Optimization of Fast Fourier Transform (FFT) on Qualcomm Adreno GPU

Vaibhav Gandhi, Hongqiang Wang, Alex Bourd
vaibgand, wangh, abourd@qti.qualcomm.com
GPU Compute Team, Qualcomm Technologies. Inc.
Qualcomm Technologies, Inc.

Introduction

Fast Fourier Transform (FFT) is a well-known algorithm that calculates the discrete Fourier Transform of discrete data, converting from temporal or spatial domain into frequency domain. It has a wide variety of applications in engineering, science and technology as the complexity of Fourier Transform is lowered from ***$O(N^2)$*** to ***$O(NlogN)$*,** where *N* is the data size. This often leads to an order of magnitude computation reduction for large data sizes.

This work illustrates how to accelerate the FFT algorithm on Qualcomm's Adreno^{TM} GPUs by using OpenCL^{TM}. We show that decent FFT perf with good power and energy efficiency can be achieved on Adreno GPUs by using various optimization techniques, such as on-chip memory, improved memory access patterns, parameter tuning, and fp16 data type.

FFT on GPU

- **2D vs 1D FFT.** We perform the 2D complex FFT by taking advantage of the separable nature of FFT. Each stage in figure below corresponds to a separate OpenCL kernel.

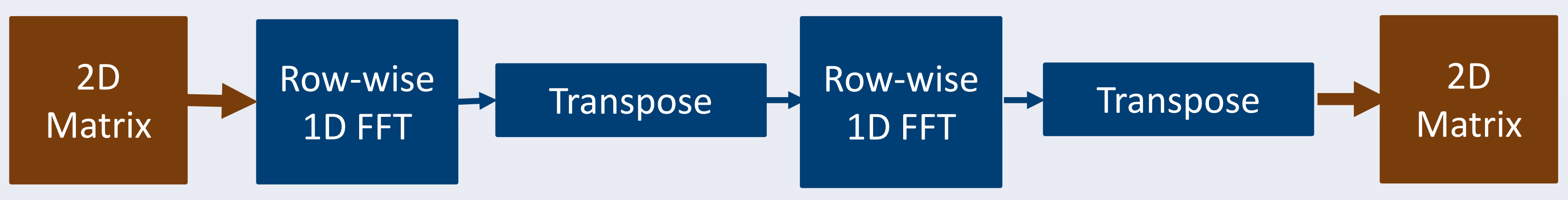


Fig. 1: Computing 2D FFT by separating into 1D row-wise and column-wise FFTs

- **Naïve implementation**

Row-wise FFT

- Compute FFT using Radix-2 Cooley-Tukey Decimation in Time (DIT) algorithm.
- Input matrix stored in global memory as an OpenCL buffer object. Each element is a complex number stored as two floats. Launch kernel with local size ***(min(MAX_WG_SIZE, width/2), 1)*** and global size ***(min(MAX_WG_SIZE, width/2)***, height). Each workgroup computes 1D FFT of one row.
- Total ***log2(FFT_LEN)*** number of iterations are needed for Radix2 FFT. Intermediate result is stored in a scratch-pad. The naïve implementation uses a buffer in global memory for this purpose. Later we will use local memory.
- Data reads in the first iteration cannot be optimally vectorized because of bit reversal. Consequently, vectorization can only be up to 64 bits. The writes can be vectorized optimally to 128 bits. For the rest of the iterations, we can fully vectorize reads by packing 128 bits in every read and write.
- Global memory barrier needed after every iteration.

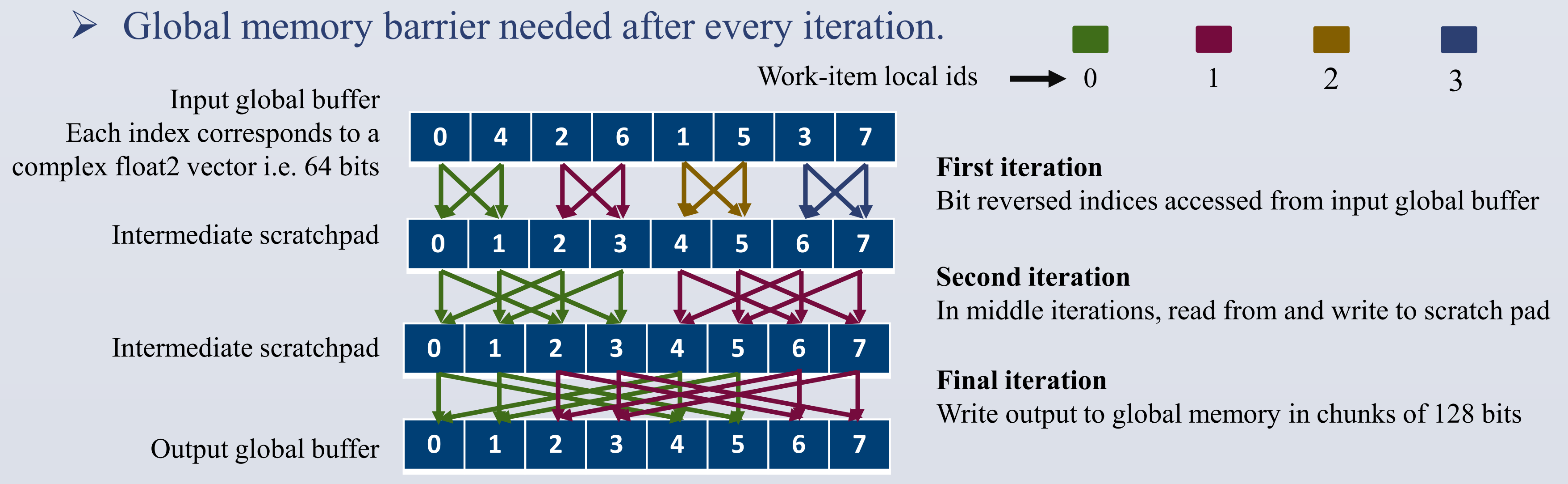

Work-item local ids
0
1
2
3
Input global buffer
Each index corresponds to a complex float2 vector i.e. 64 bits
0 4 2 6 1 5 3 7
Intermediate scratchpad
0 1 2 3 4 5 6 7
Intermediate scratchpad
0 1 2 3 4 5 6 7
Output global buffer
0 1 2 3 4 5 6 7
First iteration
Bit reversed indices accessed from input global buffer
Second iteration
In middle iterations, read from and write to scratch pad
Final iteration
Write output to global memory in chunks of 128 bits

Fig. 2: Example 1D DIT FFT of length 8

Transpose

- Each work-item reads 2x2 grid of complex numbers by two vectorized 128-bit reads. Writes 2x2 grid by two vectorized 128-bit writes.
- Performed out-of-place, so no synchronization needed.

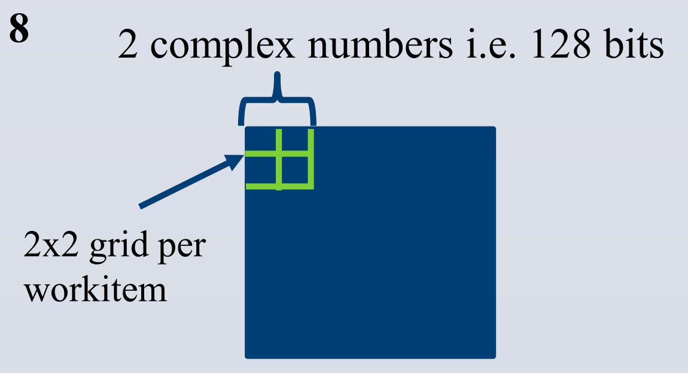


Fig. 3: Transpose kernel overview

Optimizing FFT2D for Adreno GPU

- **Use local memory for scratchpad.** Use fast, on-chip local memory to store the intermediate results of 1D FFT computations in each workgroup. As compared with global memory, local memory has shorter latency and better power efficiency, which is important for mobile devices.
- **Load matrices as image object.** For both FFT1D and transpose kernels, input matrices can be loaded into GPU for computation through read-only images. CL_RG format is used instead of CL_RGBA because though each work-item reads four fp32 (128b) data for butterfly computation, they are adjacent only in pairs, i.e., two 64b data cannot be loaded in one transaction. While writing back FFT output, each work-item can write 4xfp32 (128b) values in one transaction into an image with CL_RGBA format.

The transpose kernel reads data from the read-only CL_RGBA image, which is the output of the previous row-wise FFT stage, and writes output to a CL_RG image, which is then processed at the second FFT stage.

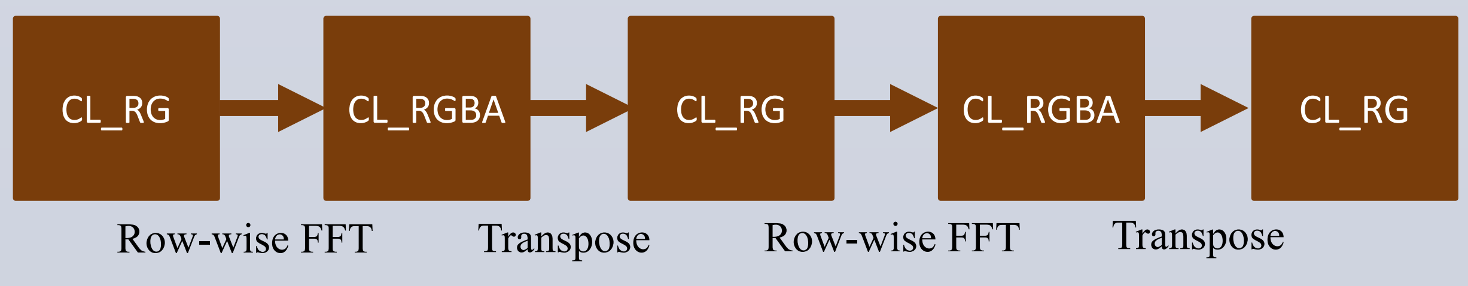


Fig. 4: Input and output image formats for different kernel stages

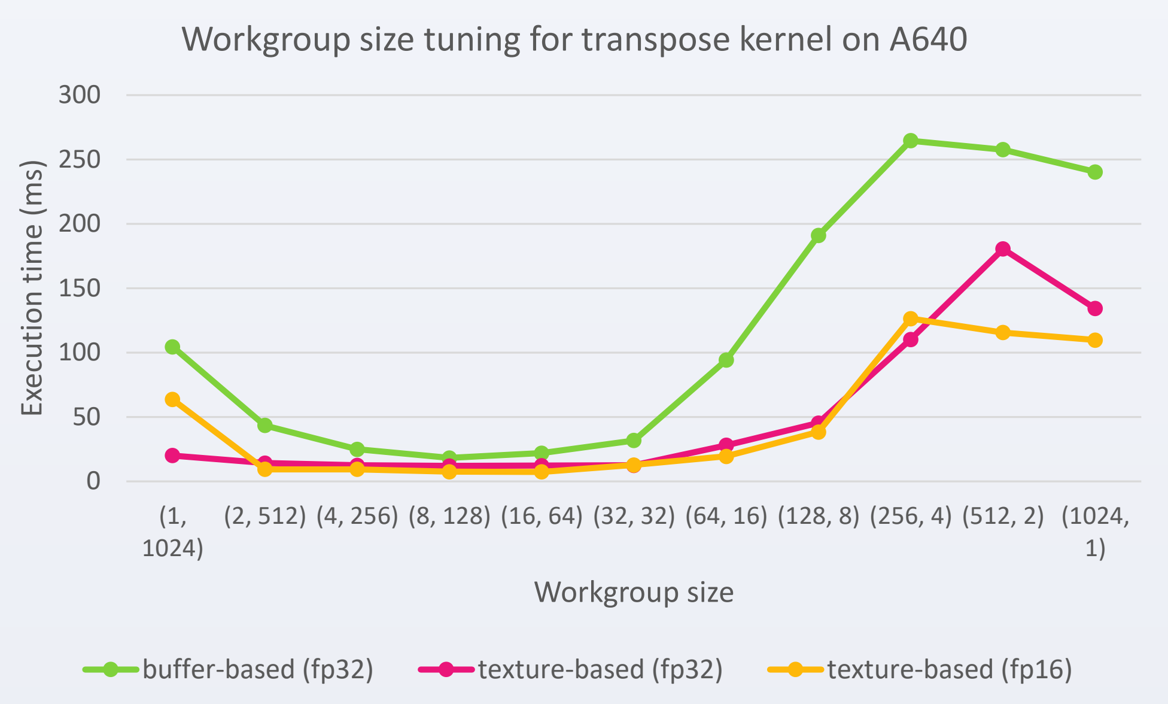
- **Use fp16 instead of fp32.** When some loss in precision is acceptable, using fp16 can provide twice the ALU capacity (measured in flops) on Adreno GPU.

Performance Analysis

- **Workgroup size/shape tuning**

The workgroup size of FFT1D kernel is set to ***min(MAX_WG_SIZE, width).***

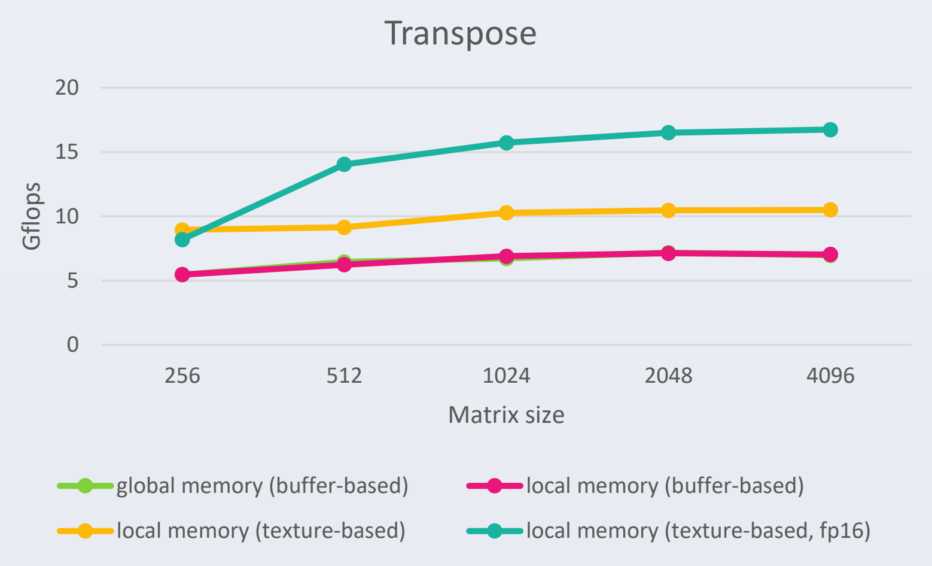
For the transpose kernel, we tune the optimal workgroup for various versions of our algorithm for different Adreno GPUs. Significant perf gains can be achieved by tuning the workgroup size and shape.

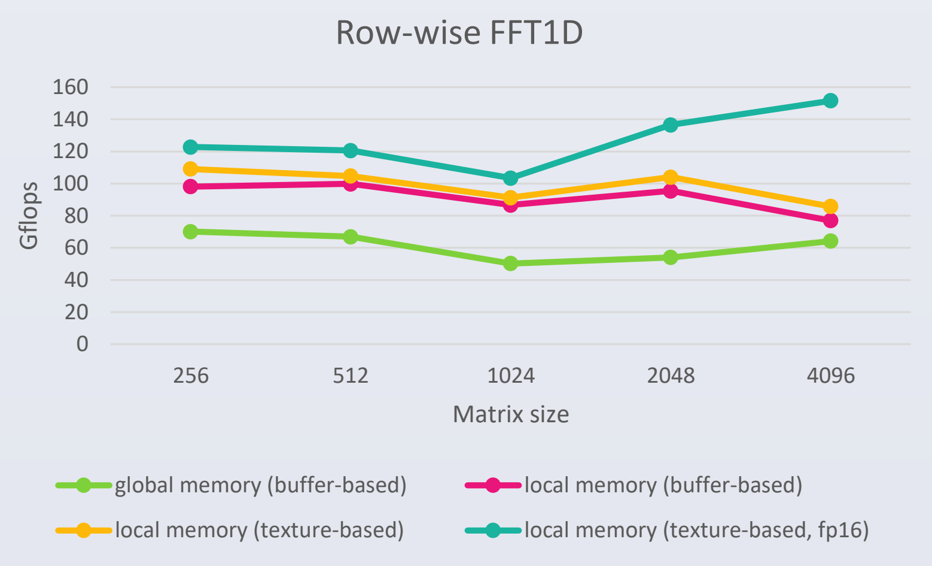


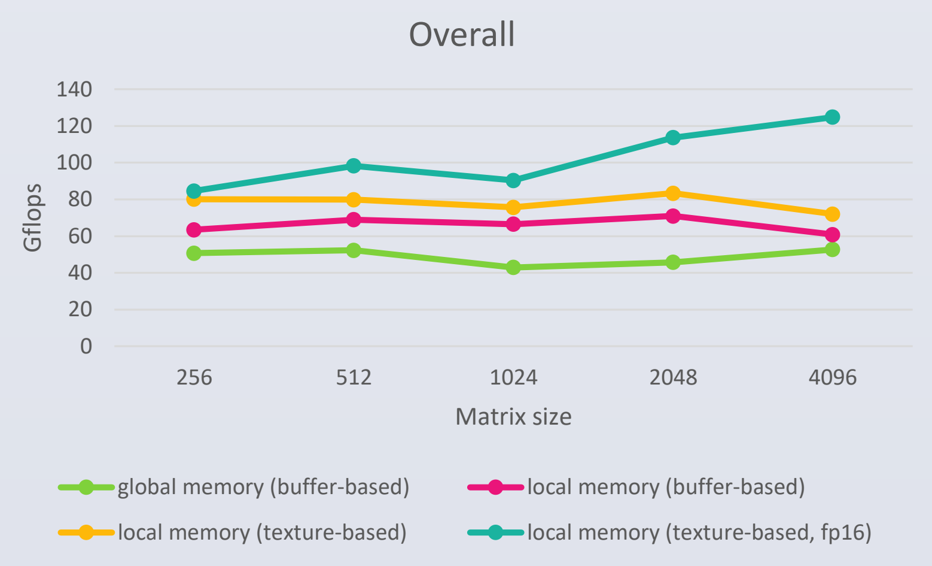
- **Perf comparison on Adreno A640**

Four approaches are implemented and optimized following the above optimization steps. We measure the actual kernel execution times and do not include the setup times at host.

- Using local memory instead of global memory as the intermediate scratchpad improves the perf of Row-wise FFT1D by ~50%.
- Using images instead of buffers improves Row-wise FFT1D perf by ~10%. The improvement is relatively minor as the bit-reversal on input indices leads to poor cache-locality. However, the perf of the transpose kernels improves considerably by ~50% thanks to good cache locality. The overall FFT2D perf is improved by ~20%.
- Using fp16 data type yields an average improvement of up to 30% for the overall FFT2D. It improves even more with larger input data sizes, as FP16 effectively reduces the overall data traffic by ~50% and improves bandwidth utilization. We highly recommend fp16 if precision loss is acceptable.



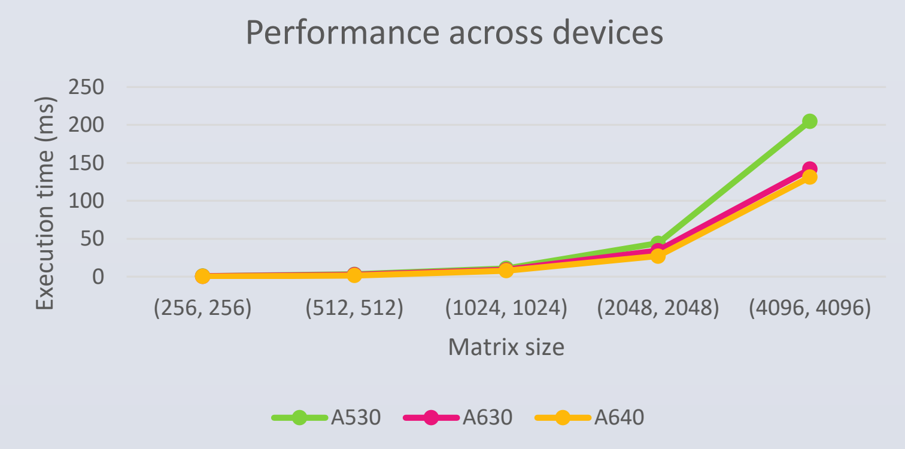




- **Perf comparison across devices**

Figure alongside shows the FFT perf across 3 different Adreno devices: A530, A630, and A640.

The execution time reported here is for local memory-based algorithm using texture for input and output.



- **Power measurement for global and local memory versions**

Using local memory

power level	freq, MHz	voltage, V	GPU, mW	CPU, mW	CX, mW	MX, mW	DDR, mW	Overall, mW
0	604	0.9227	62	53	309	65	61	550
1	500	0.7891	42	51	192	40	54	379
2	315	0.6505	28	54	174	40	47	343
3	214	0.6242	26	56	162	38	49	331

Using global memory

power level	freq, MHz	voltage, V	GPU, mW	CPU, mW	CX, mW	MX, mW	DDR, mW	Overall, mW
0	604	0.9227	89	56	321	71	90	627
1	500	0.7891	60	52	199	43	80	434
2	315	0.6505	40	62	181	43	72	398
3	214	0.6242	39	61	168	42	77	387

Table 1: Power consumption comparison for global memory (buffer-based) and local memory (buffer-based) versions of FFT2D. GPU power consumption decreases by 30% when using local memory as intermediate scratch-pad. Above data collected on Adreno 530 on Qualcomm Snapdragon 820

Acknowledgement and References

This is an extension and update of work previously done by Xujie Zhang, Vijay Ganugapati, Rotem Aviv, and Jay Yun on optimization of 2D FFT using local memory.

1. https://developer.qualcomm.com. 2018. Adreno OpenCL programming guide and best practices. Retrieved April 13, 2018 from https://developer.qualcomm.com/qfile/33472/80-nb295-11_a.pdf
2. Timmy Liu, Bragadeesh Natarajan, Pradeep Rao. 2019. clFFT. https://github.com/clMathLibraries/clFFT
3. Dan Petre, Adam Lake, Allen Hux. OpenCL FFT Optimizations for Intel Processor Graphics. Proceedings of the 4th International Workshop on OpenCL, IWOCL '16.
4. Neil Tan. 2013. Optimizing Fast Fourier Transformation on ARM Mali GPUs. https://community.arm.com/graphics/b/blog/posts/optimizing-fast-fourier-transformation-on-arm-mali-gpus

IWOCL'19, May 13–15, 2019, Boston, MA, USA © 2019 Copyright is held by the owner/author(s). ACM ISBN 978-1-4503-6230-6/19/05.
https://doi.org/10.1145/3318170.3319372