# NEW RADIX-2 AND RADIX-$2^2$ CONSTANT GEOMETRY FAST FOURIER TRANSFORM ALGORITHMS FOR GPUS

Sreehari Ambuluri and Mario Garrido
*Electronics Systems Division, Electrical Engineering Department, Linköping University*
*SE-581 83 Linköping, Sweden*

Gabriel Caffarena
*Bioengineering Laboratory, San Pablo CEU University*
*28668 Boadilla del Monte, Madrid, Spain*

Jens Ogniewski and Ingemar Ragnemalm
*Information Coding Division, Electrical Engineering Department, Linköping University*
*SE-581 83 Linköping, Sweden*

**ABSTRACT**

This paper presents new radix-2 and radix-$2^2$ constant geometry fast Fourier transform (FFT) algorithms for graphics processing units (GPUs). The algorithms combine the use of constant geometry with special scheduling of operations and distribution among the cores. Performance tests on current GPUs show a significant improvements compared to the most recent version of NVIDIA's well-known CUFFT, achieving speedups of up to 5.6x.

**KEYWORDS**

Fast Fourier transform (FFT), graphics processing unit (GPU), constant geometry, radix, CUDA, real-time.

## 1. INTRODUCTION

The Fast Fourier transform (FFT) is one of the most important algorithms for digital signal processing. Fast computation of FFTs is essential for a wide area of applications, especially those that handle large amounts of data or have to run in real time. Therefore, over the years many different projects aimed at implementing high-speed FFTs using field programmable gate arrays (FPGAs) (Garrido et. al. (2013), Garrido et. al. (2009), Duan et. al. (2011)), application-specific integrated circuits (ASICs) (Ahmed et. al. 2011) and graphics processing units (GPUs) (Volkov and Kazian (2008), Moreland and Angel (2003), Lili et. al (2010), Cui et. al. (2009), Brandon et. al. (2008), Govindaraju et. al. (2008), Duan et. al. (2011)).

FFT implementations on GPUs are especially interesting since they not only can produce a high speedup due to their massive amount of parallel cores, but also since many algorithms that depend on FFTs are executed on GPUs as well, such as (Ning et. al. 2011), (Wang et. al. 2010) or (Mazur et. al. 2011). However, when designing a GPU implementation, special care has to be taken to minimize memory transaction times and maximize the occupation of the cores.

In this paper we propose new radix-2 and radix-$2^2$ constant geometry FFT algorithms for GPUs. These algorithms are specially designed to optimize the use of the GPU resources. First, we use shared memory to minimize the global memory transactions, which are time consuming. Second, the algorithms make use of the concepts of constant geometry (Rabiner and Gold 1975) and processing using word groups (Baas 1999), and a special operation scheduling is used for the computations in the threads. This leads to simplification of the computations and better distribution of operations among the threads. Finally, the use of radix-$2^2$ provides additional improvements, as it reduces the number of multiplications in the algorithm (Garrido et. al. 2013). Although the use of radix-$2^2$ FFTs is successful in FPGAs (Garrido et. al. 2013), to the best of authors' knowledge this is the first time that radix-$2^2$ is implemented in GPUs. As a result, the improved data management and the simplifications in the operations lead to a reduction in the computation time.

Experimental results show significant improvements with respect to CUFFT (Volkov and Kazian 2008).

The paper is organized as follows. Section 2 reviews the FFT and the concepts of word group and constant geometry. Sections 3 and 4 present the proposed radix-2 and radix-$2^2$ constant geometry FFTs, respectively. Section 5 presents the experimental results. Finally, Section 6 shows the main conclusions.

## 2. THE FAST FOURIER TRANSFORM

The discrete Fourier transform (DFT) of a signal in the time domain, $x[n]$, is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, \quad k = 0,1,\ldots,N-1$$

where $X[k]$ is the resulting signal in the frequency domain, $N$ is the size of the transform and $W_N = e^{-j(2\pi/N)}$ is the so-called twiddle factor.

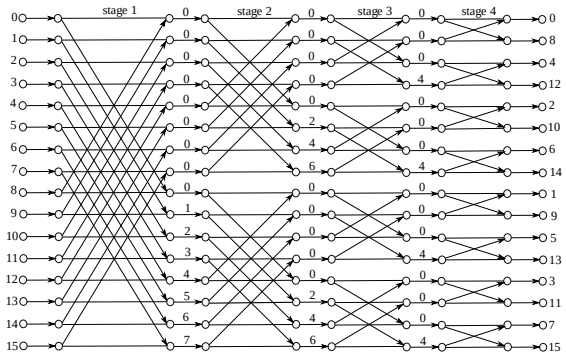



Figure 1: Flow graph of the 16-point radix-2 DIF FFT.

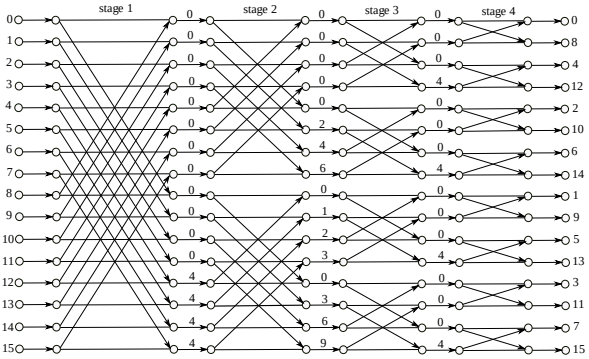



Figure 2: Flow graph of the 16-point radix-$2^2$ DIF FFT.

Fast Fourier Transforms (FFTs) are used to speed up the computations of DFTs. The Cooley-Tukey algorithm (Cooley and Tukey 1965) is the most common approach to calculate the FFT. It reduces the number of operations from $O(N^2)$ in the DFT to $O(N \log(N))$. The most commonly used decomposition methods are decimation in time (DIT) and decimation in frequency (DIF) (Oppenheim and Schafer 1989).

An $N$-point FFT is calculated in $n=\log_r N$ stages, where $r$ is called the radix of the FFT. The flow graph of a 16-point radix-2 DIF FFT is shown in Figure 1. Numbers at the input show the indexes of the input data, $x[n]$, whereas those at the outputs indicate the frequency, $k$, of the output sequence $X[k]$. Each stage of the graph consists of butterflies and rotations by the twiddle factors. Butterflies calculate additions in their upper edges and subtractions in the lower edges. Rotations are calculated according to

$$W_N^{\phi} = e^{-j\frac{2\pi}{N}\phi}$$

where $\phi$ are the numbers on the edges of the graph. Radices of the form $2^k$ are widely used in FFTs (Garrido et. al. 2013). Figure 2 shows the flow graph of a 16-point radix-$2^2$ DIF FFT. Radix-$2^2$ only differs from radix-2 FFT in the placement of the rotations (Garrido 2009). This has the advantage that rotations in odd stages are trivial, i.e., rotations by 1, j, -1 or j. These trivial rotations can be easily implemented by interchanging the real and imaginary parts of the inputs and/or changing their sign.

Further explanation of all these concepts can be found in previous literature (Oppenheim and Schafer (1989), Rabiner and Gold (1975), Garrido et. al. (2013)).

### 2.1 FFT Processing using Word Groups

In FFT architectures that consist of a memory and a processing element, groups of data are fetched from memory, processed and sent back to memory. This is done in an iterative fashion until all the operations of the FFT have been carried out. In this context, a word group (WG) is the number of data elements that are fetched, processed as a group and sent back to memory.

Figure 3 shows a word group of two data elements, i.e., WG = 2. The highlighted data are read from memory, processed by the butterfly, and the results are written back to memory. Then, the same is done with the next pair of data in the same stage. Once all the computations in the first stage of the FFT are carried out, the processor starts with the computations in the second stage.

Figure 4 shows the case of a word group equal to four. Now four data elements are read, processed and written back to memory as a group. In this context an epoch is defined as the number of FFT stages covered in each iteration. The use of larger word groups leads to a reduction in the total number of accesses to memory. This is beneficial in architectures with cache memory (Baas 1999), where the access time to the main memory is high. For instance, the computations highlighted in Fig 4 for WG = 4, require 4 read and 4 write operations, whereas by using WG = 2 (see Fig. 3) the memory is also accessed between stages 1 and 2, leading to 8 read and 8 write operations.

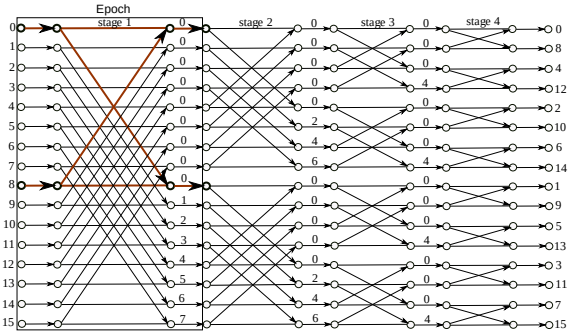



Figure 3: 16-point radix-2 DIF FFT using 2-word groups.

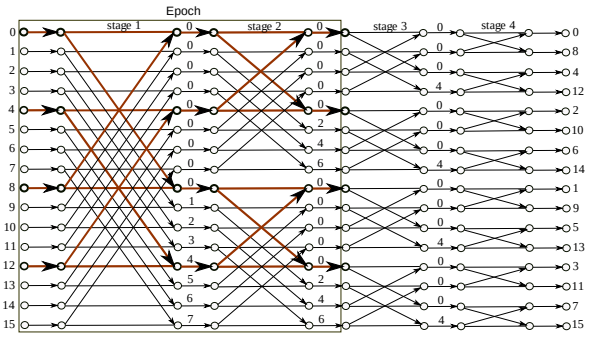



Figure 4: 16-point radix-2 DIF FFT using 4-word groups.

## 2.2 Constant Geometry FFT

Figure 5 shows a constant geometry (CG) radix-2 FFT (Rabiner and Gold 1975). The calculations are the same as in the conventional flow graph of the radix-2 FFT from Fig. 1. However, placement of the operations is different. The conventional FFT has the property that the outputs of any butterfly are stored in the same position as its inputs. Conversely, the constant geometry FFT has the property that all the stages follow the same pattern.

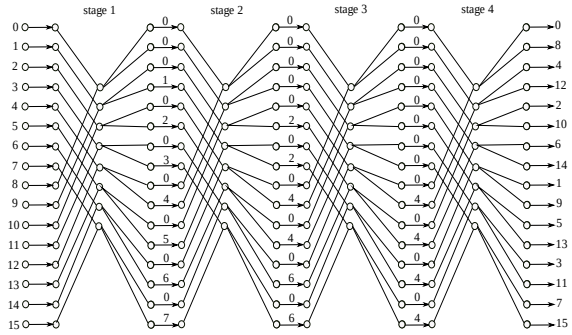



Figure 5: 16-point radix-2 DIF constant geometry FFT.

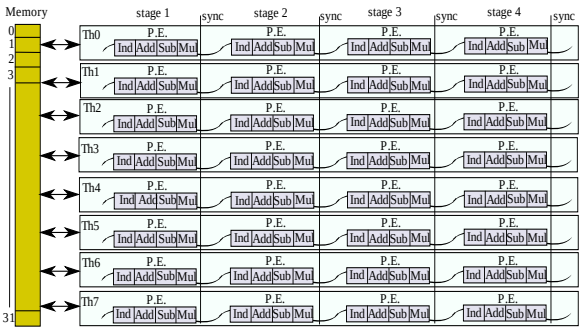



Figure 6: Parallel implementation of an FFT on GPUs.

# 3. PROPOSED RADIX-2 CONSTANT GEOMETRY FFT

The proposed algorithms are based on three main ideas: i) the processing using word groups; ii) the use of a constant geometry structure; and, iii) the use of an optimized schedule of the threads. This combination leads to important benefits when the FFT is calculated in GPUs. In this section the proposed radix-2 constant geometry FFT is explained. The next section will deal with the radix-$2^2$ implementation.

## 3.1 Parallelizing the FFT on GPUs

GPUs have the benefit that they can process large amounts of data in parallel. This parallelization increases the throughput and reduces the execution time. For this reason, it is important to find parallelism in the algorithms, so that the computations can be efficiently distributed among multiple parallel cores. Figure 6 shows the parallel implementation of the radix-2 FFT algorithm for the case of N = 16. As explained in Section 2, the number of stages is $\log_2 N$, each stage has N/2 butterflies, and each butterfly is followed by a multiplication by a twiddle factor. In Fig. 6 each processing element (PE) computes first the indexes of the required data and then calculates the butterflies and multiplications. After these operations, synchronization points are necessary since data must be redistributed among the threads at the end of each FFT stage.

## 3.2 Constant Geometry

The proposed radix-2 constant geometry FFT algorithm is based on the flow graph shown in Fig. 5. The reason why we apply the constant geometry algorithm to GPUs is that it allows for simplification of the index calculations, as will be shown next. The data elements for the FFT implementation are stored in memory. The indexing before each butterfly operation is used to determine the read and write addresses before and after the operations of each butterfly, respectively. If we consider the conventional graph in Fig. 1, the indexes of data that are processed together in a butterfly are different at each stage. The first stage processes data whose index differ in N/2 = 8. For instance, the indexes of the inputs to the first butterfly in the first stage are 0 and 8. In general, each stage $s \in \{1,...,n\}$ considers pairs of data that differ in $2^{n-s}$ (Garrido et.al. 2013). According to this, in a GPU the data indexes for a given thread are calculated from the thread ID (TID) as

$$I_{IN0} = TID + floor(TID / 2^{n-s}) \cdot 2^{n-s} \qquad I_{OUT0} = I_{IN0}$$
$$I_{IN1} = I_{IN0} + 2^{n-s} \qquad I_{OUT1} = I_{IN1}$$

where $I_{IN0}$ and $I_{IN1}$ are the indexes of the two input data that are processed in the same PE, and $I_{OUT0}$ and $I_{OUT1}$ are the output indexes. Note that the input and output indexes are the same, which allows to write the outputs in the same place where the inputs were. However, the indexes have to be recalculated at each state of the FFT, which introduces additional computations.

In the new approach using the constant geometry FFT the indexes are calculated as

$$I_{IN0} = TID \qquad I_{OUT0} = 2 \cdot TID$$
$$I_{IN1} = TID + N/2 = I_{IN0} + N/2 \qquad I_{OUT1} = 2 \cdot TID + 1$$

In this case, the indexes do not depend on the FFT stage (depicted by *s*). Therefore, they only have to be calculated once at the beginning of the computations instead of once per stage, which reduces the execution time. The fact that the input and output indexes of each stage are not the same is not an inconvenience for the GPU, because it allows for selecting the read and write addresses of the shared memory freely.

## 3.3 Processing using Word Groups

The second improvement of our algorithm is the use of word groups. For 2-WG radix-2, each PE in Fig. 6 calculates a butterfly and a rotation, and each stage consists of N/2 independent PE in parallel. Furthermore, each word group is processed by two threads. This provides a higher degree of parallelization than using a single thread for each PE.

This processing depends on multiple threads actually being processed in parallel on the same SM. In a GPU, threads are processed in groups called warps, which is a group of 32 threads. A warp is "woven together", and explicitly executed in parallel (Sanders and Kandrot 2011). Thus, the threads that work in the same word group are organized so that it is guaranteed that they belong to the same warp. This is basically a question of using threads with neighboring numbers. This avoids that data are unsynchronized and also avoids random errors due to race conditions.

## 3.4 Scheduling

Finally, an optimized scheduling has been used in order to balance the operations among the threads and, therefore, reduce the critical path in the computations. The FFT algorithm processes complex data and, thus, all the operations in the FFT are complex operations. For a 2-WG, a PE consists of a butterfly and a rotator. The butterfly requires four real additions, and the rotator needs four real multiplications and two additions. Here we assume that additions and subtractions have the same cost and count both as additions.

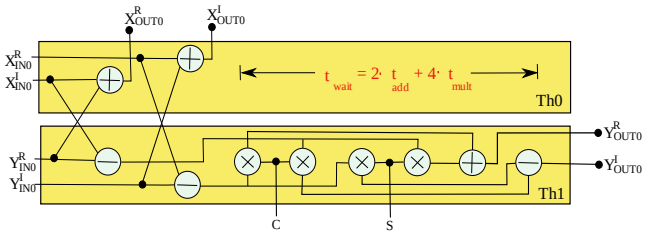



Figure 7: Unbalanced scheduling for 2-WG using 2 threads.

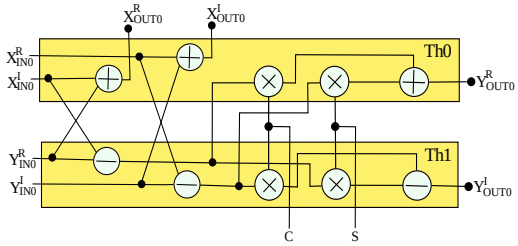



Figure 8: Balanced scheduling for 2-WG with 2 threads.

Figure 7 shows a first approach to carry out the operations using two threads for 2-WG. The operations of the upper and lower edges are carried out by the threads Th0 and Th1, respectively. The critical path (CP) is

$$CP = 4 \cdot t_{add} + 4 \cdot t_{mult}$$

In this scheduling, the thread Th1 has to calculate more computations than the thread Th0. Therefore, the thread Th0 has to wait for a time $t_{wait} = 2 \cdot t_{add} + 4 \cdot t_{mult}$ while Th1 is operating and, therefore, the workloads of the threads in Fig. 7 are unbalanced.

The critical path can be reduced by distributing the operations equally between the threads. The proposed scheduling is shown in Figure 8. This scheduling balances the operations between the threads. This reduces the critical path and, therefore, the processing time. In this case there is no waiting time and the CP is

$$CP = 3 \cdot t_{add} + 2 \cdot t_{mult}$$

The equalization of the computation paths between two threads in a word group also results in a better parallelization since both threads have now very similar computation paths, and more SIMD instruction can be executed. The result is a significant reduction in the computation time with respect to the unbalanced scheduling, leading to a faster processing of the FFT stages.

# 4. PROPOSED RADIX-$2^2$ CONSTANT GEOMETRY FFT ALGORITHM

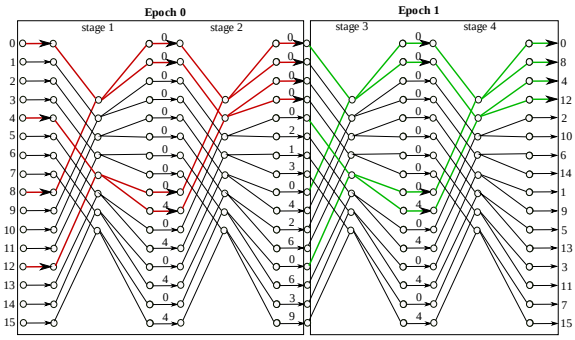
This section presents the proposed radix-$2^2$ constant geometry algorithm for GPUs. This algorithm provides additional improvements that lead to further reductions in the execution time. First, the use of radix-$2^2$ reduces the number of operations of the FFT compared to radix-2. Second, the use of constant geometry guarantees a regular computation flow for all the FFT stages. Third, the proposed radix-$2^2$ constant geometry algorithm uses 4-WG, leading to a reduction of the synchronizations required in the FFT compared to radix-2. Finally, it uses a scheduling of the 4-WG that distributes all the operations equally among four threads.

## 4.1 The Radix-$2^2$ Constant Geometry FFT Algorithm

Figure 9 shows the proposed radix-$2^2$ constant geometry algorithm. The computations are the same as in the conventional radix-$2^2$ FFT algorithm shown in Fig. 2, yet the distribution of operations is different. With respect to the radix-2 constant geometry algorithm, the radix-$2^2$ constant geometry FFT algorithm only differs in the rotations that are carried out in the stages of the FFT. This can be observed by comparing

Figures 5 and 9. The benefit of radix-$2^2$ is that the rotations in odd stages are trivial, as only multiplications by 1 and -j are needed. This simplifies the computations as those multiplications can be carried out just by changing the real and imaginary parts of the inputs and/or changing their sign. Furthermore, the pattern of the rotations in odd stages is always the same. This can be observed in Fig. 9, where the rotations at the first and at the third stages are the same.

Apart from the simplifications of the operations due to the use of radix-$2^2$, the use of constant geometry reduces the number of index calculations. As in the radix-2 constant geometry algorithm shown in Section 3, the indexing is the same for all the stages of the FFT. Thus, the indexes only have to be calculated once, and not at every stage.



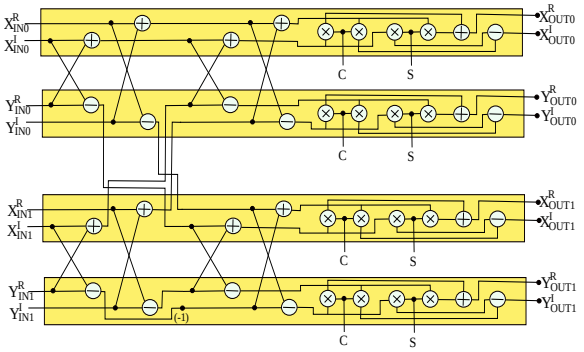




Figure 9: Proposed radix-$2^2$ constant geometry FFT algorithm. Figure 10: Scheduling for radix-$2^2$ 4-WG using 4 threads.

## 4.2 Processing using Word Groups

In general, the number of stages in an epoch, $S_E$, is defined as

$$S_E = \log_2(WG)$$

The proposed radix-$2^2$ constant geometry FFT uses a word group size WG = 4 and, therefore, each epoch covers two stages of the FFT. This can be observed in Fig. 9, where the word groups are highlighted. In the GPU, the fact that $S_E$ is 2 (due to the use of WG = 4) has the benefit that the threads only have to be synchronized every other stage. This halves the number of synchronizations compared to radix-2 and, thus, leads to a lower execution time.

## 4.3 Scheduling

The scheduling of the operations in the GPU threads is shown in Figure 10. The scheduling is balanced and distributes the operations of the 4-WG with radix-$2^2$ among 4 parallel threads. As a result, the CP is

$$CP = 6 \cdot t_{add} + 4 \cdot t_{mult}$$

In Fig. 10, it can also be observed that the trivial rotation in the 4-WG only requires a change in sign (i.e. -1). Furthermore, the synchronization after the trivial stage is not required even though several threads are used, because they are in the same warp.

## 4.4 Extension to 2D FFT

The proposed radix-2 and radix- $2^2$ algorithms can be easily extended to 2D FFTs. Since a 2D FFT is separable into two passes of 1D FFTs (Garrido 2009), a 2D FFT breaks down to a series of 1D FFTs. For an NxN data set, N N-point FFTs are calculated for each dimension, leading to a total of 2N FFT computations.

# 5. EXPERIMENTAL RESULTS

This section shows the experimental results of the proposed approach. For all the experiments we have used single precision data. We have also limited the global memory access as much as possible. The twiddle factors are precomputed in advance on the CPU (and saved on hard drive between runs), copied from the host to the device and tabulated in the shared memory. The data elements are stored in the shared memory. In each stage, the data elements are read from the shared memory, processed, and written to the shared memory. This process is repeated for all the stages.

The experiments were carried out on a NVIDIA Geforce GTX 560 and CUDA toolkit v4.0.17. The device consists of 7 SMs and a total of 336 cores running at 1620-1900 MHz.

To evaluate our proposed algorithms, they have been compared with those of the NVIDIA CUFFT 4.0 library. NVIDIA CUFFT 4.0 is the most recent and fastest FFT library developed by NVIDIA. The performance of an N-point FFT is calculated in floating point operations per second (FLOPS) as

$$FLOPS = \frac{5 \cdot N \cdot \log_2 N}{t}$$

where $t$ is the execution time. This time is measured using CUDA events. In our algorithms the execution time includes the memory transfer time of the twiddle factors and the kernel execution time. The execution time of the CUFFT 4.0 library includes the planning and the kernel execution time. We do not consider the memory transfer time of the data elements, because it is the same in all the cases.

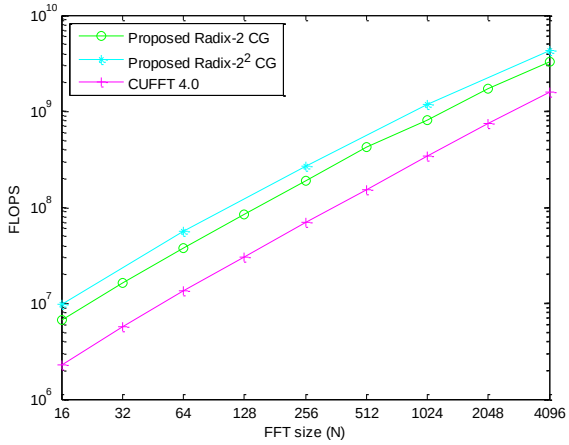



Figure 11: Comparison of the proposed 1D radix-2 and radix-$2^2$ CG FFTs with CUFFT 4.0.

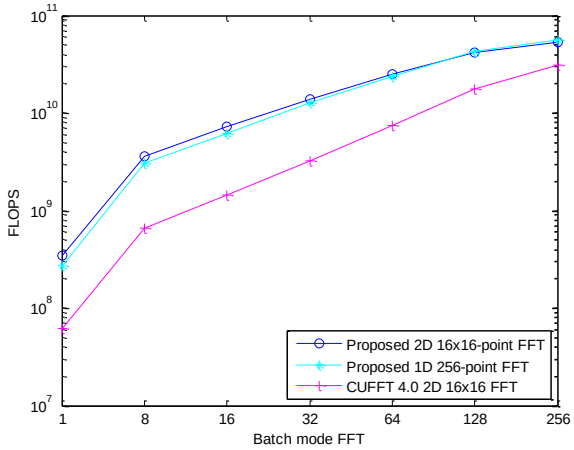



Figure 12: Batch mode comparison of the proposed FFTs with CUFFT 4.0

Figure 11 compares the execution time of a single FFT using the proposed 1D N-point radix-2 and radix-$2^2$ constant geometry FFTs, and CUFFT 4.0. Fig. 11 shows that the proposed radix-2 FFT algorithm improves 2 to 2.9 times the performance of CUFFT 4.0. This improvement is achieved thanks to the constant geometry and the balanced scheduling of the 2 WG using 2 threads. Furthermore, radix-$2^2$ achieves even higher performance than radix-2. The performance of our radix-$2^2$ algorithm is 2.7 to 4.3 times the performance of CUFFT 4.0. This results from the use of constant geometry, simplification of rotations, and less synchronization points due to the use of 4 WG.

Figure 12 compares multiple FFTs executed in batch mode (Sanders and Kandrot 2011). We have considered an image of 256x256 that is processed in tiles of 16x16. This consists of a stream of 256 2D 16x16-point FFTs. The 2D 16x16-point FFT is implemented by using the proposed 1D 16-point radix-$2^2$ CG FFT, as explained in Section 4.4. Fig. 12 shows that the performance of the proposed 2D 16x16-point FFT improves CUFFT 4.0 significantly. The speedup ranges from 1.7x to 5.6x depending on the batch mode that is chosen. Finally, in order to verify the performance of the 2D 16x16-point FFT, Fig. 12 compares it to a 1D 256-point radix-$2^2$ FFTs, whose complexity is comparable. As expected, both of them achieve similar results.

## 6. CONCLUSION

Highly efficient fast Fourier transform algorithms have been presented. The algorithms run significantly faster than CUFFT 4.0 on a modern GPU, achieving speedups up to 5.6x. This higher performance is due to the following optimizations:

1. Usage of constant geometry, which simplifies the indexing.
2. Usage of radix-$2^2$ constant geometry to simplify certain stages.
3. Usage of word groups with balanced scheduling, which distributes related calculations among several threads while reducing the synchronization points.
4. Usage of the shared memory for accessing both twiddle factors and the data elements.

There are several future research lines to expand and improve our work. First, we will continue to develop the new FFT algorithm in order to increase the speedup. Second, we will extend our library to larger sizes of both 1D and 2D FFTs, and also address the implementation of 3D FFTs. Third, we will also extend our library to support double precision. Finally, we plan to port the library onto other devices.

## REFERENCES


Ahmed, T., Garrido, M., and Gustafsson, O., 2011. A 512-point 8-parallel pipelined feedforward FFT for WPAN. *Proceedings of Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, pp. 981–984.

Baas, B., 1999. A low-power, high-performance, 1024-point FFT processor. *In IEEE Journal of Solid-State Circuits*, Vol. 34, No. 3, pp. 380-387.

Brandon, L., Boyd, C., and Govindaraju, N., 2008. Fast computation of general Fourier transforms on GPUs. *Proceedings of IEEE International Conference on in Multimedia and Expo*, Hannover, Germany, pp. 5–8.

Cooley, J.W. and Tukey, J. W., 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation.*, Vol. 19, No. 90, pp. 297–301.

Cui, X., Chen, Y., and Mei, H., 2009. Improving performance of matrix multiplication and FFT on GPU. *Proceedings of International Conference on Parallel and Distributed Systems*, Shenzhen, China, pp. 42-48.

Duan, B., Wang, W., Li, X., Zhang, C., Zhang, P., and Sun, N., 2011. Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU. *Proceedings of International Conference on Field-Programmable Technology*, New Delhi, India, pp. 1-6.

Garrido, M., 2009. *Efficient Hardware Architectures for the Computation of the FFT and Other Related Signal Processing Algorithms in Real Time*. Ph.D. dissertation, Universidad Politécnica de Madrid.

Garrido, M., Grajal, J., Sánchez, M. A., and Gustafsson, O., 2013. Pipelined Radix-$2^k$ Feedforward FFT Architectures. *In IEEE Transactions on Very Lrge Scale Integration Systems*, Vol. 21, No. 1, pp 23-32.

Garrido, M., Parhi, K.K., and Grajal, J., 2009. A Pipelined FFT Architecture for Real-Valued Signals. *In IEEE Transactions on Circuits and Systems I*, Vol. 56, no. 12, pp. 2634–2643.

Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., and Manferdelli, J., 2008. High performance discrete Fourier transforms on graphics processors. *Proceedings of IEEE Conference on Supercomputing*, Piscataway, NJ, USA, pp. 2:1–2:12.

Lili, Z., Shengbing, Z., Meng, Z., and Yi, Z., 2010. Streaming FFT asynchronously on graphics processor units," in Information Technology and Applications (IFITA), International Forum on, vol. 1, pp. 308–312.

Mazur, R., Jungmann, J. and Mertins, A., 2011. On CUDA implementation of a multichannel room impulse response reshaping algorithm based on pnorm optimization. *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pp. 305–308.

Moreland, K. and Angel, E., 2003. The FFT on a GPU. *Proceedings of Workshop on Graphics Hardware*, pp. 112-119.

Ning, X., Yeh, C., Zhou, B., Gao, W., and Yang, J., 2011. Multiple-GPU accelerated range-doppler algorithm for synthetic aperture radar imaging. *Proceedings of IEEE Radar Conference*, Kansas City, MO, USA, pp. 698–701.

Oppenheim, A. and Schafer, R., 1989. *Discrete-Time Signal Processing*. Prentice Hall.

Rabiner, L. R. and Gold, B., 1975. *Theory and Application of Digital Signal Processing*. Prentice Hall.

Sanders, J. and Kandrot, E., 2011. *CUDA by Example*. Addison-Wesley.

Volkov, V. and Kazian, B., 2008, Fitting FFT onto the G80 architecture. [Online]. Available: http://www.cs.berkeley.edu/~kubitron/courses/cs258- S08/projects/reports/project6_report.pdf

Wang, L., Shi, D., and Liu, D., 2010. Optimized GPU Framework for Pulsed Wave Doppler Ultrasound. *Proceedings of International Conference on Bioinformatics and Biomedical Engineering*, Chengdu, China, pp. 1–4.