VkFFT - a Story of Vulkan Compute
GPU HPC Library Development

Dmitrii Tolmachev, ETH Zurich
Fast Fourier Transform: theory

Discrete Fourier Transform is defined as:

\[ X_k = \sum_{n=1}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \]

The fastest known algorithm for evaluating the DFT is known as Fast Fourier Transform. The Cooley-Tukey algorithm reformulates a composite size DFT \( N = N_1 \cdot N_2 \) as a combination of two DFTs:

1. Perform bit-reversal permutation to reorder data.
2. Perform \( N_1 \) DFTs of size \( N_2 \). Here we assume \( N_1 \) to be smaller than \( N_2 \).
3. Perform \( O(N) \) multiplications by twiddle factors - complex roots of unity defined by the radix.
4. Perform \( N_2 \) DFTs of size \( N_1 \).
What is VkFFT?

- 1D/2D/3D systems, forward and inverse directions of Fast Fourier Transforms.
- Support for big FFT dimension sizes: \(2^{32}, 2^{32}, 2^{32}\).
- 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.
- 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, Level Zero and Metal as backends.
Motivation before VkFFT release – March 2020-August 2020

• For my Master’s thesis, I needed to calculate long-range Dipole-Dipole interactions in magnetic structures. This is usually done as a convolution with the help of the convolution theorem:

\[ \text{DFT}\{f \ast g\} = \text{DFT}\{f\} \cdot \text{DFT}\{g\} \]

• I only had a laptop with Intel UHD 610 at that time – so no CUDA was available.
• Decided to try Vulkan, as it was new, supported everywhere API with the most explicitly outlined rules.
• I had no GPU programming experience beforehand.
• I did not plan to draw triangles – this talk will only focus on compute part of the pipeline.

• Shortly after starting, I found out that there was no FFT library for Vulkan. I attempted to make my own.
Motivation before VkFFT release – March 2020-August 2020

• By May 2020, I managed to match Nvidia’s cuFFT for small powers of two, 2D and 3D systems.

• Got an understanding that optimization of global memory transfers is the key to writing efficient GPU algorithms (at least for FFTs).

• I have implemented a full PDE solver in Vulkan for my Master thesis and managed to outperform the alternative commonly used magnetic solver (which was using cuFFT) by a factor of 3x. This was done mainly with two simple optimizations:
  − Zero-padding: VkFFT omits sequences full of zeros and doesn’t upload memory, known to be zero.
  − Merging of the last FFT step with kernel multiplication and the first step of inverse FFT.

• This was a big success, so I decided to release the library to the public.
VkFFT version 1.0 – released August 2, 2020

- 1D/2D/3D systems, maximum dimension size is 4096, single precision, only power of two systems.
- C2C and R2C/C2R transforms.
- Matrix convolutions.
- Code is a collection of written shaders tuned with specialization constants.
VkFFT version 1.0.7 – released November 26, 2020

- Maximum dimension size is now $2^{32}$ – added the Four-step FFT algorithm. No transpositions at all.
- FP16/FP32/FP64, still only power of two systems. LUT support. Register overutilization.
- Native zero padding to model open systems.

Big issue – the shader collection reached 100k LOC, adding new features to all files was a nightmare.
VkFFT version 1.1.1 – released December 16, 2020

- VkFFT now creates shader collection during the FFT plan creation phase with glslang.
- 100k LOC of shaders -> 3k LOC of the generator (at that time). VkFFT header is at 7k LOC.
- Unrolled most loops and inlined most constants.

Within the next months:
- Added radix-3/5/7/11/13 to the generator – the code is no longer power of two limited.
- Gradually replaced all GPU-dependent values with inlined by the generator variables.
- Many performance optimizations and testing.

Idea – code generated by VkFFT already looks like a sequence of additions and multiplications
- Can be generated for any API – added support for CUDA and HIP.
- A side effect – I can now use the Nvidia ncu profiler to improve performance on Vulkan and even on non-Nvidia GPUs!
<table>
<thead>
<tr>
<th>Step Description</th>
<th>Vulkan API</th>
<th>CUDA API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get device parameters</td>
<td><code>vkGetPhysicalDeviceProperties</code></td>
<td><code>cuDeviceGetAttribute</code></td>
</tr>
<tr>
<td>Allocate additional memory</td>
<td><code>vkCreateBuffer</code> + <code>vkAllocateMemory</code> + <code>vkBindBufferMemory</code></td>
<td><code>cudaMalloc</code></td>
</tr>
<tr>
<td>Configure buffer layout</td>
<td><code>vkCreateDescriptorPool</code> + <code>vkCreateDescriptorSetLayout</code> + <code>vkAllocateDescriptorSets</code> + <code>vkUpdateDescriptorSets</code></td>
<td>Binding is done at kernel dispatch</td>
</tr>
<tr>
<td>Generate binaries from the code</td>
<td><code>glslang_shader_create</code> + <code>glslang_shader_preprocess</code> + <code>glslang_shader_parse</code> + <code>glslang_program_create</code> + <code>glslang_program_add_shader</code> + <code>glslang_program_link</code> + <code>glslang_program_SPIRV_generate</code> + <code>glslang_program_SPIRV_get_ptr</code></td>
<td><code>nvrtcCreateProgram</code> + <code>nvrtcCompileProgram</code> + <code>nvrtcGetCUBIN</code></td>
</tr>
<tr>
<td>Create executables</td>
<td><code>vkCreateShaderModule</code> + <code>vkCreatePipelineLayout</code> + <code>vkCreateComputePipelines</code></td>
<td><code>cuModuleLoadDataEx</code> + <code>cuModuleGetFunction</code></td>
</tr>
<tr>
<td>Dispatch</td>
<td><code>vkCmdBindPipeline</code> + <code>vkCmdBindDescriptorSets</code> + <code>vkCmdPushConstants</code> + <code>vkCmdDispatch</code></td>
<td><code>cuMemcpyHtoD</code> + <code>cuLaunchKernel</code> (dispatches kernel to the stream)</td>
</tr>
</tbody>
</table>
VkFFT version 1.1.13 – released April 9, 2021

- Last version of 1.1 tree was 1.1.13. VkFFT header is at 14k LOC.
- Standardized benchmark used for future versions: The test configuration takes multiple 1D FFTs of all lengths from the 2 to 4096 range, batches them together so the full system takes ~ 500MB to 1GB of data and performs multiple consecutive FFTs.
- After that, time per single FFT is obtained by averaging the result. The total system size is divided by the time taken by a single transform, resulting in the achieved bandwidth. It is then compared to the peak global memory bandwidth (for A100 it is ~1.3TB/s).
VkFFT version 1.1.13 – benchmark in FP32

Bandwidth (GB/s)

FFT length

VkFFT - CUDA

cuFFT
VkFFT version 1.1.13 – benchmark in FP32

The diagram shows a comparison of bandwidth (GB/s) against FFT length for two different versions of VkFFT: VkFFT - CUDA and VkFFT - Vulkan. The data indicates a consistent performance across varying FFT lengths, with a slight fluctuation in the bandwidth values.
VkFFT version 1.2.0 – released April 13, 2021

- This main release was done to emphasize the maturity of support of multiple API backends in VkFFT.
- 1.2.0 comes with the support of the OpenCL backend.
- As development is no longer API dependent – all focus is only on algorithms.
- 1.2 is the current tree of VkFFT, no major design alterations to the generator released yet.

We will now go over notable features added during 1.2
• I started my PhD at ETH Zurich, where I need to implement polynomial transforms used to solve the equations of magnetohydrodynamics in a full sphere and other geometries.

• Added support for Discrete Cosine Transforms (DCT) of types II, III and even-length IV in VkFFT, defined as:

  - **DCT-I:** \[ X_k = x_0 + (-1)^k x_{N-1} + 2 \sum_{n=1}^{N-2} x_n \cos \left( \frac{\pi}{N-1} nk \right) \]
  - **DCT-II:** \[ X_k = 2 \sum_{n=1}^{N-1} x_n \cos \left( \frac{\pi}{N} (n + \frac{1}{2}) k \right), \text{ inverse of DCT-III} \]
  - **DCT-IV:** \[ X_k = 2 \sum_{n=1}^{N-1} x_n \cos \left( \frac{\pi}{N} \left( n + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \right) \]

• Done as a custom read/write pre/post-processing – inlined in the kernel, no additional memory transfers.
• No other GPU FFT library supports DCT.
• VkFFT header is at 21k LOC.
VkFFT version 1.2.07 – released July 25, 2021

• DCT-I, calculated as a 2N-1 length FFT often hits unsupported by radix decomposition sequences.
• Implemented Bluestein's algorithm to cover arbitrary FFT sequences. Optimized performance by using zeropadding and convolution merging for the lowest amount of memory transfers.
• Slower than radix due to 4x shared memory transfers + kernel upload increasing memory transferred.
• Done as a custom read/write pre/post-processing – inlined in kernel.
• VkFFT header is at 23k LOC.
Bluestein's algorithm
FFT length = 2039, batch = 8192
VkFFT CUDA: $T = 1.13\text{ms}$ (220 GB/s)
VkFFT Vulkan: $T = 0.98\text{ms}$ (255 GB/s)

Stockham autosort
FFT length = 2048, batch = 8192
VkFFT CUDA: $T = 0.65\text{ms}$ (384 GB/s)
VkFFT Vulkan: $T = 0.66\text{ms}$ (380 GB/s)

Device: Nvidia 3070, peak bandwidth $\sim 400\text{GB/s}$
VkFFT version 1.2.12 – released September 21, 2021

• Added support for Discrete Cosine Transforms (DCT) of types I and odd-length IV in VkFFT.
• Documentation released on August 22, 2021.
• Code for pre/post-processing starts to suffer from extreme cases of duplication.
• VkFFT header is at 27k LOC.
VkFFT version 1.2.24 – released April 11, 2022

- Added support for Level Zero backend.
- Many performance and accuracy fixes that made library ready to be used in production.
- VkFFT header is at 29k LOC.
VkFFT version 1.2.27 – released August 17, 2022

• Added support for Rader kernels inlining for arbitrary primes from 17 (that do not have explicitly optimized kernels). Rader’s FFT calculates them as a P-1 length convolution, direct multiplication – this algorithm is extremely shared memory bandwidth limited.

VkFFT version 1.2.28 – released September 4, 2022

• Version 2 of Rader’s algorithm, inlining a P-1 length convolution theorem (FFT+IFFT) for each prime P higher than 13 in Stockham FFT algorithm. This greatly reduces the number of operations needed for these primes. No other GPU library has this version of Rader’s algorithm so far.

• VkFFT header is at 40k LOC.
Multiplication Rader’s algorithm version
FFT length = 113, batch = 131072
VkFFT CUDA: T = 1.89ms (117 GB/s)
VkFFT Vulkan: T = 1.84ms (120 GB/s)

FFT Rader’s algorithm version
FFT length = 113, batch = 131072
VkFFT CUDA: T = 0.61ms (360 GB/s)
VkFFT Vulkan: T = 0.80ms (277 GB/s)

Device: Nvidia 3070, peak bandwidth ~400GB/s
VkFFT version 1.2.30 – released October 6, 2022

• Added support for Metal backend.
• Current version of VkFFT 1.2.33 and VkFFT header is at 42k LOC. This seems to be a problem, as only I can fully understand the code.
VkFFT version 1.2.33

- Current benchmark results for Nvidia A100
VkFFT version 1.2.33

- Current benchmark results for AMD MI250
VkFFT version 1.3.0

- Addresses the issues of the single header approach – code is once again fully reorganized.
- Removes some of the code duplication.
- Standardizes the code generation – no more hardlined sprintf code paths.
- Code generator now operates on special data containers, that can hold either known during the plan creation integer/float values or strings of variable names.
- Example: MUL(A,B,C) operation that performs A=B*C.
  1. If all containers have known values, A can be precomputed during plan creation.
  2. If A, B and C are, for example, register names (eax, ebx and ecx), we print to the kernel an operation of multiplication to be executed.
- Each operation like MUL has its own representation for all supported APIs.
- Not a full compiler, but a simple tool that unifies syntaxis of different APIs and supports JIT optimizations. Like what VkFFT is already, but cleaner.
- Will be released as a separate platform.
VkFFT version 1.3.0 platform structure

Application manager
Input configuration, calls for plans' initialization and dispatch, binaries and resources management. Can contain multiple plans. User interacts with VkFFT through the application.

Plan manager
Optimization of parameters (example: number of threads, shared memory, LUT allocation) for a particular task and GPU architecture, calls for code generation and compilation.

Code manager
GPU kernel code generation. Produces a single kernel for the task defined by the plan manager.

Level 2 kernels
A clear description of the problem via a sequence of calls to lower levels, kernel layout configuration.

Level 1 kernels
Simple routines: matrix-vector multiplication, FFT, pre- and post-processing, R2C/R2R mappings.

Level 0 kernels
Memory management, basic math functions inlining, subgroup functions, API-dependent definitions.
Closing words

• Gave a brief history of VkFFT development – hopefully, it was interesting.
• Shared the way I approach GPU kernel optimizations – hopefully, it can be useful.
• Announced a big redesign of VkFFT, which will come out soon – hopefully.
• The paper on VkFFT has been finished and is currently under review.
Dmitrii Tolmachev
PhD Student
dtolm96@gmail.com
dmitrii.tolmachev@erdw.ethz.ch

ETH Zürich
Institute of Geophysics
Sonneggstrasse 5,
8092 Zurich, Switzerland

VkFFT repository (MIT licensed):
https://github.com/DTolm/VkFFT