

VkFFT – Vulkan/CUDA/HIP/OpenCL/Level Zero GPU FFT library

Dmitrii Tolmachev
ETH Zurich
Zurich, Switzerland
dmitrii.tolmachev@erdw.ethz.ch

***Abstract**—The Fast Fourier Transform is an essential algorithm of modern computational science. The highly parallel structure of the FFT allows for its efficient implementation on graphics processing units (GPUs), which are now widely used for general-purpose computing. This poster presents the VkFFT - an efficient GPU-accelerated multidimensional Fast Fourier Transform library for Vulkan/CUDA/HIP/OpenCL/Level Zero projects. VkFFT aims to provide the community with a cross-platform open-source alternative to vendor-specific solutions while achieving comparable or better performance. This poster presents the optimizations implemented in VkFFT and compares its performance and precision against Nvidia cuFFT and AMD's rocFFT libraries on their latest HPC GPUs. This poster also presents the first performant implementation of Discrete Cosine Transforms on GPUs. VkFFT is released under MIT license.*

***Keywords**—GPU, FFT, VkFFT, cuFFT, rocFFT*

I. INTRODUCTION

Discrete Fourier Transform is defined as:

The fastest known algorithm of evaluating DFT is known as Fast Fourier Transform. Cooley-Tukey FFT algorithm reformulates composite size DFT as a combination of DFTs:

1. Perform DFTs of size - radix of transformation
2. Perform $O(N)$ multiplications by twiddle factors - complex roots of unity defined by the radix
3. Perform DFTs of size

Recursively applying the algorithm, FFT can be done with a collection of small prime radix routines, written explicitly.

VkFFT[1] uses Stockham autosort version of the FFT algorithm, which has a better data-access pattern for GPUs by shuffling data between threads at each radix step instead of an initial bit-reversal.

If a DFT has a prime radix that has no explicit routine (which is the case for big primes numbers), the Rader's FFT algorithm is used, calculating arbitrary prime radix as a length convolution, using convolution theorem:

If is not decomposable as small primes (which is the case for Sophie Germain primes) Bluestein's FFT algorithm is used:

1. Pad sequence with zeros to a size of at least , that can be decomposed with supported prime radix kernels
2. Perform a convolution with a precomputed Bluestein's kernel

II. VKFFT FUNCTIONALITY

- 1D/2D/3D systems, Forward and inverse directions of FFT.
- Support for big FFT dimension sizes: (2^{32} , 2^{32} , 2^{32})
- Radix-2/3/4/5/7/8/11/13 FFTs and composite radix kernels.

- **New:** Rader's and Bluestein's FFT algorithms for all other sequences.
- Single, double and half-precision support.
- Complex to complex (C2C), real to complex (R2C), complex to real (C2R) transformations.
- **New:** real to real (R2R) Discrete Cosine Transformations of types I, II, III and IV.
- Convolutions and cross-correlations.
- Native zero padding to model open systems.
- Works on Nvidia, AMD, Intel, Apple and mobile GPUs. And Raspberry Pi GPU.
- Works on Windows, Linux and macOS.
- VkFFT supports Vulkan, CUDA, HIP, OpenCL and Level Zero as backends.

III. VKFFT LIBRARY DESIGN AND API

VkFFT has a **hierarchical structure design**: Application $\Rightarrow$ Plan $\Rightarrow$ Code. This allows to make **code optimizations** for the target device architecture at runtime. Multiple levels of kernels that can be **merged and reused**.

This platform will soon be released as a standalone project with VkFFT being a part of its kernel collection. With it, it will be possible to inline VkFFT in the user's kernels, reducing memory transfers even more.

IV. MEMORY MANAGEMENT TECHNIQUES IMPLEMENTED IN VKFFT

GPUs employ a wide range of memory types to achieve high bandwidth and FLOPS. FFT is an extremely global memory bandwidth-limited algorithm, which means having two global memory data round trips instead of one often decreases performance by a factor of two. VkFFT incorporates the following techniques to reduce memory transfers:

- All FFT sequences that can fit in shared memory are done as a single upload from global memory. Including all advanced FFT algorithms, like Rader's/Bluestein's algorithms.
- Big sequences use the four-step FFT algorithm – the sequence is split into two (three for even bigger sizes) uploads with an inlined twiddle multiplication.
- VkFFT does multiple small FFTs at once – increases thread block size and allows to access continuous blocks of memory.
- VkFFT does not have a transposition for the strided FFT axes – they are done by coalescing the neighboring sequences. Strided FFTs are done in a single upload/download, just like non-strided FFTs.
- Zero padding support - VkFFT can omit sequences full of zeros and do not perform the corresponding memory transfers and computations, as the output result will be zero. This way it is possible to get up to two times the speed increase in the 2D case and up to 3x increase in the 3D case.
- Removing additional last forward FFT/first inverse FFT memory requests for convolutions by inlining kernel multiplication in the generated code. Removes one data round-trip.

- VkFFT utilizes R2C/C2R Hermitian symmetry properties. Reduces calculations and data transfers by a factor of two.
- All R2R pre- and post-processing are inlined in the FFT algorithm.

V. PERFORMANCE BENCHMARKS

We compare VkFFT performance against Nvidia's cuFFT[2] on Nvidia A100 HPC GPU (40GB, 250W, P0 profile, CUDA 11.7)

The test configuration takes multiple 1D FFTs of all lengths from the 2 to 4096 range, batch them together so the full system takes $\sim 500\text{MB}$ to 1GB of data and perform multiple consecutive FFTs.

After that, time per single FFT is obtained by averaging the result. Total system size is divided by the time taken by a single transform, resulting in the achieved bandwidth. The peak global memory bandwidth of A100 is $\sim 1.3\text{TB/s}$.

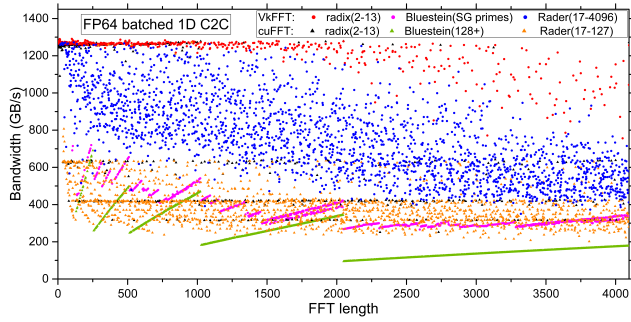


Fig. 1 Double precision A100 benchmark results: VkFFT against cuFFT.

For double precision, both VkFFT and cuFFT use radix decomposition for sequences representable as a multiplication of arbitrary number of primes up to 13.

Both VkFFT and cuFFT have Rader's algorithm implementation. VkFFT uses FFT version of it for sequences decomposable as a multiplication of primes up to 4096 (if P-1 FFT can be done with the first algorithm). cuFFT only uses Rader's algorithm for primes up to 127 and implements it as a direct matrix multiplication.

In both codes Bluestein's algorithm is used for all other sequences - Sophie Germain primes and their multiples for VkFFT and primes after 128 and their multiples for cuFFT.

We also compare VkFFT performance against AMD's rocFFT[3] on AMD MI250 HPC GPU (64GB, 250W, single-chip, ROCm 5.2). The peak bandwidth of MI250 is $\sim 1.3\text{TB/s}$.

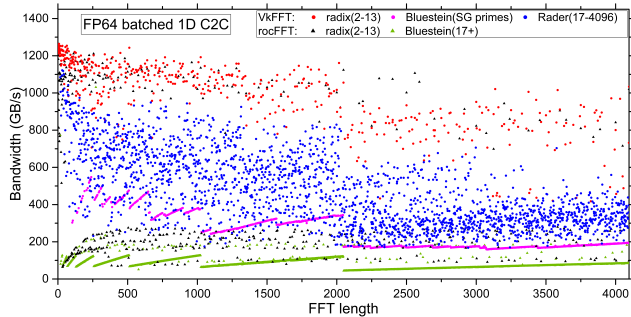


Fig. 2 Double precision MI250 benchmark results: VkFFT against rocFFT.

VI. PRECISION VERIFICATION

VkFFT precision is verified by comparing its results with long double version of FFTW[4]. We test all FFT lengths from the $[2, 100000]$ range.

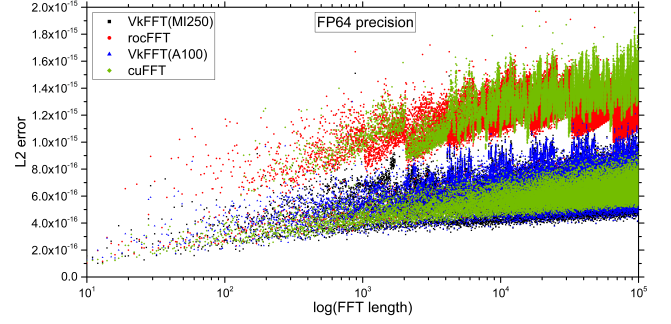


Fig. 3 Double precision A100 and MI250 precision verification results: VkFFT, cuFFT and rocFFT.

VII. DISCRETE COSINE TRANSFORMS

- No existing fast GPU DCT I-IV library before.
- Implemented as a mapping between DCT and a regular Fourier transform of a similar length, performed with VkFFT.
- The same accuracy scaling as FFTW.

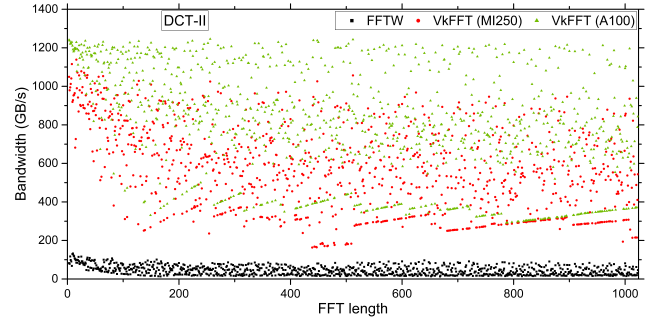


Fig. 4 DCT-II Double precision A100 and MI250 performance of VkFFT against threaded FFTW on AMD EPYC 7742 (64 cores)

ACKNOWLEDGMENT

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme grant agreement No 833848 (UEMHP) and from The Swiss National Supercomputing Centre's PASC initiative supporting the AQUA-D software development project.

REFERENCES

1. D. Tolmachev, VkFFT library, <https://github.com/DTolm/VkFFT>
2. Nvidia, cuFFT library, <https://developer.nvidia.com/cufft>
3. AMD, rocFFT library, <https://github.com/ROCmSoftwarePlatform/rocFFT>
4. Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005)