the blocks are ordered in the grid does not matter for the GPU, but it makes sense to order them resembling the problem or data structure at hand. To differentiate the threads and instruct them to execute the same code but with different data each thread has a variable threadIdx that contains its coordinates within the block and a variable blockIdx that contains the coordinates of its block within the grid. Additionally, the gridDim and blockDim can be accessed. An example of such a grid of blocks is given in Figure 2.1.

2.5 GPU Memory model

The major part of the GPU memory is the global memory. Compute focused datacentre GPUs can have up to 40GB of global memory. It can be accessed from every thread and even from the CPU. But it has the big trade-off of being slow as it is accessed by many threads in parallel. Additionally, it is not located on the GPU processing chip but around it which leads to higher access latencies. By following access patterns, the throughput can be significantly improved compared to random accesses. For example, threads can be organized within a block in a way that they access data from a consecutive block of memory. Then all individual memory access requests can be coalesced into a single or very few requests. As the memory bandwidth of modern GPUs is very wide, a request for a big block of memory takes the same amount of time than a request for a small block of memory. Depending on the GPU there are also other efficient patterns. A subset of the global memory can be declared as texture memory. This significantly increases the access speed, as this memory is read only and thus can be efficiently cached in the texture cache, which is fast. Additionally, there is special hardware in the GPU that accelerates often used texture access patterns.

The rest of a GPU's accessible memory consists of constant memory, shared memory, registers and local memory. Constant memory is independent of the global memory but can also be accessed by all threads. Similar to texture memory it is read only and cached in its own constant cache. But it is limited to 64KB and misses the texture specific access features of the texture memory. Shared memory is memory that can only be accessed by threads of the same block. Thus it can be physically closer to the individual cores and has a latency roughly 100x lower that uncached global memory access (Harris 2013). Also, its access bandwidth is much faster than global memory. Each thread also has its own memory, called registers, that only itself can access. Registers are the fastest memory, but they are very small with only 64k 32-bit registers for each SM. These registers are then split between all threads that are currently being processed on the SM. If the registers are too small, parts of the data will be moved to the local memory. Local memory is a segment of the global memory that can only be accessed by the current thread. But as it is not within the SM, it's significantly slower (NVIDIA Corporation 2020).

2.6 Example: Matrix Product

The standard example to demonstrate the GPU thread and memory model is the matrix product. Let's name the two input matrices A and B and give them a dimension of 32x16 and 16x32. This will result in a matrix R with dimension 32x32. Using 2D arrays on the GPU is not recommended as this means that for every lookup memory is accessed twice. Thus the arrays are flattened to a 1D array by appending the rows. The smallest independent task in this problem is calculating a single position of the result matrix and thus should be chosen as the task for a single thread. This means that 1024 threads are needed to calculate the matrix product.

In the simplest solution, the matrices A, B and R are saved into global memory. Now the 1024 threads have to be assigned into blocks. One could launch a single block with all the threads. But then the whole execution would happen within a single SM and all other SM would be idling. Alternatively, one could group a column of R, 32 threads, into a block. This means the blockDim is 32x1x1 and the gridDim is 32x1x1. Now, during kernel execution each thread has to get its first factor from A, multiply it with the first factor from B and add it to the product of the second factors and so on. This results in 16 global memory accesses from A, 16 from B and one write to R. As 1024 threads execute this sequence global memory is accessed 263168 times. This is not efficient.

To reduce the global memory accesses shared memory can be utilized. Accessing data from shared memory is still slow but orders of magnitude faster than from global memory. In the above model all threads of the same block access the same values from matrix B. This is good as these values could be initially loaded into shared memory which would severely reduce the number of global memory accesses. But every thread accesses different values of matrix A, so those can't be loaded into shared memory efficiently. By grouping threads differently into blocks the amount of different positions accessed in the matrices A and B by each block can be reduced. By grouping the result matrix into 16 squares (4x4) of 8x8 blocks (illustrated in Figure 2.2), each block has to load only 8 rows from matrix A and 8 columns from matrix B. This means launching the kernel with a blockDim of 8x8x1 and a gridDim of 4x4x1. Each block firstly loads the parts needed from matrix A and B into local memory. Then the all threads within this block have to be synchronized to ensure that all the data was read into shared memory. Then each thread calculates the vector-product and then writes its result back to the result matrix. This results in 5120 global memory reads which is significantly more efficient than the above approach. An additional advantage of this grouping is that the requests to load data from global memory might be coalescence. Coalescence is what happens when multiple threads in the same warp access adjacent values from global memory at the same time. As the bandwidth between shared memory and global memory is wide, the coalescence request, loading a chunk of data, takes the same time as an individual request, loading a single datapoint. This further reduces the number of global memory accesses.

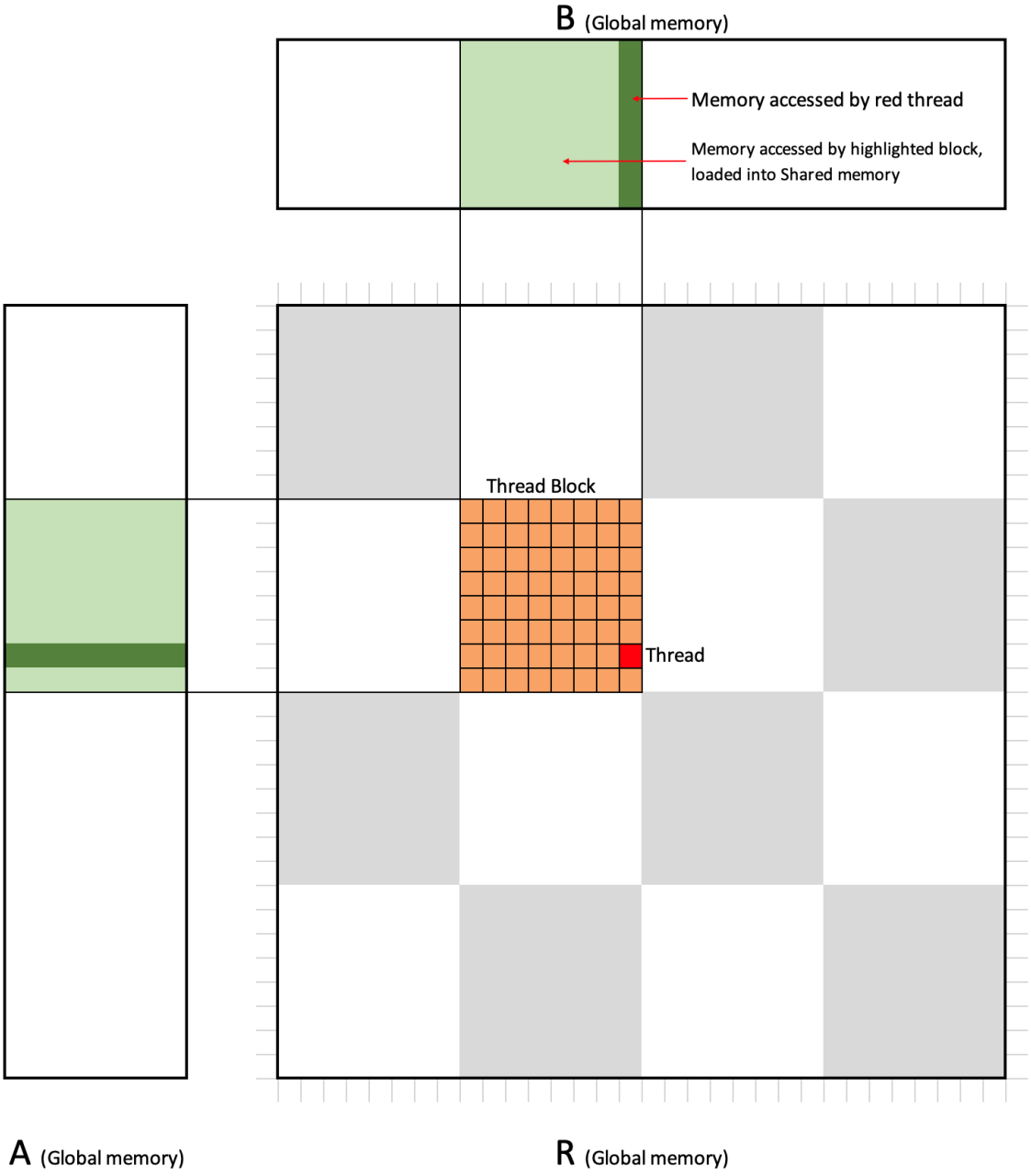

Figure 2.2: Final thread model of the matrix product

3 Experimental Setup

3.1 Hardware

The algorithm was implemented and tested on a modern compute focused server containing the following components:

Table 3.1: Description of the testing server

Component	Version
CPU	2x Intel Xeon Gold 6230 20 Core / 40 Thread @ 3.90GHz
Motherboard	Super X11DPG-OT-CPU
Memory	385610MiB ECC DDR4-2933
GPU	6x ASPEED NVIDIA GeForce RTX 2080 Ti
GPU driver	NVIDIA 418.113
Operating System	Debian GNU/Linux 10 (buster) x86_64
Kernel	4.19.0-9-amd64

The server consisted of a dual CPU system containing two Intel Xeon Gold 6230 20 Core / 40 Thread CPUs. 385GB of DDR4-2933 memory were present. As GPUs six NVIDIA RTX 2080ti-s were used. As the operating system the Linux distribution Debian was used.

3.2 Software

The implementations were done using C++ 17 with the CUDA API. Cuda is an API developed by NVIDIA to efficiently execute code on their GPUs. Initially it was an extension for the programming language C, but since the introduction of the NVIDIA Fermi-Architecture in 2010 it also supports C++ (NVIDIA Corporation 2020). Contrary to similar APIs like OpenCL, Cuda only runs on NVIDIA GPUs and not on AMD or Intel GPUs. Many higher level languages like Python or MATLAB have wrapped Cuda and are thus also able to execute code on GPUs. But to get the maximum performance and flexibility, C++ is the best option as it allows for a better optimization and thus faster execution on supported cards (Karimi, Dickson and Hamze, 2010).

3.3 Dataset

Test data was used to check the correctness of the algorithm and to compare run times of different versions of the algorithm. To verify the correctness of the algorithms several random

input images with different dimensions were transformed into frequency space once using MATLABs 2D-FFT. These pairs of original and transformed images were then used to verify the correctness of each algorithm. This was done by transforming the transformed input image back to the time space using the currently developed SPIFT and comparing each point of the computed image with the original image. To accommodate for rounding errors a threshold of 10^{-4} was set for the absolute difference between every position in the original and computed image.

To test the performance of the algorithms test data was randomized before every execution. Since the visibility at a certain position does not change the flow of execution it was held constant. Only the positions of the datapoint were randomized. To generate random datapoints the function std::rand() was used. The datapoint was scaled into the correct range using the modulo operation. The data sets are not perfectly random as std::rand() does not provide truly random figures and the scaling using the modulo operation can show a bias towards certain values. But since better random generators are more computationally intensive and thus not practical for large datasets, the std::rand() function was decided to be good enough for this use case. Matrices with the dimensions 2^{10} x 2^{10}; 2^{11} x 2^{11}; 2^{12} x 2^{12} and 2^{13} x 2^{13} were tested. Each entry in the matrix consists of two 32bit floats representing a single complex number. This results in matrices with sizes between 8.4MB and 537MB. The random order of generated coordinates of the datapoints in the test data accurately resembles the order of the coordinates in a real data stream as neither shows any significant bias towards any datapoints.

3.4 Experimental Setup

All tests were executed on the same server with exactly the described components. The cuda-code was compiled with the cuda compiler “nvcc”, whereas the CPU-only implementation was compiled with “gcc”. The optimization flag was always set to “-O3” to maximise the compiler’s optimization of the code.

To find the best implementation of the algorithm four different versions using GPUs were implemented and tested. To provide for a baseline for evaluating the performance gain through the use of GPUs, a CPU based fifth version was implemented and compared.

3.5 Evaluation of data

To evaluate the performance of the five algorithms the execution times were measured in different configurations. The total execution time of an algorithm only captures an overview over the efficiency of the algorithm. To get a more detailed view on the performance and identify bottlenecks 20 different parameters describing the algorithm were measured. The most important are total execution time, time while stream was active, time to finish up the calculations, combining matrices and dividing by number of updates, average time per row & column shift update, time to calculate the shift type, shift index and shift vector and number of updates done.

Each algorithm was tested in different configurations. For these configurations, different

matrix-dimensions, numbers of GPUs available and numbers of CPU-threads processing data were combined. Each permutation was tested once and consistency with similar combinations was checked.

4 Results

4.1 General implementation structure

All algorithms follow the same basic structure. A group of threads, termed "Fourier threads", read the datapoint coming from the data stream and perform a Single Point Fourier Transform on them. Then they pass their result to socalled "update threads". These threads then integrate this Single Point Fourier Transform into the result matrix. Two main optimizations of the SPIFT algorithm were made in all implementations: (1) reduce the time it takes to update the result matrix, (2) reduce the number of updates needed. (1) was done by parallelizing this step on two levels. Multiple updates for different datapoints were processed in parallel. This is possible as the result matrix is the sum of all the Single Point Fourier Transforms. Thus the Single Point Fourier Transforms can be grouped and these groups summed individually. These groups can be combined at the end into the complete result in (N^2) steps. This allows for multiple update threads to completely independently update their own result matrices. A second level of parallelism was achieved when using GPUs to update the result matrices. GPUs can handle many threads and thus can update many parts of their result matrix in parallel. (2) was done by exploiting the fact that if two datapoints have the same shift type and index they can be unified in (N) operations into one shift vector and undergo the fourth step as one update. By efficiently combining many shift vectors the number of updates needed can be significantly reduced.

Five different approaches were implemented that differ in various parts of the algorithm. They all try to implement these optimization as efficiently as possible. But as they basically implement the same algorithm, they share many parts of the code, especially the kernels, as shown in Table 4.1 below. Of course the CPU-based approach doesn't use kernels.

Table 4.1: Comparison of the five algorithms regarding their use of common kernels

Implementation	Uses Update Kernels	Uses SumResults	Uses DivideResult
Unsynchronized approach	Yes	Yes	Yes
Block update approach	Yes	Yes	Yes
GPU parallel approach	different gridDim	No	Yes
Queued approach	modified version	Yes	Yes
CPU based approach	No	No	No

4.2 Kernels

4.2.1 General kernel information

The main task of the GPU is clearly step four of SPIFT: given a vector, a shift and a shift type update the GPUs result matrix. As the procedure for the row shift is significantly different to the procedure for the column update the algorithm was split into two separate kernels, one for each shift type. Another part of the algorithm that can be efficiently calculated is the combination of the result matrices after all datapoints were incorporated into them. Finally, the result matrix has to be divided by the number of datapoints processed. This means that 4 kernels are needed in these approaches:

updateWithRowShift

updateWithColumnShift

divideResult

sumResults

4.2.2 Update Kernels

The smallest independent subproblem is updating a single datapoint of the result matrix. This means retrieving the current state of the result matrix, adding the correct value from the shift vector and writing back this sum. Letting each thread execute this has several disadvantages.

Firstly, a lot of threads are needed, which means a lot of scheduling overhead.

Secondly each thread has to individually calculate which position it has to update in the result matrix and which position in the shift vector it needs. An efficient approach would group the positions sensibly, then it would be enough to calculate the position only once and then derive the positions for the other datapoints from the first.

Thirdly there is no control in which order a SM executes the threads within a block. Thus, the chance of multiple threads in a warp accessing adjacent data at the same time is low. This means that there might be little memory access coalescence.

To solve these problems the result matrix was divided into squares of 128x128 which are updated by a block of 128 threads. Other block dimensions were tested as well (Figure 4.2). Kernels with a blockDim between 32 and 512 performed slightly worse. Each thread updates a column from the result matrix. By letting the threads update columns the threads access adjacent datapoints in the result matrix at the same time and thus allowing for efficient coalescing.

As each datapoint of the result matrix is read exactly once and written once, no optimization using shared memory is possible here. Also, for the shift vector, optimization using shared memory is not possible. Even though multiple threads in the same block might access the same shift positions, this is very difficult to predict. And probably not all positions in the

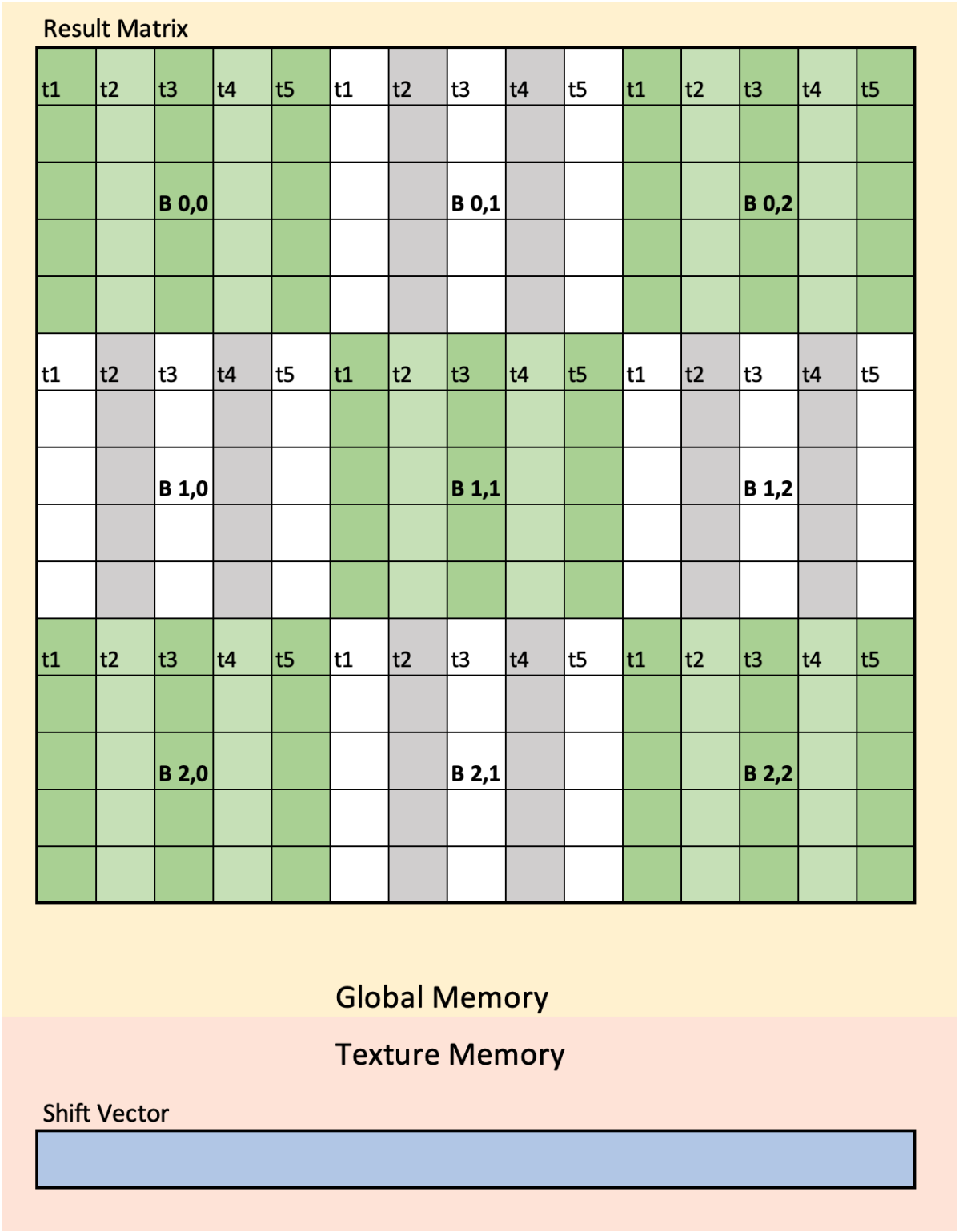

Figure 4.1: Miniature thread and memory model with a blockDim of 5x1x1 and a gridDim of 3x3x1

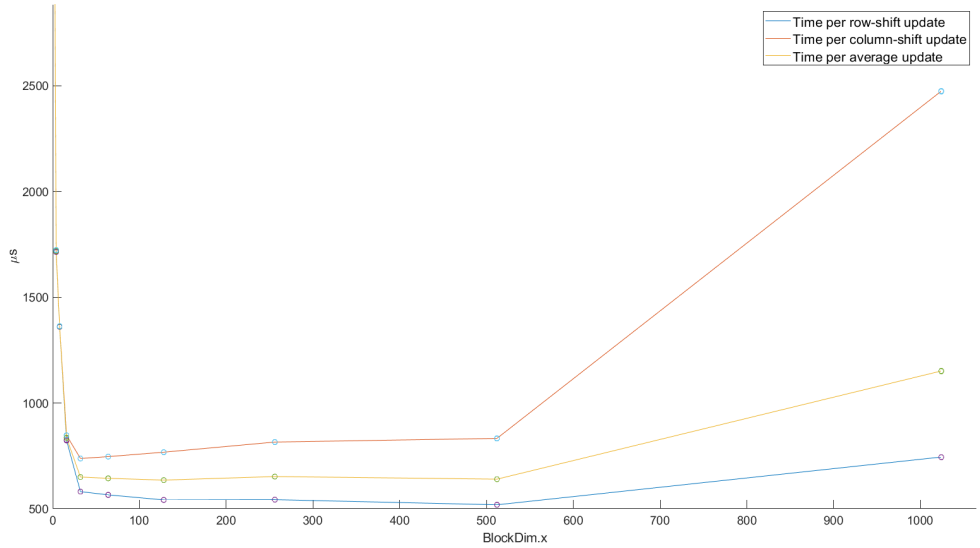

Figure 4.2: Influence of the blockDim on the execution time for both update kernels. Example using 50 Fourier threads, a result matrix of 4096x4096, 6 GPUs and 2048^2 datapoints

shift vector are read in a block either, thus loading the whole vector is not efficient. As the shift vector is constant over a single update it could be moved to constant memory. As constant memory is of limited size, this is not a solution that scales to larger matrices. Therefore texture memory was chosen as it fulfills two important properties: it is (1) independently cached, thus very fast and (2) optimized for fast non-coalescented access as it is processed using specialized shader hardware. This significantly improved performance compared to a shift vector in global memory. Figure 4.1 shows a miniature version of how the threads are grouped out and how the different memory types are used.

4.2.3 Row Shift Kernel

To compute a row-shift update, the kernel gets the pointer to its result matrix, called `dev_matrix` in the code below, the pointer to the shift vector, `dev_vector`, the matrix dimension, `matrix_dim` and the shift index, shift, passed in. Information about the threads' coordinates within the block and the blocks' coordinates within the grid are available in the variables `threadIdx` and `blockIdx` by default. Additionally, the block dimension and grid dimension are available in `blockDim` and `gridDim` by default.

Firstly, each thread calculates the index of the first datapoint in the result matrix that it updates. This index is then saved in the integer variable `start_index`. Secondly each thread calculates the starting index in the shift vector, which is then saved into the integer `vectorPos`. Having these two values, the indices of all datapoints needed for different positions can be derived efficiently.

After calculating these initial indices, each thread loops over the positions it has to update. Their number does not have to depend on the `blockDim` as in this implementation. As long as enough threads are launched in total to saturate the GPU it does not affect performance. For simplicity the number of datapoints updated per thread is equal to the number of threads in a block in this implementation.

Within the loop, two operations are needed: updating the datapoint and updating the indices. As the result matrix is a standard 1D matrix of complex numbers, accessing them follows the normal C++ array access pattern. As the shift vectors are saved into texture memory, they are accessed using the texture function `tex2D`. Because no `tex1D` function exists, the y-coordinate passed in, is always set to 0. Additionally, the datatype accessed has to be passed in as a template parameter. Textures do not support complex numbers, therefore two consecutive floats are implicitly grouped together. Thus, the `vectorPos` has to be multiplied by two to get the real part and multiplied by two and +1 for the imaginary part. To improve coalescence, it might be more efficient to group all real parts of the shift vector and all imaginary parts into separate blocks. This improvement would only work for row-shift and not for column-shift updates as only there the shift vector is accessed sequentially. But due to the caching of the texture data, this improvement would probably be minimal. Updating the result matrix index is simply done by incrementing it by the result matrix dimension. Similarly, the `vectorPos` is incremented by the shift. But afterward the modulo with the result matrix dimension has to be taken to ensure the circularity of the shift vector. The Cuda/C++ implementation of this kernel is given in Algorithm 5.

```
__global__ void updateWithRowShift(cuFloatComplex  dev_matrix, \
            cudaTextureObject_t dev_vector, int matrix_dim, int shift)
{
  int start_index = blockIdx.x   blockDim.x   matrix_dim + \
                    blockIdx.y   blockDim.x + threadIdx.x;
  int vectorPos = (shift   blockIdx.x   blockDim.x + threadIdx.x + blockIdx.y   blockDim.x)\
                  % matrix_dim;
  for (int i = 0; i < blockDim.x; i++) {
    dev_matrix[start_index + i   matrix_dim].x += tex2D<float>(dev_vector, 2 vectorPos,0);
    dev_matrix[start_index + i   matrix_dim].y += tex2D<float>(dev_vector, 2 vectorPos+1,0);
    vectorPos += shift;
    vectorPos %= matrix_dim;
  }
}
```

Algorithm 5: Row shift update kernel

4.2.4 Column Shift Kernel

In its structure the column shift kernel is very similar to the row shift kernel. Firstly, the starting index of the matrix, start_index, and the starting index of the vector, vectorPos, are calculated. The calculation of `vectorPos` is different than in the row-shift kernel whereas the calculation of the `start_index` is identical to the row-shift kernels. Then the threads loop over the `dev_matrix` and update the datapoints. In the column-shift kernel an improvement was made by first loading the datapoint to be updated into a local variable, then update the real and imaginary part on the local variable before writing it back to the global

matrix. This change only improved the performance in the column-shift kernel. After updating a datapoint the indices are updated similar to the row-shift kernel. Again the Cuda/C++ implementation of this kernel is given in Algorithm 6.

```
__global__ void updateWithColumnShift(cuFloatComplex  dev_matrix, \
            cudaTextureObject_t dev_vector, int matrix_dim, int shift)
{
    int start_index = blockIdx.y  blockDim.x  matrix_dim + \
                      blockIdx.x  blockDim.x + threadIdx.x;
    int vectorPos = (shift  (threadIdx.x + blockDim.x  blockIdx.x) + blockIdx.y  blockDim.x) \
                    % matrix_dim;
    cuFloatComplex dev_matrix_point;
    for (int i = 0; i < blockDim.x; i++) {
        dev_matrix_point = dev_matrix[start_index + i  matrix_dim];
        dev_matrix_point.x += tex2D<float>(dev_vector, 2  vectorPos,0);
        dev_matrix_point.y += tex2D<float>(dev_vector, 2  vectorPos+1,0);
        dev_matrix[start_index + i  matrix_dim] = dev_matrix_point;
        vectorPos++;
        vectorPos %= matrix_dim;
    }
}
```

Algorithm 6: Row shift update kernel

4.2.5 SumResults & DivideResult

After all datapoints are combined the different result matrices from the GPUs have to be combined. After the combination each position in the result matrix has to be divided by the total number of datapoints processed. As both of these actions can be efficiently separated into independent subproblems, they can be executed on GPUs. But as they are only executed once, their performance is not as crucial as the row & column shift kernels.

The SumResults kernel takes pointers to two result matrices and the matrix dimension. To access the two result matrices, they both have to be in the GPU's global memory. This means one of them has to be moved from a different GPU to the current one. The block and grid dimensions are exactly the same as in the row & column shift kernels. Also, the iteration of each thread over the matrix is the same. But instead of updating each datapoint of one matrix with a value from the shift vector, the value is incremented by the value of the corresponding datapoint from the other matrix. This kernel only combines two matrices. To combine more matrices, the kernel has to be executed multiple times with the different matrices.

The divideResult kernel is even simpler. It also only needs the pointer to the result matrix and the number of datapoints processed. The thread and block dimensions and the iteration of each thread are again the same. The only difference lies within the loop. Here each datapoint is divided by the number of datapoints processed and then written back.

4.3 Implementations

4.3.1 Unsynchronized approach

In this implementation the Fourier threads are executed by the CPU and the update threads on the GPU. This means that any number of CPU threads each independently read the next

datapoint in the stream, calculate its shift type, shift index and shift vector. Then the thread checks if there already exists a vector with the same type and index in a shared aggregation matrix, waiting to be integrated into the result. If there is, the two vectors are be combined. Else the new one is added to the waiting list. On the other side the update threads on the GPU threads go through all possible shift types and indices and check if there is a vector with them waiting to be processed. If there is, it is integrated into their individual result matrix and removed from the shared aggregation matrix. After the last datapoint is read, processed and integrated into a result matrix, the result matrices are combined. A schemata of this approach is given in Figure 4.3

The big advantage of this approach is the very strong independence between the different steps as there are only 3 points of synchronization: the Fourier threads read from the same stream, the Fourier and the update threads read & write to the same aggregation matrix and the update threads need to aggregate the results at the end. To prevent race-conditions when multiple threads access the aggregation matrix every entry is mutex protected. As there are N possible shift indices and 2 different shift types, the chance of a collision is low and thus little efficiency should be lost to synchronization.

A disadvantage of this approach is that the data is not being processed in any order. So, if an intermediate result has to be calculated, the whole process needs to be stopped in order to ensure that all datapoints that are already read are integrated. This means that the approach is not truly stream processing.

4.3.2 Block update

This approach is very similar to the unsynchronized one, except that the GPUs are not constantly updating the matrix. In fixed intervals of numbers of processed datapoints the reading of data is stopped and all aggregated points are integrated into the result. Then the Fourier threads start reading and processing again. Reducing the number of updates should reduce the load on the GPU. The execution times for the Block update approach are highly similar to the unsynchronized approach in regard to all time components. The only difference lies within the number of updates, as these can be set in the block update approach as a launch parameter. Thus no further data was collected.

4.3.3 GPU Parallel updates

In this approach a different aspect of the algorithm is optimized. In all other cases, each update thread is updating its own result matrix. But here the result matrix is split among the different update threads and each update is split and each part processed on a GPU. This reduces the memory needed on each GPU and could increase performance as the smaller matrix parts might be cached more efficiently. Also, the combination of the results in the end only takes constant time. A schemata of this approach is given in Figure 4.4.

The GPU parallel update approach is not a feasible approach as testing showed a negative correlation between the number of GPUs and the number of updates. Thus, it is less or equally efficient as the unsynchronized approach in almost all parameters. The only benefit of parallel

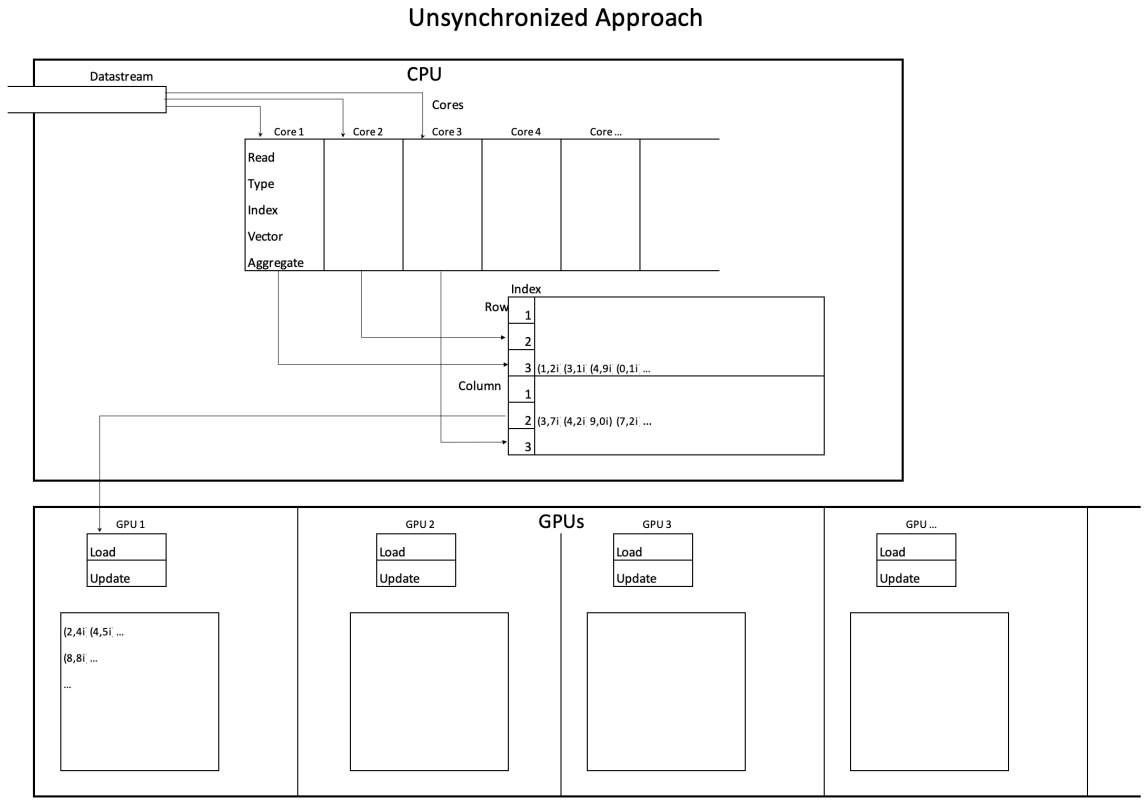

Figure 4.3: Processing steps in the unsynchronized approach

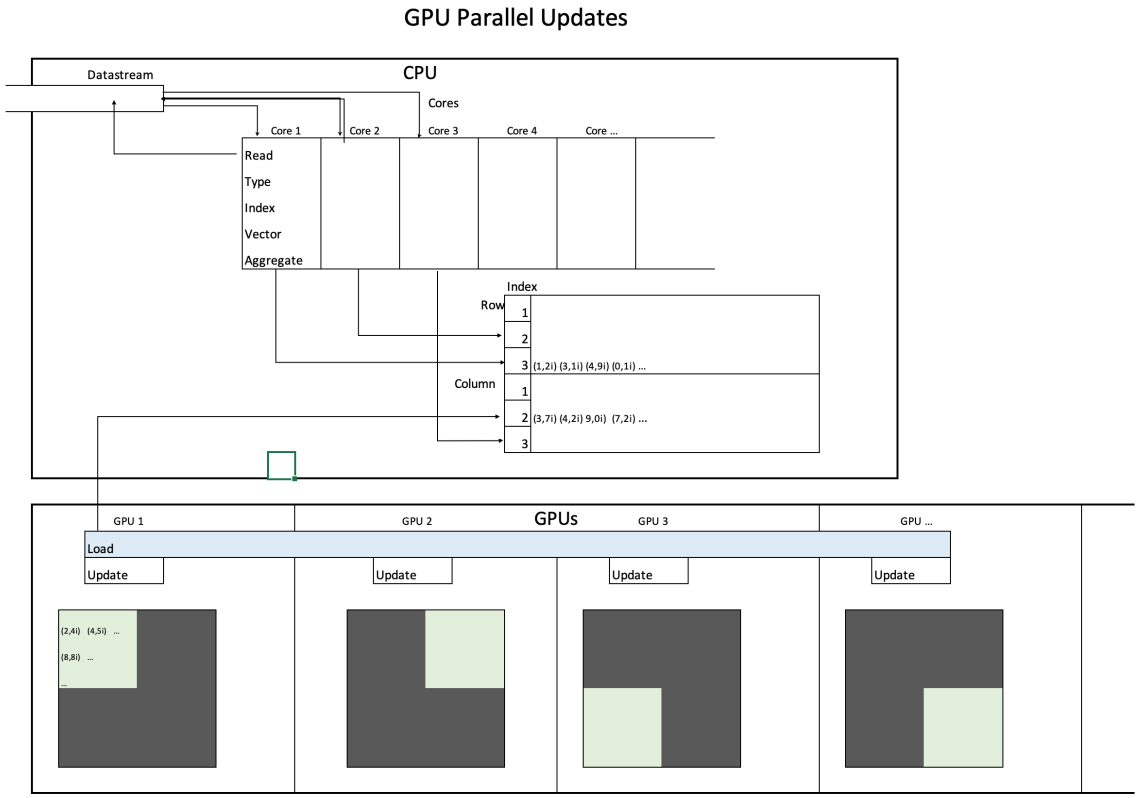

Figure 4.4: Processing steps in the GPU Parallel Updates approach

GPU updates is a reduction in final processing time. This reduction did not influence the total execution time enough to be significant. Thus this approach was abandoned.

4.3.4 Queued Approach

This approach tries to address the problem that the algorithm does not truly represent stream processing as the order the datapoints are processed does not equal the order they are read from the stream. In this approach the Fourier threads only process the datapoints but do not deal with the aggregation part. After they process the datapoint, its three results, the shift type, the shift index and the shift vector are just enqueued into a shared queue. The GPU-threads then dequeue the next triple and linearly search a fixed number of triple in the queue if any can be aggregated with the first one. This ensures that the datapoints are roughly processed in the same order as they are read. Also, the combination of multiple shift vectors can be efficiently performed on a GPU, which reduces the load on the CPU. A drawback might be the inefficient search in the queue for similar datapoints. A schemata of this approach is given in Figure 4.3. The Queued approach was found infeasible as even prototype implementations were not able to produce results comparable to the unsynchronized approach.

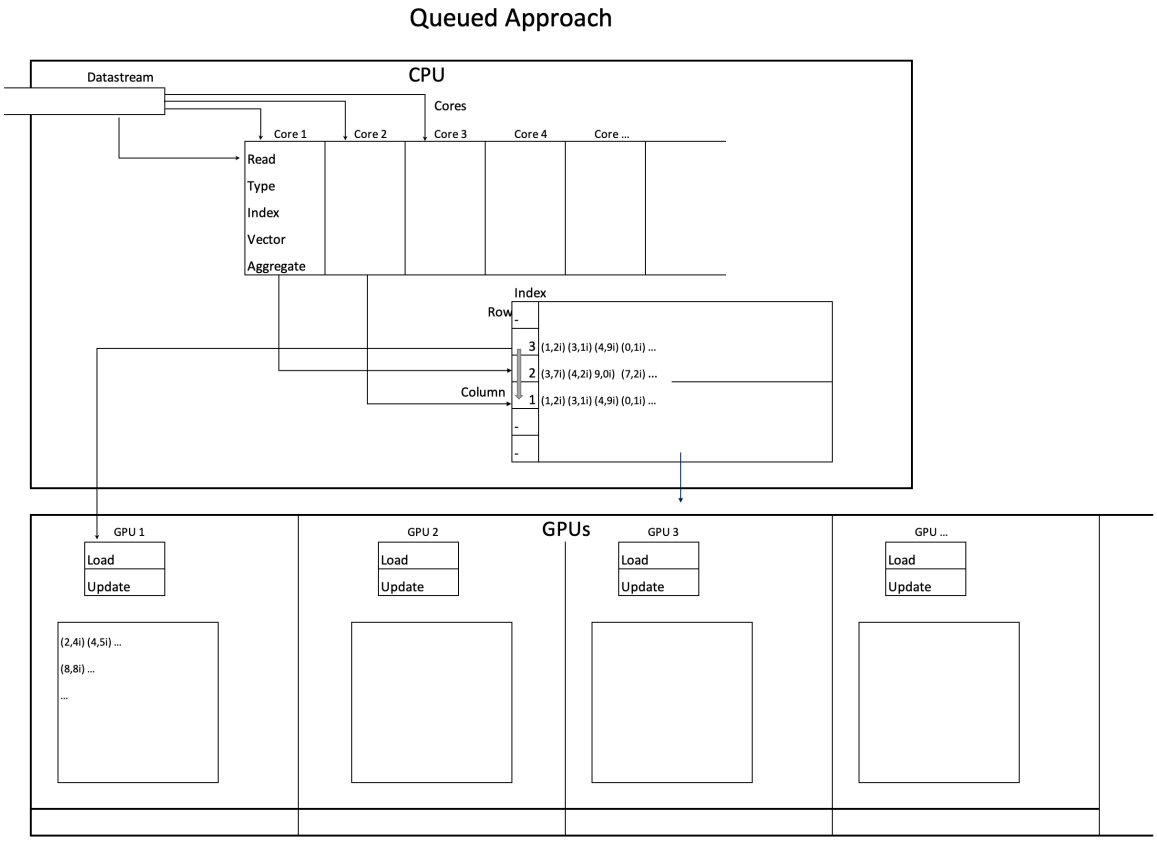

Figure 4.5: Processing steps in the Queued approach

4.3.5 CPU based Implementation

This implementation follows the unsynchronized approach with the only difference being that the update threads are executed on the CPU.

4.4 Execution times

4.4.1 Overview

To quantify the performance gain by using GPUs, the different algorithms have to be compared. As the performance of each algorithm depends on its launch configuration, the best configuration for each algorithm has to be found first. For this the algorithms were tested in various configurations and the results presented below. After presenting the effects of the different configurations, the algorithms are compared against each other in the next section.

For these tests the data was randomly generated before the start of execution. In all tests 2048^2 datapoints were processed. On the y-axis the execution parameters of the individual tests are displayed. The first defines the number of Fourier threads, the second the matrix dimension and the third the number of update threads. On the x-axis the execution time in microseconds is displayed. The **total execution time** of the computation, the red bar, is subdivided into three consecutive sections:

> **Time while reading**: The green bar displays time while the stream is active and new datapoints are coming.
>
> **Time after reading**: The blue bar displays the time after the stream has ended when only the update threads are active.
>
> **Time for final processing**: The cyan bar displays the time after all datapoints are integrated into the result matrices and these matrices have to be combined into a single matrix and this combined matrix has to then be element wise divided by the number of datapoints.

Even though the total execution time consists of only these three components, they often do not add up to the total time. This occurs as the measurement for the time after reading measures the duration each update thread takes to finish their part of the remaining updates. These durations are then averaged to the time after reading. But the relevant duration for the total execution time is not the average duration the update threads take but the longest duration. In this sense the total execution time measures the actual time for the whole algorithm whereas the time after reading measures the minimal duration for the time after reading that could be achieved by further optimizing the final updates.

4.4.2 CPU-based approach

The execution time of the CPU-based approach depends on three parameters: the number of threads reading the datapoints and doing the processing steps 1-3, called “Fourier threads”, the

dimension of the matrix and the number of threads updating the result matrix, called “update threads”.

When keeping the number of update threads and the matrix dimension constant increasing the number of Fourier threads consistently decreased the total execution time. Even though the decrease is significant the reduction is not linear to the number of Fourier threads. The number of Fourier threads only influences the time to read all datapoints and does not influence the time after reading or the time for the final update. This is illustrated in Figure 4.6.

When keeping the number of Fourier threads and the matrix dimension constant increasing the number of update threads significantly reduced the total execution time. This reduction resulted from a reduced time while reading and a reduced time after reading. The increase in the number of update threads not only has diminishing returns on the reduction of the time while reading but even shows an increase in read time for more than 20 update threads.

4.4.3 Unsynchronized approach

The execution time of the unsynchronized approach depends on the same three parameters: number of Fourier threads, matrix dimension and number of update threads (= number of GPUs). Again, the number of Fourier threads is negatively correlated to the execution time. This increase in the number of Fourier threads has diminishing returns. For most matrix dimension the time after reading and the time for final processing is insignificantly small. Only for matrix dimensions of 8192 and greater they become significant. These findings are illustrated in Figure 4.8.

When keeping the number of Fourier threads and the matrix dimension constant an increase in the number of update threads (GPUs) leads to a reduction in total execution. This reduction can be best observed at higher matrix dimensions. There the data shows that mainly the time after processing is reduced but also the time while reading decreases. An increase in the time for final processing can be measured, but the time for final processing remains marginal in comparison to the total execution time.

4.4.4 Unsynchronized approach vs CPU based approach

These two approaches are very similar regarding their implementation, thus their results can be compared directly. Even though execution times are an important measurement, similarly important is the time per updates of the result matrices on a single GPU. With this measurement the true power of GPUs can be directly shown, whereas the execution times in both approaches are also heavily dependent on the CPU. As data clearly shows, the GPUs are orders of magnitude faster than the CPUs when it comes to time per update. Form these two approaches the average execution per datapoint can be calculated and also the average execution time per datapoint in primitive approaches where no aggregation matrix was used and the result matrix updated for every datapoint. In Table 4.2 it can be clearly seen that the aggregation matrix reduces the average time per datapoint by orders of magnitude.

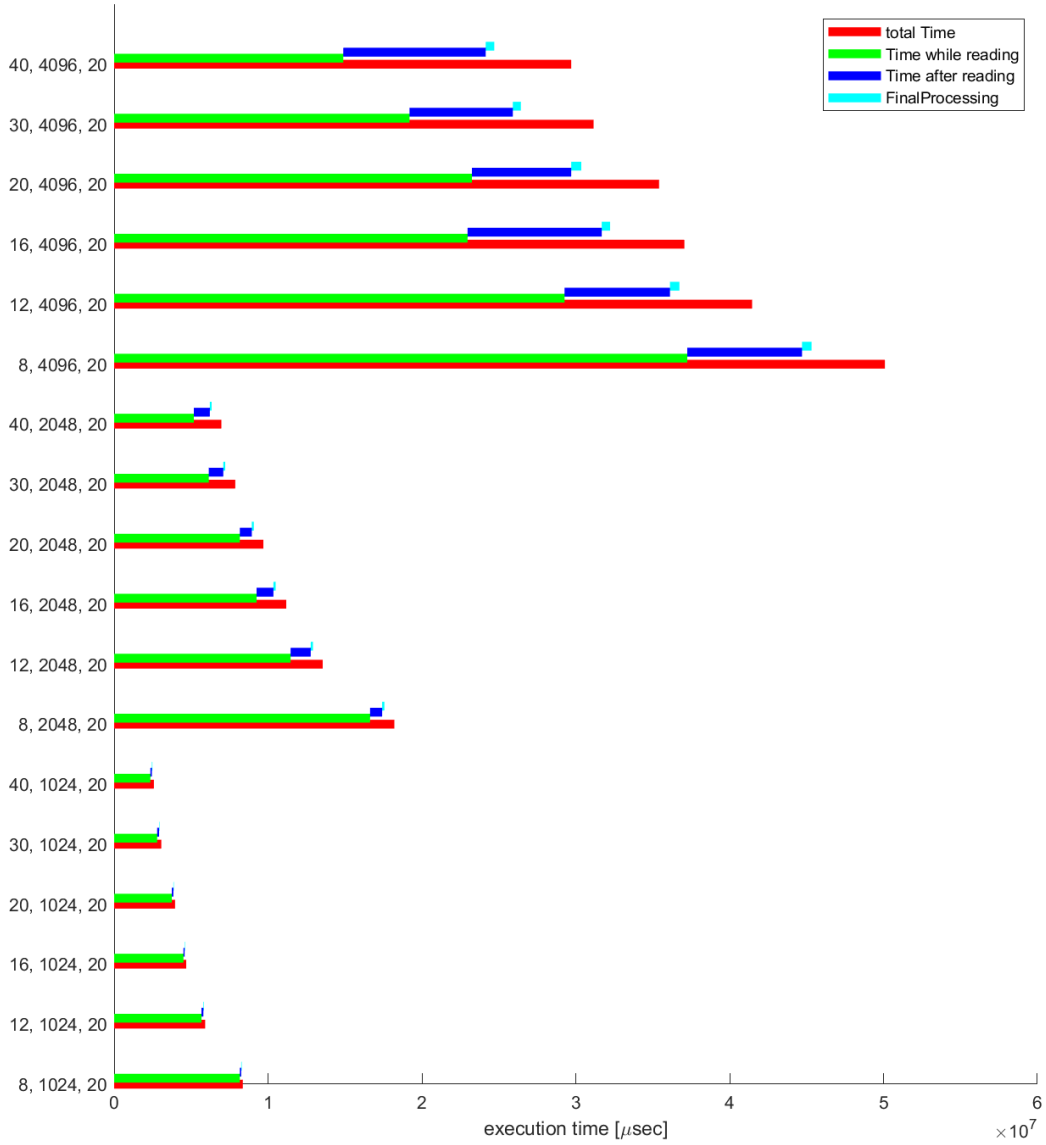

Figure 4.6: Influence of the number of threads processing steps 1-3 on execution time of the CPU based approach

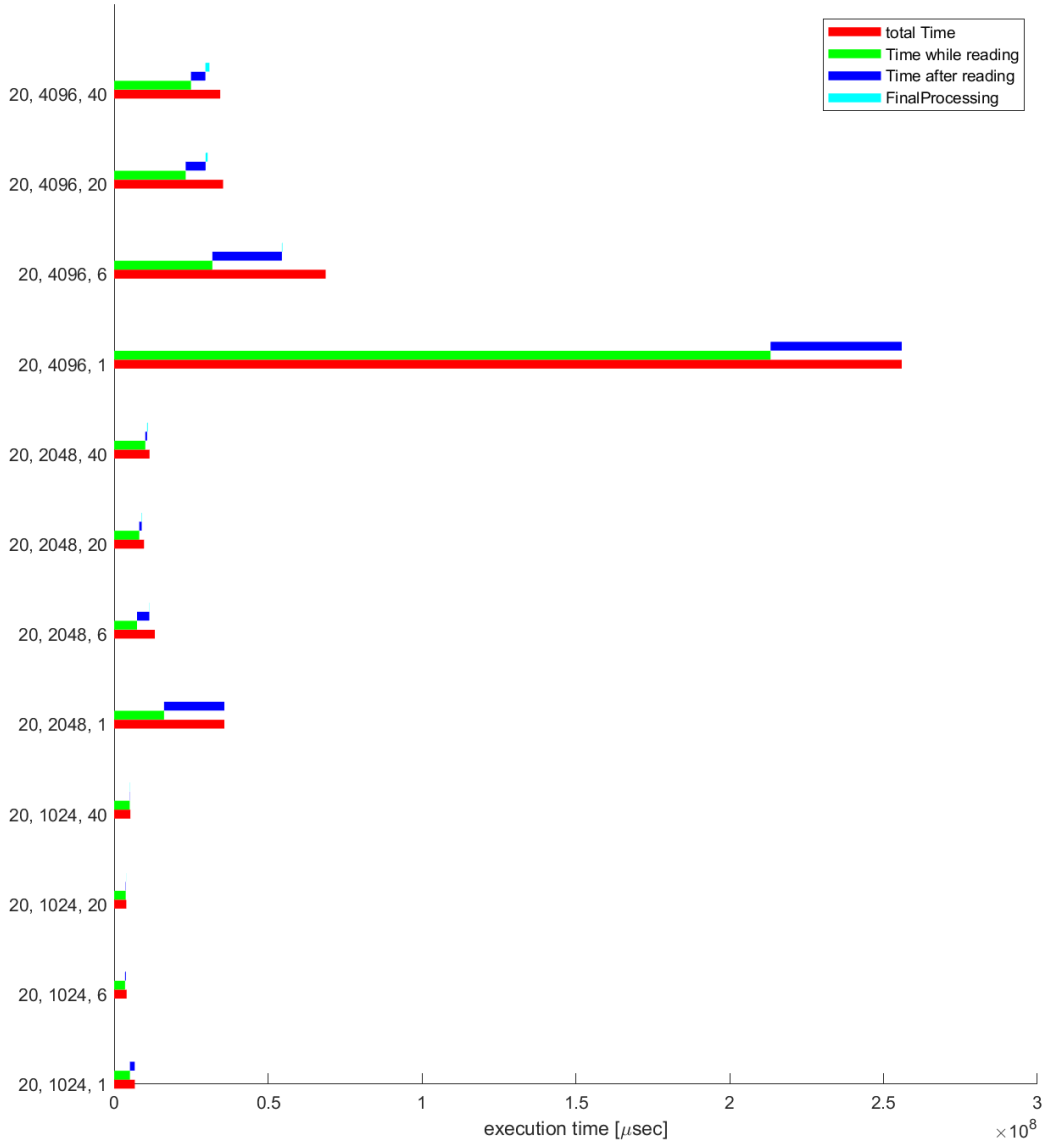

Figure 4.7: Influence of the number of threads updating the matrices on the execution time in the CPU-only approach

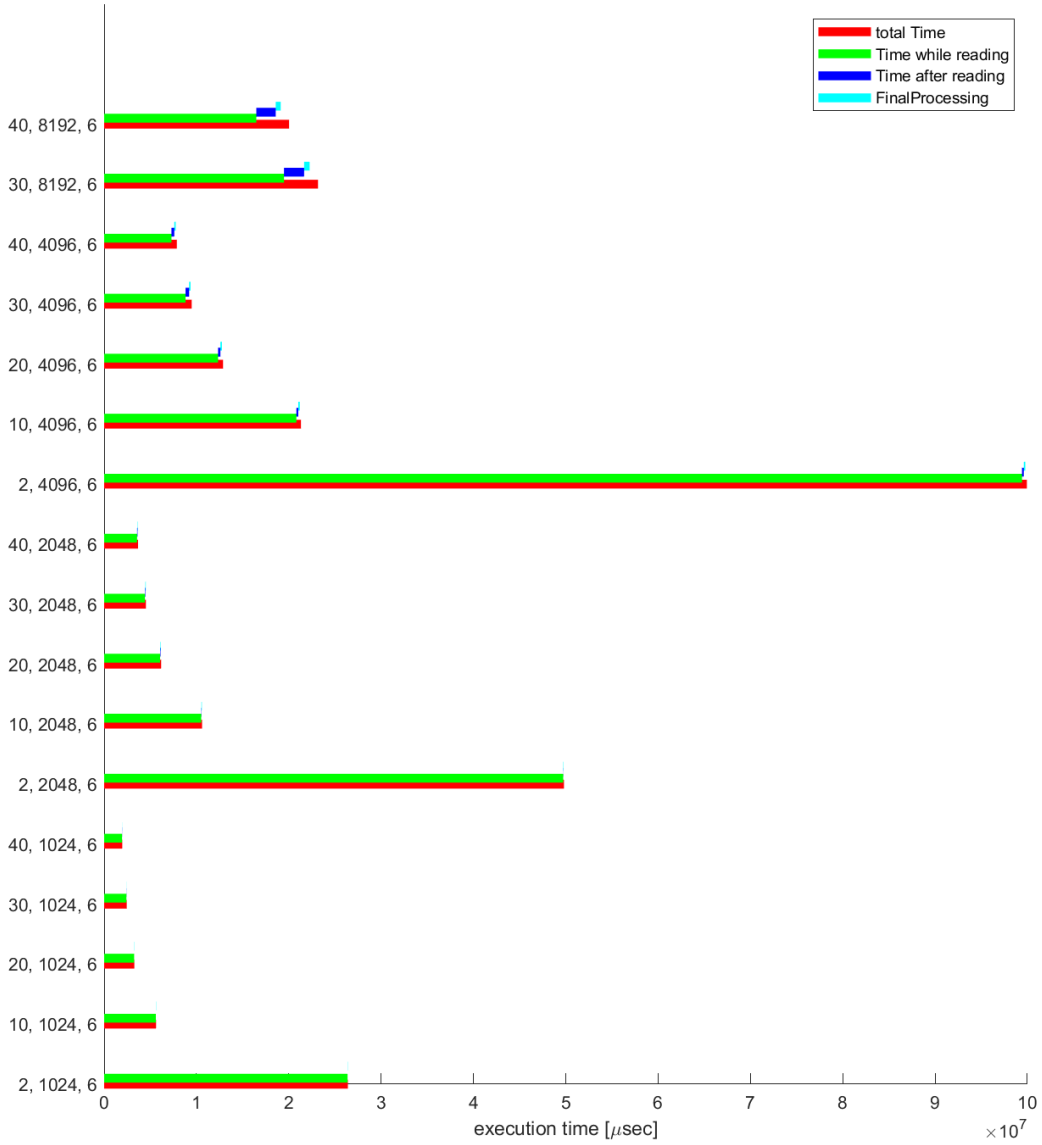

Figure 4.8: Influence of the number of threads processing steps 1-3 on execution time of the Unsynchronized approach

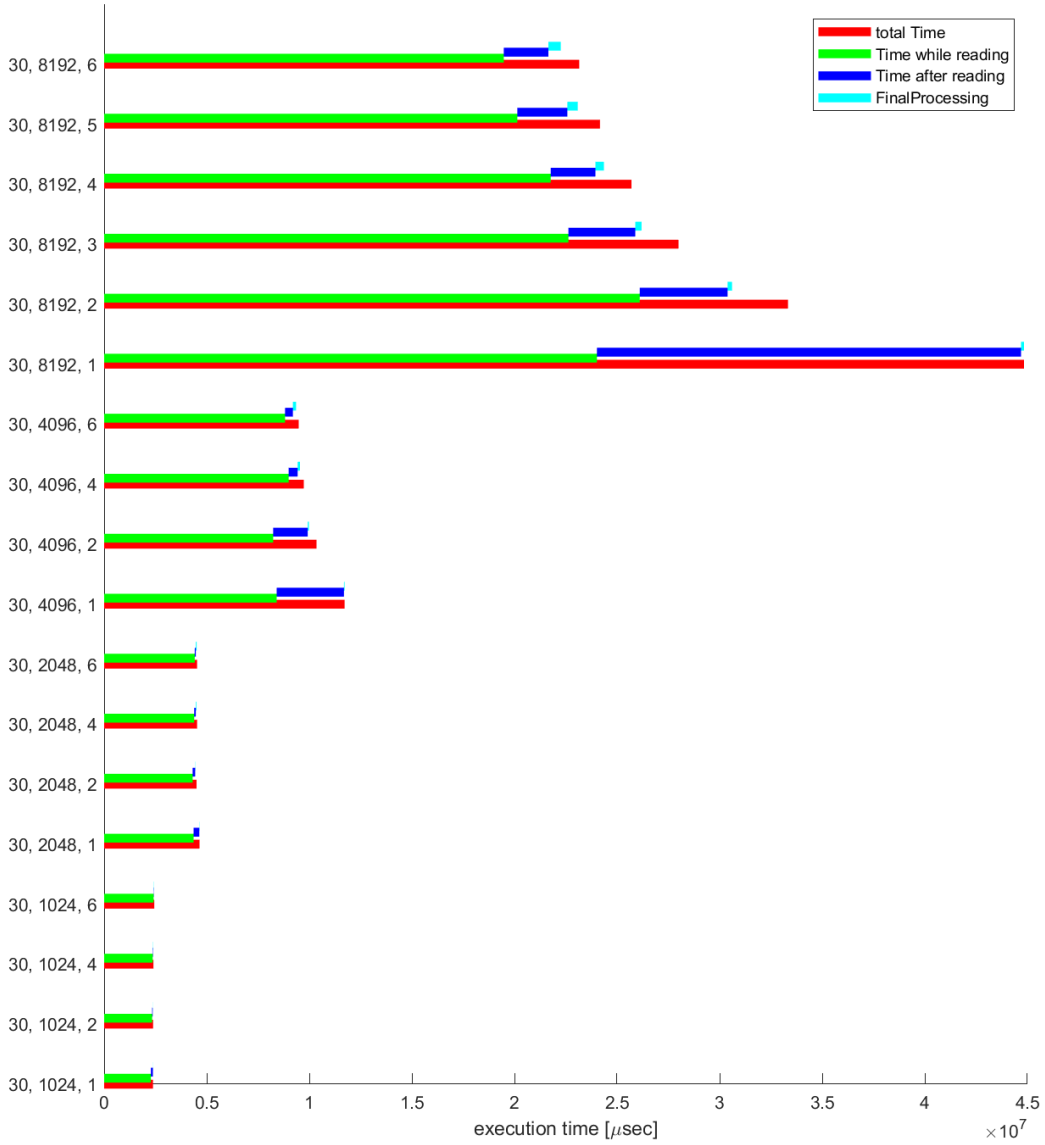

Figure 4.9: Influence of the number of GPUs updating the matrices on the execution time in the Unsynchronized approach

Table 4.2: Compute time per datapoint on different matrix dimensions using different implementations

Implementation \ matrix dimension	1024	2048	4096
Primitive CPU based implementation	3701 s	14430 s	57123 s
Primitive GPU based implementation	76 s	182 s	665 s
CPU based implementation	18 s	43 s	143 s
GPU based implementation	5 s	10 s	20 s

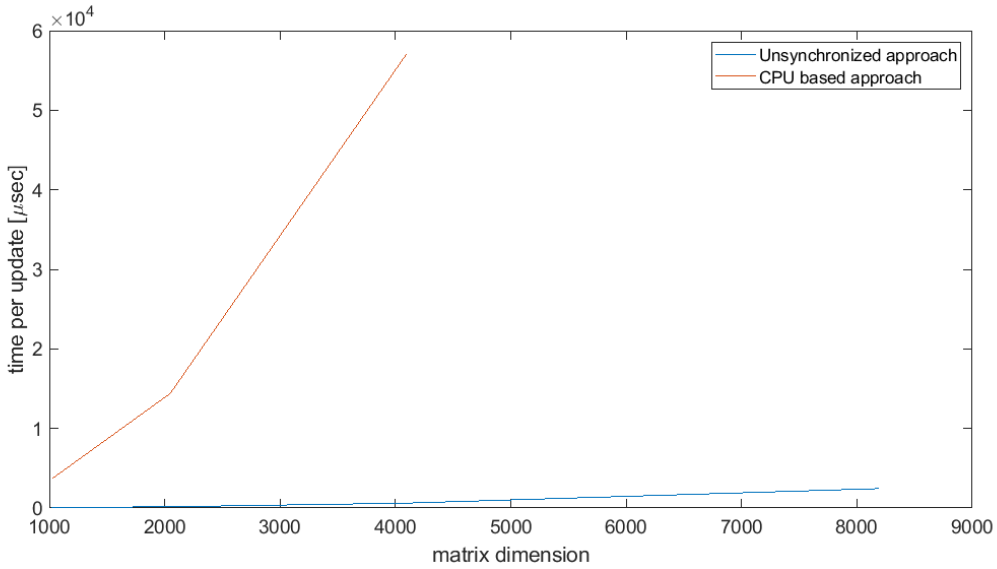

Figure 4.10: Time per matrix update in CPU based approach and unsynchronized approach

5 Discussion

5.1 Fourier transforms in a streaming environment

Fourier transforms fall into two different categories. The standard approaches optimize their execution times, but they rely on having static set of data before starting the calculations. The fastest of these standard approaches is the Fast Fourier transform FFT with an asymptotic complexity of $(N \quad log(N))$. The second cluster enables to integrate the Fourier transform into a streaming environment. In a streaming environment the Fourier transform processes each datapoint individually and combines the results. aim is only to recompute thus They just have to be fast enough to keep up with the incoming data and continuously produce results in the form of intermediate images. Thus, their execution time is not their main objective, but they are optimized for throughput. SPIFT was designed to be used in a streaming environment and therefore belongs to the latter group.

To measure the throughput of our SPIFT implementations they were given the complete dataset which they read linearly. This simulates the maximal speed of an incoming data stream the implementation could handle. The time until all datapoints of the given dataset are processed results in an execution time. From this execution time the throughput can be calculated. The execution time consists of two main parts: (1) The calculations done while the simulated stream is active and (2) concluding calculation after the stream has ended. By dividing (1) by the number of datapoints in the dataset the throughput can be calculated. Thus, the crucial part of the algorithm is part (1). In all experiments the number of datapoints was held constant and thus part (1) correlates inversely to the throughput. Part (2) can be assumed to be independent of the number of datapoints read. Therefore, it becomes insignificant with long data streams. As part (2) becomes insignificant, the main optimizations of the algorithm focus on part (1).

In the original SPIFT algorithm (N) are needed to compute a Single Point Fourier Transform and (N^2) to update the result matrix with every new datapoint. Therefor the bottleneck was identified as the updating of the result matrix. To accelerate the updating process, GPUs were taken into consideration because matrix addition can be efficiently parallelized.

With the introduction of the aggregation matrix into the algorithm, the result matrix didn't need to be updated after every datapoint. This shifts the main computational load to the processing of the individual Fourier transforms and thus to the CPU. Therefore, the CPU has to be fast enough to keep up with the stream of data whereas the processes run on the GPUs don't influence the throughput. Thus, GPUs cannot be used to accelerate the processing of the incoming data.

An intermediate image is calculated based on the current state of the result matrices. As there is a delay between the processing of a datapoint on the CPU and its integration into a result matrix, the intermediate images don't represent the complete state of the data stream at

the time of their generation. To minimize this delay, the result matrices have to be updated as often as possible. A minimum of 2N updates are needed, but up to the number of datapoints is theoretically possible. 2N updates would mean that all updates take place after the stream has finished. Thus, no intermediate images can be calculated, and in this case the FFT can do the same faster. To generate a close to real time representation of the data, which is important in many applications, the rate of updates has to be maximized. This is limited by the computational power of the GPUs.

5.2 Approaches

In the SPIFT algorithm as proposed in (Saad et al. 2020) every datapoint has to be transformed individually. The first part of the calculation, the Single Point Fourier Transform, has an asymptotic complexity of (N). The second part, updating the result matrix, has an asymptotic complexity of (N^2). On the tested CPU each update thread can perform an update in around 57000 s on a matrix with a dimension of 4096. This is orders of magnitudes slower than the ca 60 s a Fourier thread takes to perform step 1. Thus around 950 CPU threads would be needed to keep up with every Fourier thread. When performing the updates with GPUs, the time per update is 550 s on a GPU. Thus, 10 GPUs would be needed to for every Fourier thread. Exaggerations on this scale are very inaccurate, but the estimates still show the order of magnitude the real values lie in. As the time per update increases quadratically with the matrix dimension, this approach does not scale.

Thus, the introduction of the aggregation matrix is crucial for the performance of the algorithm. It shifts the load of N^2 complex summations on the update threads to an additional N complex summations on the Fourier threads. This is done by letting the Fourier thread combine their shift vectors (size N) of two datapoints with the same shift type and index instead of having the update threads twice update the result matrices (size N^2). This combined shift vector behaves as if it came from a single datapoint and thus requires only one update on the result matrix. This can be generalized so that many datapoints with identical shift type and index can be combined into a single combined shift vector. As the Fourier threads already process an algorithm with an asymptotic complexity of (N), these additional N complex additions do not change its asymptotical complexity. Therefore, this approach scales well and thus is incorporated in all implemented approaches.

5.2.1 CPU based approach

In this approach the resources of the CPU are split between the two tasks: calculating the shift vectors and combining them, and updating the matrix. No GPUs were used. This was done to get a baseline to compare the GPU based approaches to. This approach performed well concerning the throughput as this is mainly dependent on the CPU in all implementations. But updates took significantly longer and therefore the number of updates it was able to achieve were very limited as seen in Figure 4.10.

A certain improvement was achieved by efficiently utilising hyperthreading. Hyperthreading allows a CPU to very quickly switch between two threads when one is waiting for a slow

memory access or something similar. This means a CPU can execute two threads seemingly in parallel. But as these threads are not executed truly in parallel on separate cores, some unexpected effects can occur when increasing the number of threads over the number of physical cores. This is why in Figure 4.7 an increase from 20 to 40 update threads increased the total execution time. When the distribution was 20 Fourier threads and 20 update threads, 50% of the CPU's performance was allocated to the Fourier threads. With 40 update threads the Fourier threads used only 33.3% of the CPU's performance. Even though the CPU was able to process more due to hyperthreading between multiple threads in total less performance was put into reading data and more into updating the result matrices. This has to be taken into consideration when distributing the computing capacity of the CPU.

The separation between Fourier threads and update threads is an inefficient simplification in the allocation of CPU threads. After all the datapoints are read, all Fourier threads are idling, while the update threads are still processing. To speed up the process after reading is completed, the Fourier threads could be repurposed to update threads. This would reduce the time after read significantly. Thus, the time after read should not be seen as an important factor when deciding upon the number of update threads.

Still CPUs are suboptimal for this algorithm in a streaming environment. Partially this could be compensated by significantly increasing the processing power by using better or more CPUs. Datacenters could provide this infrastructure but as CPUs are not the best component for these computations anyway this is not the preferable approach.

5.2.2 Unsynchronized Approach

In this approach GPUs are utilized to replace the CPUs in the parts where they performed the least: updating the result matrices. This can be efficiently handled by GPUs because the updating can be easily executed in parallel. Using GPUs cuts down the time per update by a factor of 18 for a matrix dimension of 1024 up to a factor of 28 for a matrix dimension of 4096 (compare Figure 4.10). As this effect increases with the size of the matrix dimension, GPUs are essential for larger matrices.

This approach is the minimal approach as it purely implements the SPIFT with the aggregation matrix. It focuses on separating the individual processes and thus minimizing the synchronization overhead. But some synchronization might bring a performance increase greater than its synchronization overhead. Thus, several variations of this approach were implemented, each trying to improve in a different area.

5.2.3 Block Update approach

In this approach the number of updates could be set. This allows to reduce the load on the GPUs but still maintain the number of updates required to achieve the desired output. As this approach makes fewer or an equal number of updates than the unsynchronized approach and performs equally in the other parameters it is not optimizing the throughput. Its main advantage is the reduction in computing power used and thus a reduction in energy consumption. This makes sense, when the desired output can be quantified and lies below the maximum

possible output. As the aim in these experiments was to maximize output, this approach was abandoned.

5.2.4 Queued Approach

Two main features were introduced in the Queued Approach. One was to increase the throughput by reducing the tasks of the Fourier threads. This was implemented by letting the GPUs instead of the Fourier threads combine the shift vectors of the same shift type and shift index. The other was to further reduce the lag between the measurement of a datapoint and its integration into an intermediate result. This was implemented by changing the layout of the aggregation matrix. The aggregation matrix was implemented as a queue where each new shift vector was enqueued together with its shift type and index. The update threads then dequeue the shift vectors and their types and indices and integrate them into the result matrices. This ensures that the datapoints are processed in the order they appear in the data stream. To still profit from the idea of combining similar shift vectors the update threads linearly searched the queue for matching vectors. The vectors found would then individually be copied to the GPU and combined there.

Combining the shift vectors on the GPU was efficient and the throughput of the Fourier threads was increased. But searching linearly through the queue for matching vectors was highly inefficient. For larger matrix sizes the chance of finding matching vectors decreased. Less datapoints were aggregated and the update threads were not able to keep up with taking out datapoints from the queue. Thus, the queue grew indefinitively. In a streaming environment this is not acceptable. Additionally, it completely failed to reduce the lag between the measurement of a datapoint and its integration into an intermediate result as the datapoints had to wait even longer in the result matrix than in the Unsynchronized Approach. In the implemented prototype not even the Fourier threads were faster as the access to the one aggregation queue had to be serialized to avoid race conditions. In contrast, in the normal aggregation matrix only the access to a vector with the same shift index and type had to be serialized. The bigger the matrix the more different combinations there are, and the serialization happened less often.

5.2.5 GPU parallel approach

In the GPU parallel approach only one result matrix exists. But this result matrix is split up among all GPUs and each GPU updates its part of the result matrix (Figure 4.4). The main benefit of this is to be able to generate the final image and thus also intermediate images faster. This approach failed as it is inefficient to synchronize GPUs. One reason could be that the speed of the individual GPUs fluctuates and by synchronizing them, the fast ones have to wait for the slower ones. The speed of the GPUs depends on many parameters ranging from its current core temperature to its silicon quality. Another reason might be that the shift vector has to be copied to all the GPUs at the same time which might overload the bus. Several of these small asynchronicities can have a large impact as the update itself takes only a very short amount of time but many updates must be synchronized. In the prototype implementation the synchronization overhead was so big that increasing the number of GPUs

from 4 to 5 decreased the number of updates performed. Even with few GPUs the number of updates were not comparable to those of the unsynchronized approach. Thus this approach was abandoned too.

Further investigation could be beneficial as with this approach the generation of an intermediate image can be accomplished very fast. Each GPU could divide its result matrix by the number of updates in parallel and then the matrix parts could be assembled into an intermediate image in constant time as they only have to be appended. If the GPU synchronization were to be improved, this approach could perform very well in applications where a lot of intermediate images are needed.

6 Conclusion

Even though the maximum throughput of the algorithm cannot be significantly increased by harnessing the computational power of GPUs the output of the algorithm is heavily dependent on them. Already when using an exemplary 6 GPUs, up to 28x more updates of the result matrices are possible which increases the rate at which output can be generated. The larger the matrices are the more important the use of GPUs becomes.

Further investigation is needed into the best data structure to aggregate the shift vectors, as a better data structure has potential to not only increase the algorithms throughput but also increase the algorithms output and reduce the lag between the measurement of a datapoint and its integration into the result matrix.

To find out how much performance is truly needed and how well these approaches scale, they need to be tested in a larger testing environment. There a real data stream would have to be used and real stream of output data generated based on the needs of astronomers.

Regardless of optimizations in the aggregation of shifts or even in the computation of the Single Point Fourier Transform of the individual datapoints, GPUs will always excel in their key part in all SPIFT implementation: updating the result matrix. These N^2 independent complex summations per updated are always needed and for that a modern GPU will outperform any comparable CPU.

Bibliography

Cooley, James W and John W Tukey (1965). *An Algorithm for the Machine Calculation of Complex Fourier Series*. Tech. rep. URL: https://www.ams.org/journal-terms-of-use.

Harris, Mark (2013). *Using Shared Memory in CUDA C/C++ | NVIDIA Developer Blog*. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (visited on 07/02/2020).

NVIDIA Corporation (July 2020). *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/index.html (visited on 07/09/2020).

Saad, Muhammad et al. (2020). "Single Point Incremental Fourier Transform in Apache Flink". under review.

7 Appendix

The complete source code used for testing is accessible on Github at https://github.com/johannschwabe/SPIFT.