VkFFT - Vulkan/CUDA/HIP/OpenCL Fast Fourier Transform library

API guide with examples

Dmitrii Tolmachev

August 2021, version 1.2.11

Contents

1 Introduction

This document describes VkFFT - Vulkan/CUDA/HIP/OpenCL Fast Fourier Transform library. It describes the features and current limitations of VkFFT, explains the API and compares it to other FFT libraries (like FFTW and cuFFT) on the set of examples. It is by no means the final version, so if there is something unclear - feel free to contact me (dtolm96@gmail.com), so I can update it.

2 Using the VkFFT API

This chapter will cover the basics of VkFFT. Fourier transform of a sequence is called Discrete Fourier Transform (DFT). It is defined by the following formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} = \mathrm{DFT}_N(x_n, k), \tag{1}$$

where $x_n$is the input sequence, N is the length of the input sequence and $k \in [0, N-1], k \in \mathbb{Z}$is the output index, corresponding to frequency in Fourier space. Corresponding to that, inverse DFT is defined as following:

$$x_n = \sum_{k=0}^{\mathrm{N}-1} X_k e^{\frac{2\pi i}{N}nk} = \mathrm{iDFT}_N(X_k, n) \tag{2}$$

VkFFT follows the same definitions as FFTW and cuFFT - forward FFT has the exponent sign -1, while the inverse has the exponent sign 1. Note, that inverse transform by default is unnormalized, so to get the input sequence after FFT + iFFT, the user has to divide the result by N.

2.1 Installing VkFFT

VkFFT is distributed as a header-only library. The installation process consists of the following steps:

1. Copy vkFFT.h file into one of the directories included in the user's project.
2. Define VKFFT_BACKEND as a number corresponding to the API used in the user's project: 0 - Vulkan, 1 - CUDA, 2 - HIP, 3 - OpenCL. Definition is done like:

   ```
   -DVKFFT_BACKEND=X
   ```

   in GCC or as

   ```
   set(VKFFT_BACKEND 1 CACHE STRING "0 - Vulkan, 1 - CUDA, 2 -
   ↪  HIP, 3 - OpenCL")
   ```

   in CMake.

3. Depending on the API backend, the project must use additional libraries for run-time compilation:

   (a) Vulkan API: SPIRV, glslang and Vulkan. Define VK_API_VERSION to the available Vulkan version. Sample CMakeLists can look like this:

```
find_package(Vulkan REQUIRED)
target_compile_definitions(${PROJECT_NAME} PUBLIC -DVK_API_VERSION=11)#10 - Vulkan 1.0, 11 - Vulkan 1.1, 12 - Vulkan 1.2
target_include_directories(${PROJECT_NAME} PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/glslang-master/glslang/Include/)
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/glslang-master)

target_include_directories(${PROJECT_NAME} PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/vkFFT/)
add_library(VkFFT INTERFACE)
target_compile_definitions(VkFFT INTERFACE -DVKFFT_BACKEND=0)

target_link_libraries(${PROJECT_NAME} PUBLIC SPIRV glslang Vulkan::Vulkan VkFFT)
```

   (b) CUDA API: CUDA and NVRTC. Sample CMakeLists can look like this:

```
find_package(CUDA 9.0 REQUIRED)
enable_language(CUDA)
set_property(TARGET ${PROJECT_NAME} PROPERTY CUDA_ARCHITECTURES 35 60 70 75 80 86)
target_compile_options(${PROJECT_NAME} PUBLIC "$<$<COMPILE_LANGUAGE:CUDA>:SHELL:
    -DVKFFT_BACKEND=${VKFFT_BACKEND}
    -gencode arch=compute_35,code=compute_35
    -gencode arch=compute_60,code=compute_60
    -gencode arch=compute_70,code=compute_70
    -gencode arch=compute_75,code=compute_75
    -gencode arch=compute_80,code=compute_80
    -gencode arch=compute_86,code=compute_86>")
set_target_properties(${PROJECT_NAME} PROPERTIES CUDA_SEPARABLE_COMPILATION ON)
set_target_properties(${PROJECT_NAME} PROPERTIES CUDA_RESOLVE_DEVICE_SYMBOLS ON)
```

```
find_library(CUDA_NVRTC_LIB libnvrtc nvrtc HINTS
↪   "${CUDA_TOOLKIT_ROOT_DIR}/lib64"
↪   "${LIBNVRTC_LIBRARY_DIR}"
↪   "${CUDA_TOOLKIT_ROOT_DIR}/lib/x64" /usr/lib64
↪   /usr/local/cuda/lib64)

target_include_directories(${PROJECT_NAME} PUBLIC
↪   ${CMAKE_CURRENT_SOURCE_DIR}/vkFFT/)
add_library(VkFFT INTERFACE)
target_compile_definitions(VkFFT INTERFACE
↪   -DVKFFT_BACKEND=1)

target_link_libraries(${PROJECT_NAME} PUBLIC
↪   ${CUDA_LIBRARIES} cuda ${CUDA_NVRTC_LIB} VkFFT)
```

(c) HIP API: HIP and HIPRTC. Sample CMakeLists can look like this:

```
list(APPEND CMAKE_PREFIX_PATH /opt/rocm/hip /opt/rocm)
find_package(hip)

target_include_directories(${PROJECT_NAME} PUBLIC
↪   ${CMAKE_CURRENT_SOURCE_DIR}/vkFFT/)
add_library(VkFFT INTERFACE)
target_compile_definitions(VkFFT INTERFACE
↪   -DVKFFT_BACKEND=2)

target_link_libraries(${PROJECT_NAME} PUBLIC hip::host
↪   VkFFT)
```

(d) OpenCL API: OpenCL. Sample CMakeLists can look like this:

```
find_package(OpenCL REQUIRED)

target_include_directories(${PROJECT_NAME} PUBLIC
↪   ${CMAKE_CURRENT_SOURCE_DIR}/vkFFT/)
add_library(VkFFT INTERFACE)
target_compile_definitions(VkFFT INTERFACE
↪   -DVKFFT_BACKEND=3)

target_link_libraries(${PROJECT_NAME} PUBLIC
↪   OpenCL::OpenCL VkFFT)
```

2.2 Fourier Transform Setup

VkFFT follows a plan structure like FFTW/cuFFT with a notable difference - there is a unified interface to all transforms. This means that there are no separate functions like fftPlan1D/fftPlan2D/fftPlanMany/etc. The initialization is done through a single configuration struct - VkFFTConfiguration. Each parameter of it will be covered in detail in this document. Plans in VkFFT are called VkFFTApplication and they are created with a unified initializeVkFFT call.

As the code is written in C, don't forget to zero-initialize used structs!

During the initializeVkFFT(VkFFTApplication* app, VkFFTConfiguration inputLaunchConfiguration) call VkFFT performs kernel generation and compilation from scratch (kernel reuse may be added later). The overall process of initialization looks like this:

1. Get device parameters, perform default initialization of internal copy of configuration struct inside the VkFFTApplication, then fill in user-defined parameters from inputLaunchConfiguration. VkFFTApplication is passed as a pointer, so initializeVkFFT modifies the user-provided application.

2. By default, there are two internal FFT plans created - inverse and forward. Multidimensional FFT is done as a combination of 1D FFTs in each axis direction. For each axis, the VkFFTPlanAxis function is called.

3. VkFFTPlanAxis configures parameters for each axis. It may perform additional memory allocations (see: memory allocated by VkFFT).

4. shaderGenVkFFT generates corresponding to the axis code in a char buffer (each axis may require more than one kernel: see Four-step FFT, Bluestein's algorithm for FFT).

5. Code is then compiled with the run-time compiler of the specified backend.

Once the plan is no longer need, a call to the deleteVkFFT function frees all the allocated resources. There are no processes launched that continue to work outside of the VkFFT related function calls.

2.3 Fourier Transform types and their definitions

VkFFT supports commonly used Complex to complex (C2C), real to complex (R2C), complex to real (C2R) transformations and real to real (R2R) Discrete Cosine Transformations of types II, III and IV. VkFFT uses the same definitions as FFTW, except for the multidimensional FFT axis ordering: in FFTW dimensions are ordered with the decrease in consecutive elements stride, while VkFFT does the opposite - the first axis is the non-strided axis (the one that has elements located consecutively in memory with no gaps, usually named as the X-axis). So, in FFTW dimensions are specified as ZYX and in VkFFT as XYZ. This felt more logical to me - no matter if there are 1, 2 or 3 dimensions, the user can always find the axis with the same stride at the same position. This choice doesn't require any modification in the user's data management - just provide the FFT dimensions in the reverse order to VkFFT.

In addition to up to the 3 dimensions of FFT, VkFFT supports two forms of batching: the number of coordinates and the number of systems. The choice of two distinct batching ways is made to support matrix-vector convolutions, where the kernel is presented as a matrix.

Overall, the layout of VkFFT can be described as WHDCN - width, height, depth, coordinate and number of systems (in order of increasing strides, starting with 1 for width). Coordinate and number of systems can be 1, if the user has 1 as one of the FFT dimensions, the user can omit it from setup altogether as FFT of size 1 produces the same number as the input. Often, the coordinate part of the layout is not used, so the main batching is done by specifying N.

VkFFT assumes that complex numbers are stored consecutively in memory: RIRIRI... where R denotes the real part of the complex number and I denotes the imaginary part. There is no difference between using a float2/double2/half2 container or access memory as float/double/half as long as the byte order remains the same.

This section and the next one will cover the basics of VkFFT data layouts and memory management.

2.3.1 C2C transforms

The base FFT algorithm - C2C in VkFFT has the same definition as FFTW. Forward FFT has the exponent sign -1, while the inverse has the exponent sign 1. By default, the inverse transform is unnormalized. $N_x N_y N_z$ complex numbers map to $N_x N_y N_z$ complex numbers and no additional padding is required. The resulting data order will be the same as in FFTW/cuFFT, unless special parameters are provided in configuration (see: advanced memory management)

2.3.2 R2C/C2R transforms

R2C/C2R transforms can be explained as C2C transforms with imaginary part set to zero. They exploit Hermitian symmetry of the result: $X_k = X^*_{N-k}$ on the non-strided axis (the one that has elements located consecutively in memory with no gaps). This results in s reduction of required memory to store the complex result - we may only store $floor(\frac{N_x}{2}) + 1$ complex numbers instead of N_x. However, this results in memory requirements mismatch between input and output in R2C: $floor(\frac{N_x}{2}) + 1$ complex elements will require $N_x + 2$ real numbers worth of memory for even N_x and $N_x + 1$ real numbers worth of memory for odd N_x. For C2R the situation is reversed. There are two approaches to this problem: pad each sequence of the non-strided axis with zeros to the required length or use out-of-place mode. More information on how to do this will be given in the next section.

2.3.3 R2R (DCT) transforms

R2R transforms in VkFFT are implemented in the form of Discrete cosine transforms of types II, III and IV. Their definitions and transforms results match FFTW:

1. DCT-I: $X_k = x_0 + (-1)^k x_{N-1} + 2\sum\limits_{n=0}^{N-1} x_n cos(\frac{\pi}{N-1} nk)$, not implemented yet
2. DCT-II: $X_k = 2\sum\limits_{n=0}^{N-1} x_n cos(\frac{\pi}{N}(n+\frac{1}{2})k)$, inverse of DCT-III

3. DCT-III: $X_k = x_0 + 2\sum_{n=0}^{N-1} x_n cos(\frac{\pi}{N}n(k+\frac{1}{2}))$, inverse of DCT-II

4. DCT-IV: $X_k = 2\sum_{n=0}^{N-1} x_n cos(\frac{\pi}{N}(n+\frac{1}{2})(k+\frac{1}{2}))$, inverse of DCT-IV (itself)

R2R transforms are performed by redefinition of them to the C2C transforms (internal C2C sequence length can be different from the input R2R sequence length). R2R transform performs a one-to-one mapping between real numbers, so they don't require stride management, unlike R2C/C2R.

2.4 Memory management, data layouts for different transforms

2.4.1 VkFFT buffers

VkFFT allows for explicit control over the data flow, which makes both in-place and out-of-place transforms possible. Buffers are passed to VkFFT as VkBuffer pointer in Vulkan, as double void pointers in CUDA/HIP and as cl_mem pointer in OpenCL. This is done to maintain a uniform data pattern because some of the buffers can be allocated automatically.

The main buffer is called buffer and it always has to be provided, either during the plan creation or when the plan is executed. All calculations are performed in this buffer and it is always overwritten. To do calculations out-of-place, VkFFT provides an option to specify inputBuffer/outputBuffer buffer. The logic behind their usage is fairly simple - the user specifies inputBuffer if the input data has to be read from a buffer, different from the main buffer. As the data is read only once and nothing is written back to the inputBuffer, this allows doing truly out-of-place transformations. The same logic applies to outputBuffer with the difference that it is responsible for the absolute last write of the VkFFT. It is possible to use all three buffers to create complex data management paths.

It must be noted, that sometimes FFT can not be done inside one buffer (see: Four-Step FFT algorithm, Bluestein's algorithm). To compute FFT in these cases, there exists tempBuffer buffer and data is transferred between the main buffer and tempBuffer during the FFT execution. The ordering of transfers between the main buffer and tempBuffer is done in such a way, so the initial data read and final data write are obeying the configuration from the previous paragraph. Users can allocate tempBuffer themselves of some memory that does not have any useful information at the time of FFT execution (the tempBuffer size can depend on the configuration, so this is a rather advanced operation - read more in the advanced memory management section) or allow VkFFT to manage tempBuffer allocation itself (tempBuffer will be freed at the deleteVkFFT call).

To compute convolutions and cross-correlations, a kernel buffer has to be specified. It must have the same layout as the result of the FFT transform.

2.4.2 VkFFT buffers strides. A special case of R2C/C2R transforms

To have better control of memory, the user can specify the strides between consecutive elements of different axis for H (height), D (depth) and C (coordinate) parts of the WHDCN layout (W (width) stride is fixed to be 1, N (number of systems) stride will be consecutive of C in memory if C is used, otherwise N will propagate the previous non-uniform stride multiplied by the corresponding axis length). Strides are specified not in bytes, but in the element type used - similar to the way how the user would access the corresponding element in the array. If all elements are consecutive in C2C, stride for H will be equal to the FFT length of W axis, stride for D will be multiplication of first two FFT axis lengths, stride for C will be multiplication of first three FFT axis lengths, etc. These are the default values of C2C and R2R strides if they are not explicitly specified.

One of the main use-cases of strides comes to solve the R2C/C2R Hermitian symmetry H stride mismatch - for real space, it is equal to N_x real elements and for the frequency space it is equal to $floor(\frac{N_x}{2})+1$ complex numbers. So, with strides it is possible to use a buffer, padded to $2\cdot(floor(\frac{N_x}{2})+1)$ real elements in H stride (all elements between N_x and $2\cdot(floor(\frac{N_x}{2})+1)$ will not be read so it does not matter what data is there before the write stage). All other strides are done as a multiplication between the previous stride and the number of elements in the previous axis. These are the default values of R2C/C2R strides in in-place mode if they are not explicitly specified.

It is possible to specify separate sets of strides for all user-defined buffers: bufferStride for the main buffer, inputStride for input buffer, outputStride for output buffer (kernel stride is assumed to be the same as bufferStride, tempBuffer strides are configured automatically).

For an out-of-place R2C FFT, there is no need to pad buffer with real numbers, so VkFFT uses natural H stride as default there - N_x real elements for real space and $floor(\frac{N_x}{2}) + 1$ complex numbers for the frequency space.

An out-of-place C2R FFT is a more tricky transform. In the multidimensional case, the main buffer will be written to and read from multiple times. The intermediate stores have a complex layout, which requires more space than the output real layout, so in order not to modify the input data, there exist two options. First, pad the real data layout has to $2 \cdot (floor(\frac{N_x}{2}) + 1)$ real elements in H stride (complex buffer will be used as inputBuffer, real buffer as buffer). Second, use the third buffer, so both input and output buffers have their original layouts (complex buffer will be used as inputBuffer, the main buffer for calculations is buffer and output real buffer as outputBuffer).

2.5 VkFFT algorithms

VkFFT implements a wide range of algorithms to compute different types of FFTs but all of them can be reduced to a mixed-radix Cooley-Tukey FFT algorithm in the Stockham autosort form. The main idea behind it is to decompose the sequence as a set of primes, for each of which FFT can be written down exactly. As of now, VkFFT has radix implementations for primes up to 13, so all C2C sequences decomposable into a multiplication of such primes will be done purely with the Stockham algorithm. Below additional algorithms and their use-cases are described.

2.5.1 Bluestein's algorithm

A complex algorithm that is used in cases where the sequence is not decomposable with implemented radix butterflies (currently - primes up to 13). It is derived by replacing $nk = \left(n^2 + k^2 - (n-k)^2\right)/2$ in 1:

$$X_k = \left(e^{-\pi i \frac{k^2}{N}}\right) \sum_{n=0}^{N-1} \left(x_n e^{-\pi i \frac{n^2}{N}}\right) \left(e^{\pi i \frac{(k-n)^2}{N}}\right) = b_k^* \sum_{n=0}^{N-1} a_n b_{k-n} \tag{3}$$

$$a_n = x_n b_n^* \tag{4}$$

$$b_n = e^{\pi i \frac{n^2}{N}} \tag{5}$$

Here FFT is represented as a convolution between two sequences: a_n and b_n, which can be performed by the means of convolution theorem:

$$F\{a * b\} = F\{a\} \cdot F\{b\} \tag{6}$$

By padding a_n and b_n to a sequence length decomposable with implemented radix butterflies with a size of at least $2N-1$ (because the length of b_n is $2N-1$), we can perform FFT of any length. FFT of b_n can be precomputed, so overall this algorithm requires at least 4x the computations and more memory transfers. This algorithm can be combined with all other algorithms implemented in VkFFT. If an FFT can not be done in a single upload, a tempBuffer has to be allocated (because the logical FFT buffer size is bigger than the original system).

2.5.2 The Four-Step FFT algorithm

GPUs and CPUs have a hierarchical memory model - the closer memory to the unit that performs the computations, the faster its speed and the lower the size. So it is advantageous to split FFTs, not to the lowest primes, but to some bigger multiplication of those primes, then upload this subsequence to the closest cache level to the cores and do the final prime split there. The absolute lowest level is the register file, however, it does not allow for thread communications outside the warp. For this purpose, modern GPUs employ shared memory - a fast memory with a bank structure that is visible to all threads in a thread block. The usual sizes of it change on a scale from 16KB to 192KB and it is often beneficial to use it fully. However, if the full sequence can not fit inside the shared memory, FFT has to be done in multiple uploads - with the Four Step FFT algorithm. The main idea behind it is to represent a big 1D sequence as a 2D (or 3D for the three-upload scheme) FFT - we first do FFT along the columns, then the rows, then transpose the result and multiply by a special set of phase vectors. Similar decomposition idea as the main Cooley-Tukey algorithm. However, performing transpositions in-place is a complicated task - especially for a non-trivial ratio between dimensions. It will also require an additional read/write stage, as it can not be merged with the last write of the FFT algorithm. The easiest and the most performant solution is to use a tempBuffer (it is the main reason for having this functionality, actually) and store intermediate FFT results out-of-place. This way the last transposition step can be merged with the write step, as we can overwrite the output buffer without losing data.

To estimate if your sequence size is single upload or not, divide the amount of available shared memory (48KB - Nvidia GPUs with Vulkan/OpenCL API, 64KB - AMD GPUs, 100KB - Nvidia GPUs in CUDA API) by the complex size used for calculations (8 byte - single precision, 16 byte - double precision). For 64KB of shared memory, we get 8192 as max single upload single-precision non-strided FFT, 4096 for double precision. For strided axes (H and D parts of the layout) these numbers have to be divided by 4 and 2 respectively to achieve coalescing, resulting in 2048 length for single upload in both precisions. For more information on coalescing see: coalescing API reference.

In the case of the Four-Step FFT algorithm, tempBuffer size has to be at least the same as the default main buffer size. It does not matter how many uploads are in the Four Step FFT algorithm - only a single tempBuffer is required. In this document, all systems that can fit in the shared memory entirely and be done without the Four Step FFT algorithm (and multiple uploads) are called single upload systems.

If the last transposition is not required (the output data is allowed to be in not unshuffled form) it can be disabled during the configuration phase. This way tempBuffer will not be needed and all computations will be done in-place (unless Bluestein's algorithm is used). An example use-case of this is convolutions - if the kernel is computed with the same operation ordering, point-wise multiplication in the frequency domain is not dependent on the correct data ordering and the inverse FFT will restore the original layout.

2.5.3 R2C/C2R FFTs

A typical approach to a single upload R2C/C2R system is to just set the imaginary part to zero inside the shared memory and do a simple C2C transform. This doesn't affect the amount of memory transferred from VRAM and is not a bad approach as FFT is a memory-bound algorithm, however, this can be improved in multidimensional (in HDCN part of the layout) case by the composition of a single C2C sequence from two real sequences and some write for R2C/read for C2R post-processing. Both of these algorithms are implemented in VkFFT. Note, that R2C/C2R only affects the non-strided axis (W). All strided axes are still done as C2C.

2.5.4 R2C/C2R multi-upload FFT algorithm

For even sequences there exists an easy mapping between R2C/C2R FFTs and the C2C of half the size. In this case, all even indices (starting from 0) are read as the real values of a complex number and all odd indices are read as the imaginary values. This C2C sequence can be done with the help of the Four-Step FFT algorithm. When FFT is done, separate post-processing for R2C/pre-processing for C2R is applied.

2.5.5 R2R Discrete Cosine Transforms

There exist many different mappings between DCT and FFT. As of now, VkFFT has the following algorithms implemented (all single-upload for now):

- DCT-II/DCT-III - mapping between R2R and C2C of the same length. For non-strided axis can use an optimization similar to the R2C/C2R multidimensional case (setting the imaginary part to the next FFT sequence).
- DCT-IV - implemented only for even sizes, mapping between R2R and C2C sequence of half-length.

2.5.6 Register overutilization

Not an FFT algorithm by itself, but an optimization to do bigger sequences in a single upload instead of switching to the Four Step FFT algorithm. The main idea behind it is to use a register file (which is often bigger than the amount of shared memory) to store the sequence and use shared memory only as a communication buffer. This is useful in Vulkan and OpenCL APIs on Nvidia GPU, as they are only allowed to allocate 48KB of shared memory with a register file having the size of 256KB.

2.5.7 Zero padding

Not an FFT algorithm by itself, but a memory management optimization. If the user's system has parts that are known to be zero - for example, when an open system is modeled, to avoid a circular part of the FFT system has to be padded with zeros up to 2x in each direction. VkFFT can omit sequences full of zeros and don't 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 speed increase in the 2D case and up to 3x increase in the 3D case.

2.5.8 Convolution and cross-correlation support

With the help of the Convolution theorem, which states that the Fourier transform of a convolution is the pointwise product of signals Fourier transforms, it is possible to perform convolution with $NlogN$ complexity, compared to N^2 complexity of the simple multiplication approach. This is extremely useful for kernels spanning more than 50 elements in size. VkFFT can merge the last step FFT, kernel multiplication in the Fourier domain and the first step of inverse FFT to provide substantial memory transfer savings. Moreover, FFTs of big sequences can be performed without data reordering, which results in a better locality.

2.6 VkFFT accuracy

To measure how VkFFT (single/double/half precision) results compare to cuFFT/rocFFT (single/double/half precision) and FFTW (double precision), multiple sets of systems covering full supported C2C/R2C+C2R/R2R FFT range are filled with random complex data on the scale of [-1,1] and one transform was performed on each system. Samples 11(single), 12(double), 13(half), 14(non-power of 2 C2C, single), 15(R2C+C2R, single), 16(DCT-

II/III/IV, single), 17(DCT-II/III/IV, double), 18(non-power of 2 C2C, double) are available in VkFFT Benchmark Suite to perform VkFFT verification on any of the target platforms. Overall, the Cooley-Tukey algorithm (Stockham autosort) exhibits logarithmic relative error scaling, similar to those of other GPU FFT libraries. Typically, the more computationally expensive algorithm is - the worse its precision is. So, Bluestein's algorithm has lower accuracy than Stockham autosort algorithm.

Single precision in VkFFT supports two modes of calculation - by using the on-chip Special Function Units that can compute sines and cosines on the go or by using the precomputed on CPU look-up tables. For Nvidia and AMD GPUs, SFU provide great precision, while Intel iGPUs and mobile GPUs must use LUT to perform FFTs correctly.

Double precision in VkFFT also supports two modes of calculation - by using polynomial sincos approximation and computing them on-chip or by using precomputed LUT as well. The second option is the better one, as polynomial sincos approximation is too compute-heavy for modern GPUs. It is selected by default on all devices.

Half precision is currently only supported in the Vulkan backend and is often experiencing precision problems with the first number of the resulting FFT sequence, which is the sum of all input numbers. Half precision is implemented only as a memory trick - all on-chip computations are done in single precision, but this doesn't help with the first number problem. Half precision can use SFU or LUT as well.

VkFFT also supports mixed-precision operations, where memory storing is done at lower precision, compared to the on-chip calculations. For example, it is possible to read data in single precision, do calculations in double and store data back in single precision.

2.7 VkFFT additional memory allocations

In this section, all GPU memory allocations that are done by VkFFT are described. There are up to three situations when VkFFT allocates memory. All of the VkFFT allocated memory is freed at the deleteVkFFT call.

2.7.1 LUT allocations

This memory is used to store precomputed twiddle factors and phase vectors used during the computation. This buffer can have:

- twiddle factors for each radix stage of Stockham FFT calculation
- phase vectors used in the Four Step FFT algorithm between stages
- phase vectors used in DCT-II/III/IV to perform a mapping between R2R and C2C

VkFFT manages LUT allocations by itself and they are performed during the initializeVkFFT call. LUT are allocated per axis, though some of them can be reused if the axes have the same LUT. Inverse and forward FFT plans share the same LUT (conjugation is performed on-chip).

2.7.2 The Four-Step FFT algorithm - tempBuffer allocation

To perform the merging of the transposition with the last upload of an axis, VkFFT requires additional memory to mimic an out-of-place execution. This memory is located in tempBuffer and has to be of at least the same size as the main buffer. It is possible for the users to allocate it themselves, though if this is not done, VkFFT can do the allocation automatically (the size of the tempBuffer will be the same as the main buffer, unless the logical dimensions of FFT are bigger than user-defined - then, it will allocate the system with the minimal size, that can cover maximal logical system size used in any of the axes - see next subsection).

2.7.3 Bluestein's buffers allocation

To do Bluestein's FFT algorithm, precomputed sequences $b_n = e^{\pi i \frac{n^2}{N}}$, $FFT(b_n)$ and $iFFT(b_n)$ are required. For each axis, they can be different and are computed separately (unless VkFFT can determine that they match, then the buffers are allocated only once). Notably, as Bluestein's algorithm pads the sequence length to at least $2N-1$, if it can not be done in a single upload and the Four Step algorithm has to be used, the intermediate storage required will be bigger than the main buffer size. In this case, tempBuffer must always be allocated. As the padded sequence can be different for each of the dimensions, the required size of the tempBuffer will also vary. VkFFT determines the biggest size needed among axes and allocated tempBuffer of this size.

3 VkFFT API Reference

This section covers error codes, API functions that can be used by the user and configuration parameters.

3.1 Return value VkFFTResult

All VkFFT Library return values except for VKFFT_SUCCESS are used in case of a failure and provide information on what has gone wrong. VkFFTResult is unified among different backends, though some of its values may not be used in specific backends. Possible return values of VkFFTResult are defined as following:

```
typedef enum VkFFTResult {
VKFFT_SUCCESS = 0,  // The VkFFT operation was successful
VKFFT_ERROR_MALLOC_FAILED = 1,  // Some malloc call inside
 ↪  VkFFT has failed. Report this to the GitHub repo
VKFFT_ERROR_INSUFFICIENT_CODE_BUFFER = 2,   // Generated
 ↪  kernel is bigger than default kernel array. Increase it
 ↪  with maxCodeLength parameter of configuration.
VKFFT_ERROR_INSUFFICIENT_TEMP_BUFFER = 3,   // Temporary
 ↪  string used in kernel generation is bigger than default
 ↪  temporary string array. Increase it with maxTempLength
 ↪  parameter of configuration.
```

```
VKFFT_ERROR_PLAN_NOT_INITIALIZED = 4,   // Code attempts to use
 ↪  uninitialized plan (it is zero inside
 ↪  VkFFTApplication)
VKFFT_ERROR_NULL_TEMP_PASSED = 5,   // Internal kernel
 ↪  generation error
VKFFT_ERROR_INVALID_PHYSICAL_DEVICE = 1001,     // No physical
 ↪  device is provided (Vulkan API)
VKFFT_ERROR_INVALID_DEVICE = 1002,  // No device is provided
 ↪  (All APIs)
VKFFT_ERROR_INVALID_QUEUE = 1003,   // No queue is provided
 ↪  (Vulkan API)
VKFFT_ERROR_INVALID_COMMAND_POOL = 1004,    // No command pool
 ↪  is provided (Vulkan API)
VKFFT_ERROR_INVALID_FENCE = 1005,   // No fence is provided
 ↪  (Vulkan API)
VKFFT_ERROR_ONLY_FORWARD_FFT_INITIALIZED = 1006,    // VkFFT
 ↪  tries to access inverse FFT plan, when appliction is
 ↪  created with makeForwardPlanOnly flag
VKFFT_ERROR_ONLY_INVERSE_FFT_INITIALIZED = 1007,    // VkFFT
 ↪  tries to access forward FFT plan, when appliction is
 ↪  created with makeInversePlanOnly flag
VKFFT_ERROR_INVALID_CONTEXT = 1008,     // No context is
 ↪  provided (OpenCL API)
VKFFT_ERROR_INVALID_PLATFORM = 1009,    // No platform is
 ↪  provided (OpenCL API)
VKFFT_ERROR_EMPTY_FFTdim = 2001,    // Number of dimensions is
 ↪  not provided in the configuration
VKFFT_ERROR_EMPTY_size = 2002,  // Array of dimensions is not
 ↪  provided in the configuration
VKFFT_ERROR_EMPTY_bufferSize = 2003,    // Buffer size has to
 ↪  be provided during the application creation
VKFFT_ERROR_EMPTY_buffer = 2004,    // Buffer has te be
 ↪  specified either at the application creation stage or
 ↪  during launch through VkFFTLaunchParams struct
VKFFT_ERROR_EMPTY_tempBufferSize = 2005,    // Same error as
 ↪  VKFFT_ERROR_EMPTY_bufferSize if userTempBuffer is enabled
VKFFT_ERROR_EMPTY_tempBuffer = 2006,    // Same error as
 ↪  VKFFT_ERROR_EMPTY_buffer if userTempBuffer is enabled
VKFFT_ERROR_EMPTY_inputBufferSize = 2007,   // Same error as
 ↪  VKFFT_ERROR_EMPTY_bufferSize if isInputFormatted is enabled
VKFFT_ERROR_EMPTY_inputBuffer = 2008,   // Same error as
 ↪  VKFFT_ERROR_EMPTY_buffer if isInputFormatted is enabled
```

```
VKFFT_ERROR_EMPTY_outputBufferSize = 2009,  // Same error as
↪  VKFFT_ERROR_EMPTY_bufferSize if isOutputFormatted is
↪  enabled
VKFFT_ERROR_EMPTY_outputBuffer = 2010,  // Same error as
↪  VKFFT_ERROR_EMPTY_buffer if isOutputFormatted is enabled
VKFFT_ERROR_EMPTY_kernelSize = 2011,    // Same error as
↪  VKFFT_ERROR_EMPTY_bufferSize if performConvolution is
↪  enabled
VKFFT_ERROR_EMPTY_kernel = 2012,    // Same error as
↪  VKFFT_ERROR_EMPTY_buffer if performConvolution is enabled
VKFFT_ERROR_UNSUPPORTED_RADIX = 3001,   // VkFFT has
↪  encountered unsupported radix (more than 13) during
↪  decomposition and Bluestein's FFT fallback did not work
VKFFT_ERROR_UNSUPPORTED_FFT_LENGTH = 3002,  // VkFFT can not do
↪  this sequence length currently - it requires mor than
↪  three-upload Four step FFT
VKFFT_ERROR_UNSUPPORTED_FFT_LENGTH_R2C = 3003,  // VkFFT can
↪  not do this sequence length currently - odd multi-upload
↪  R2C/C2R FFTs
VKFFT_ERROR_UNSUPPORTED_FFT_LENGTH_DCT = 3004,  // VkFFT can
↪  not do this sequence length currently - multi-upload R2R
↪  transforms, odd DCT-IV transforms
VKFFT_ERROR_UNSUPPORTED_FFT_OMIT = 3005,    // VkFFT can not
↪  omit sequences in convolution calculations and R2C/C2R case
VKFFT_ERROR_FAILED_TO_ALLOCATE = 4001,  // VkFFT failed to
↪  allocate GPU memory
VKFFT_ERROR_FAILED_TO_MAP_MEMORY = 4002,    // 4002-4052 are
↪  handlers for errors of used backend APIs. They may indicate
↪  a driver failure. If they are thrown - report to the GitHub
↪  repo
VKFFT_ERROR_FAILED_TO_ALLOCATE_COMMAND_BUFFERS = 4003,
VKFFT_ERROR_FAILED_TO_BEGIN_COMMAND_BUFFER = 4004,
VKFFT_ERROR_FAILED_TO_END_COMMAND_BUFFER = 4005,
VKFFT_ERROR_FAILED_TO_SUBMIT_QUEUE = 4006,
VKFFT_ERROR_FAILED_TO_WAIT_FOR_FENCES = 4007,
VKFFT_ERROR_FAILED_TO_RESET_FENCES = 4008,
VKFFT_ERROR_FAILED_TO_CREATE_DESCRIPTOR_POOL = 4009,
VKFFT_ERROR_FAILED_TO_CREATE_DESCRIPTOR_SET_LAYOUT = 4010,
VKFFT_ERROR_FAILED_TO_ALLOCATE_DESCRIPTOR_SETS = 4011,
VKFFT_ERROR_FAILED_TO_CREATE_PIPELINE_LAYOUT = 4012,
VKFFT_ERROR_FAILED_SHADER_PREPROCESS = 4013,
VKFFT_ERROR_FAILED_SHADER_PARSE = 4014,
VKFFT_ERROR_FAILED_SHADER_LINK = 4015,
VKFFT_ERROR_FAILED_SPIRV_GENERATE = 4016,
```

```
VKFFT_ERROR_FAILED_TO_CREATE_SHADER_MODULE = 4017,
VKFFT_ERROR_FAILED_TO_CREATE_INSTANCE = 4018,
VKFFT_ERROR_FAILED_TO_SETUP_DEBUG_MESSENGER = 4019,
VKFFT_ERROR_FAILED_TO_FIND_PHYSICAL_DEVICE = 4020,
VKFFT_ERROR_FAILED_TO_CREATE_DEVICE = 4021,
VKFFT_ERROR_FAILED_TO_CREATE_FENCE = 4022,
VKFFT_ERROR_FAILED_TO_CREATE_COMMAND_POOL = 4023,
VKFFT_ERROR_FAILED_TO_CREATE_BUFFER = 4024,
VKFFT_ERROR_FAILED_TO_ALLOCATE_MEMORY = 4025,
VKFFT_ERROR_FAILED_TO_BIND_BUFFER_MEMORY = 4026,
VKFFT_ERROR_FAILED_TO_FIND_MEMORY = 4027,
VKFFT_ERROR_FAILED_TO_SYNCHRONIZE = 4028,
VKFFT_ERROR_FAILED_TO_COPY = 4029,
VKFFT_ERROR_FAILED_TO_CREATE_PROGRAM = 4030,
VKFFT_ERROR_FAILED_TO_COMPILE_PROGRAM = 4031,
VKFFT_ERROR_FAILED_TO_GET_CODE_SIZE = 4032,
VKFFT_ERROR_FAILED_TO_GET_CODE = 4033,
VKFFT_ERROR_FAILED_TO_DESTROY_PROGRAM = 4034,
VKFFT_ERROR_FAILED_TO_LOAD_MODULE = 4035,
VKFFT_ERROR_FAILED_TO_GET_FUNCTION = 4036,
VKFFT_ERROR_FAILED_TO_SET_DYNAMIC_SHARED_MEMORY = 4037,
VKFFT_ERROR_FAILED_TO_MODULE_GET_GLOBAL = 4038,
VKFFT_ERROR_FAILED_TO_LAUNCH_KERNEL = 4039,
VKFFT_ERROR_FAILED_TO_EVENT_RECORD = 4040,
VKFFT_ERROR_FAILED_TO_ADD_NAME_EXPRESSION = 4041,
VKFFT_ERROR_FAILED_TO_INITIALIZE = 4042,
VKFFT_ERROR_FAILED_TO_SET_DEVICE_ID = 4043,
VKFFT_ERROR_FAILED_TO_GET_DEVICE = 4044,
VKFFT_ERROR_FAILED_TO_CREATE_CONTEXT = 4045,
VKFFT_ERROR_FAILED_TO_CREATE_PIPELINE = 4046,
VKFFT_ERROR_FAILED_TO_SET_KERNEL_ARG = 4047,
VKFFT_ERROR_FAILED_TO_CREATE_COMMAND_QUEUE = 4048,
VKFFT_ERROR_FAILED_TO_RELEASE_COMMAND_QUEUE = 4049,
VKFFT_ERROR_FAILED_TO_ENUMERATE_DEVICES = 4050,
VKFFT_ERROR_FAILED_TO_GET_ATTRIBUTE = 4051,
VKFFT_ERROR_FAILED_TO_CREATE_EVENT = 4052
} VkFFTResult;
```

3.2 VkFFT application management functions

VkFFT has a unified plan management model - all different transform types/ dimensionalities/ precision use the same calls with configuration done through VkFFTConfiguration struct. This section shows how to initialize/use/free VkFFT with this unified model, while the next one will go into how to configure VkFFTConfiguration correctly. All of the functions operate on VkFFTApplication and VkFFTConfiguration assuming they have been zero-initialized before usage, so do not forget to do this when initializing:

```
VkFFTConfiguration configuration = {};
VkFFTApplication app = {};
```

3.2.1 Function initializeVkFFT()

```
VkFFTResult initializeVkFFT(VkFFTApplication* app,
 ↪  VkFFTConfiguration inputLaunchConfiguration)
```

Creates an FFT application (collection of forward and inverse plans). As forward and inverse FFTs may have different memory layouts, can have different normalizations - they are done as separate internal plans inside VkFFTApplication. This call assumes the application to be zero-initialized, so can be only done once on a particular application, until it is deleted.

If the initializeVkFFT call fails, it frees all allocated by VkFFT CPU/GPU resources and sets the application to zero. VkFFTResult is returned with an error code corresponding to what went wrong.

In case of success, VkFFTApplication will contain initialized plans with compiled kernels ready for execution with VKFFT_SUCCESS returned.

3.2.2 Function VkFFTAppend()

```
VkFFTResult VkFFTAppend(VkFFTApplication* app, int inverse,
 ↪  VkFFTLaunchParams* launchParams)
```

Performs FFT in the int inverse direction (-1 for forward FFT, 1 for inverse FFT). FFT plans are selected from the VkFFTApplication collection automatically. VkFFTApplication must be initialized with initializeVkFFT call before. VkFFTLaunchParams struct allows for pre-launch configuration of some parameters, namely:

- buffer - similar to how FFTW/cuFFT expects input/output data pointers in *execC2C (and other) function calls, VkFFT allows specifying memory used for computations at launch. It must have the same size/layout/strides as defined during the application creation.
- inputBuffer/outputBuffer/tempBuffer/kernel - other buffers can also be specified at launch. In addition to them having the same size/layout/strides as defined during the application creation, the application must be created with flags enabling the corresponding buffer usage: isInputFormatted/isOutputFormatted/userTempBuffer/performConvolution respectively.

Depending on the API, the execution model may vary and require additional information at launch:

- Vulkan API: VkFFT appends a sequence of vkCmdDispatch calls to the user-defined VkCommandBuffer (with respective push constants/descriptor sets/pipelines/memory barriers bindings). VkCommandBuffer must be provided as a pointer in VkFFTLaunchParams.

VkCommandBuffer must be in the writing stage, started with vkBeginCommandBuffer call. After VkFFTAppend has finished, provided VkCommandBuffer will contain a sequence of operations performing FFT. The first call of the sequence has no input memory barrier, the last call has one, ensuring FFT has finished execution.

- CUDA/HIP API: if the user wants to use streams, they have to be provided during the application configuration stage. VkFFTAppend performs a series of cuLaunchKernel, which are sequential if appended to one stream and synchronized if appended to multiple streams.
- OpenCL API: similar to Vulkan, VkFFT appends a sequence of clEnqueueNDRangeKernel calls to user-defined cl_command_queue. Currently, they are all assumed to be sequential. cl_command_queue must be provided as a pointer in VkFFTLaunchParams.

If VkFFT fails during the VkFFTAppend call, it will not free the application and allocated there resources - use a separate call for that.

3.2.3 Function deleteVkFFT()

```
void deleteVkFFT(VkFFTApplication* app)
```

Performs deallocation of resources used in the provided application. Returns application to the zero-initialized state.

3.2.4 Funcion VkFFTGetVersion()

```
int VkFFTGetVersion()
```

Returns the version of the VkFFT library in the X.XX.XX format (without dots).

3.3 VkFFT configuration

This section will cover all the parameters that can be specified in the VkFFTConfiguration struct. It will start with a short description of the struct (intended to be used as a cheat sheet), then go for each field in detail.

```
typedef struct {
// Required parameters:
uint64_t FFTdim;    // FFT dimensionality (1, 2 or 3)
uint64_t size[3];   // WHD - system dimensions
#if(VKFFT_BACKEND==0) //Vulkan API
VkPhysicalDevice* physicalDevice;   // Pointer to Vulkan
 ↪  physical device, obtained from vkEnumeratePhysicalDevices
VkDevice* device;   // Pointer to Vulkan device, created with
 ↪  vkCreateDevice
VkQueue* queue; // Pointer to Vulkan queue, created with
 ↪  vkGetDeviceQueue
```

```
VkCommandPool* commandPool;     // Pointer to Vulkan command
 ↪  pool, created with vkCreateCommandPool
VkFence* fence; // Pointer to Vulkan fence, created with
 ↪  vkCreateFence
uint64_t isCompilerInitialized;     // Specify if glslang
 ↪  compiler has been intialized before (0 - off, 1 - on).
 ↪  Default 0
#elif(VKFFT_BACKEND==1) //CUDA API
CUdevice* device;   // Pointer to CUDA device, obtained from
 ↪  cuDeviceGet
cudaStream_t* stream;   // Pointer to streams (can be more than
 ↪  1), where to execute the kernels. Deafult 0
uint64_t num_streams;   // Try to submit CUDA kernels in
 ↪  multiple streams for asynchronous execution. Default 1
#elif(VKFFT_BACKEND==2) //HIP API
hipDevice_t* device;    // Pointer to HIP device, obtained from
 ↪  hipDeviceGet
hipStream_t* stream;    // Pointer to streams (can be more than
 ↪  1), where to execute the kernels. Deafult 0
uint64_t num_streams;   // Try to submit HIP kernels in
 ↪  multiple streams for asynchronous execution. Default 1
#elif(VKFFT_BACKEND==3) //OpenCL API
cl_platform_id* platform;   // Pointer to OpenCL platform,
 ↪  obtained from clGetPlatformIDs
cl_device_id* device;   // Pointer to OpenCL device, obtained
 ↪  from clGetDeviceIDs
cl_context* context;    // Pointer to OpenCL context, obtained
 ↪  from clCreateContext
#endif

// Data parameters (buffers can be specified at launch):
uint64_t userTempBuffer;    // Buffer allocated by app
 ↪  automatically if needed to reorder Four step algorithm.
 ↪  Setting to non zero value enables manual user allocation (0
 ↪  - off, 1 - on)
uint64_t bufferNum;     // Multiple buffer sequence storage is
 ↪  Vulkan only. Default 1
uint64_t tempBufferNum;     // Multiple buffer sequence
 ↪  storage is Vulkan only. Default 1, buffer allocated by app
 ↪  automatically if needed to reorder Four step algorithm.
 ↪  Setting to non zero value enables manual user allocation
 ↪
uint64_t inputBufferNum;    // Multiple buffer sequence storage
 ↪  is Vulkan only. Default 1, if isInputFormatted is enabled
```

```
uint64_t outputBufferNum;   // Multiple buffer sequence storage
 ↪  is Vulkan only. Default 1, if isOutputFormatted is enabled
uint64_t kernelNum;     // Multiple buffer sequence storage is
 ↪  Vulkan only. Default 1, if performConvolution is enabled
uint64_t* bufferSize;   // Array of buffers sizes in bytes
uint64_t* tempBufferSize;   // Array of temp buffers sizes in
 ↪  bytes. Default set to bufferSize sum, buffer allocated by
 ↪  app automatically if needed to reorder Four step algorithm.
 ↪  Setting to non zero value enables manual user allocation
uint64_t* inputBufferSize;  // Array of input buffers sizes in
 ↪  bytes, if isInputFormatted is enabled
uint64_t* outputBufferSize;     // Array of output buffers
 ↪  sizes in bytes, if isOutputFormatted is enabled
uint64_t* kernelSize;   // Array of kernel buffers sizes in
 ↪  bytes, if performConvolution is enabled
#if(VKFFT_BACKEND==0) //Vulkan API
VkBuffer* buffer;   // Pointer to array of buffers (or one
 ↪  buffer) used for computations
VkBuffer* tempBuffer;   // Needed if reorderFourStep is enabled
 ↪  to transpose the array. Same sum size or bigger as buffer
 ↪  (can be split in multiple). Default 0. Setting to non zero
 ↪  value enables manual user allocation
VkBuffer* inputBuffer;  // Pointer to array of input buffers
 ↪  (or one buffer) used to read data from if isInputFormatted
 ↪  is enabled
VkBuffer* outputBuffer;     // Pointer to array of output
 ↪  buffers (or one buffer) used to write data to if
 ↪  isOutputFormatted is enabled
VkBuffer* kernel;   // Pointer to array of kernel buffers (or
 ↪  one buffer) used to read kernel data from if
 ↪  performConvolution is enabled
#elif(VKFFT_BACKEND==1) //CUDA API
void** buffer;  // Pointer to device buffer used for
 ↪  computations
void** tempBuffer;  // Needed if reorderFourStep is enabled to
 ↪  transpose the array. Same size as buffer. Default 0.
 ↪  Setting to non zero value enables manual user allocation
void** inputBuffer;     // Pointer to device buffer used to
 ↪  read data from if isInputFormatted is enabled
void** outputBuffer;    // Pointer to device buffer used to
 ↪  write data to if isOutputFormatted is enabled
void** kernel;  // Pointer to device buffer used to read kernel
 ↪  data from if performConvolution is enabled
#elif(VKFFT_BACKEND==2) //HIP API
```

```
void** buffer;  // Pointer to device buffer used for
 ↪  computations
void** tempBuffer;  // Needed if reorderFourStep is enabled to
 ↪  transpose the array. Same size as buffer. Default 0.
 ↪  Setting to non zero value enables manual user allocation
void** inputBuffer;     // Pointer to device buffer used to
 ↪  read data from if isInputFormatted is enabled
void** outputBuffer;    // Pointer to device buffer used to
 ↪  write data to if isOutputFormatted is enabled
void** kernel;  // Pointer to device buffer used to read kernel
 ↪  data from if performConvolution is enabled
#elif(VKFFT_BACKEND==3) //OpenCL API
cl_mem* buffer; // Pointer to device buffer used for
 ↪  computations
cl_mem* tempBuffer; // Needed if reorderFourStep is enabled to
 ↪  transpose the array. Same size as buffer. Default 0.
 ↪  Setting to non zero value enables manual user allocation
cl_mem* inputBuffer;    // Pointer to device buffer used to
 ↪  read data from if isInputFormatted is enabled
cl_mem* outputBuffer;   // Pointer to device buffer used to
 ↪  write data to if isOutputFormatted is enabled
cl_mem* kernel; // Pointer to device buffer used to read kernel
 ↪  data from if performConvolution is enabled
#endif

// Optional: (default 0 if not stated otherwise)
uint64_t coalescedMemory;   // In bytes, for Nvidia and AMD is
 ↪  equal to 32, Intel is equal 64, scaled for half precision.
 ↪  Going to work regardless, but if specified by user
 ↪  correctly, the performance will be higher.
uint64_t aimThreads;    // Aim at this many threads per block.
 ↪  Default 128
uint64_t numSharedBanks;    // How many banks shared memory
 ↪  has. Default 32
uint64_t inverseReturnToInputBuffer;    // return data to the
 ↪  input buffer in inverse transform (0 - off, 1 - on).
 ↪  isInputFormatted must be enabled
uint64_t numberBatches;     // N - used to perform multiple
 ↪  batches of initial data. Default 1
uint64_t useUint64;     // Use 64-bit addressing mode in
 ↪  generated kernels
uint64_t omitDimension[3];  // Disable FFT for this dimension
 ↪  (0 - FFT enabled, 1 - FFT disabled). Default 0. Doesn't
 ↪  work for R2C for now. Doesn't work with convolutions.
```

```
uint64_t fixMaxRadixBluestein;  // Controls the padding of
 ↪  sequences in Bluestein convolution. If specified, padded
 ↪  sequence will be made of up to fixMaxRadixBluestein primes.
 ↪  Default: 2 for up to 1048576 combined dimension FFT system,
 ↪  7 after. Min = 2, Max = 13.
uint64_t doublePrecision;   // Perform calculations in double
 ↪  precision (0 - off, 1 - on).
uint64_t halfPrecision;     // Perform calculations in half
 ↪  precision (0 - off, 1 - on)
uint64_t halfPrecisionMemoryOnly;   // Use half precision only
 ↪  as input/output buffer. Input/Output have to be allocated
 ↪  as half, buffer/tempBuffer have to be allocated as float
 ↪  (out-of-place mode only). Specify isInputFormatted and
 ↪  isOutputFormatted to use (0 - off, 1 - on)
uint64_t doublePrecisionFloatMemory;    // Use FP64 precision
 ↪  for all calculations, while all memory storage is done in
 ↪  FP32.
uint64_t performR2C;    // Perform R2C/C2R decomposition (0 -
 ↪  off, 1 - on)
uint64_t performDCT;    // Perform DCT transformation (X - DCT
 ↪  type, 1-4)
uint64_t disableMergeSequencesR2C;  // Disable merging of two
 ↪  real sequences to reduce calculations (0 - off, 1 - on)
uint64_t normalize;     // Normalize inverse transform (0 -
 ↪  off, 1 - on)
uint64_t disableReorderFourStep;    // Disables unshuffling of
 ↪  Four step algorithm. Requires tempbuffer allocation (0 -
 ↪  off, 1 - on)
uint64_t useLUT;    // Switches from calculating sincos to
 ↪  using precomputed LUT tables (0 - off, 1 - on). Configured
 ↪  by initialization routine
uint64_t makeForwardPlanOnly;   // Generate code only for
 ↪  forward FFT (0 - off, 1 - on)
uint64_t makeInversePlanOnly;   // Generate code only for
 ↪  inverse FFT (0 - off, 1 - on)
uint64_t bufferStride[3];   // Buffer strides - default set to
 ↪  x - x*y - x*y*z values
uint64_t isInputFormatted;  // Specify if input buffer is
 ↪  padded - 0 - padded, 1 - not padded. For example if it is
 ↪  not padded for R2C if out-of-place mode is selected (only
 ↪  if numberBatches==1 and numberKernels==1)
```

```
uint64_t isOutputFormatted;     // Specify if output buffer is
↪  padded - 0 - padded, 1 - not padded. For example if it is
↪  not padded for R2C if out-of-place mode is selected (only
↪  if numberBatches==1 and numberKernels==1)
uint64_t inputBufferStride[3];  // Input buffer strides. Used
↪  if isInputFormatted is enabled. Default set to bufferStride
↪  values
uint64_t outputBufferStride[3];     // Output buffer strides.
↪  Used if isInputFormatted is enabled. Default set to
↪  bufferStride values
uint64_t considerAllAxesStrided;    // Will create plan for
↪  non-strided axis similar as a strided axis - used with
↪  disableReorderFourStep to get the same layout for Bluestein
↪  kernel (0 - off, 1 - on)
uint64_t keepShaderCode;    // Will keep shader code and print
↪  all executed shaders during the plan execution in order (0
↪  - off, 1 - on)

// Optional zero padding control parameters: (default 0 if not
↪  stated otherwise)
uint64_t performZeropadding[3];     // Don't read some
↪  data/perform computations if some input sequences are
↪  zeropadded for each axis (0 - off, 1 - on)
uint64_t fft_zeropad_left[3];   // Specify start boundary of
↪  zero block in the system for each axis
uint64_t fft_zeropad_right[3];  // Specify end boundary of zero
↪  block in the system for each axis
uint64_t frequencyZeroPadding;  // Set to 1 if zeropadding of
↪  frequency domain, default 0 - spatial zeropadding

// Optional convolution control parameters: (default 0 if not
↪  stated otherwise)
uint64_t performConvolution;    // Perform convolution in this
↪  application (0 - off, 1 - on). Disables reorderFourStep
↪  parameter
uint64_t coordinateFeatures;    // C - coordinate, or dimension
↪  of features vector. In matrix convolution - size of a
↪  vector
uint64_t matrixConvolution;     // If equal to 2 perform 2x2,
↪  if equal to 3 perform 3x3 matrix-vector convolution.
↪  Overrides coordinateFeatures
uint64_t symmetricKernel;   // Specify if kernel in 2x2 or 3x3
↪  matrix convolution is symmetric
```

```
uint64_t numberKernels;     // N - only used in convolution
 ↪  step - specify how many kernels were initialized before.
 ↪  Expands one input to multiple (batched) output
uint64_t kernelConvolution;     // Specify if this application
 ↪  is used to create kernel for convolution, so it has the
 ↪  same properties. performConvolution has to be set to 0 for
 ↪  kernel creation

// Register overutilization (experimental): (default 0 if not
 ↪  stated otherwise)
uint64_t registerBoost;     // Specify if register file size is
 ↪  bigger than shared memory and can be used to extend it X
 ↪  times (on Nvidia 256KB register file can be used instead of
 ↪  32KB of shared memory, set this constant to 4 to emulate
 ↪  128KB of shared memory). Defaults: Nvidia - 4 in
 ↪  Vulkan/OpenCL, 1 in CUDA backend; AMD - 2 if shared memory
 ↪  >= 64KB, else 4 in Vulkan/OpenCL backend, 1 in HIP backend;
 ↪  Intel - 1 if shared memory >= 64KB, else 2 in Vulkan/OpenCL
 ↪  backend; Default 1
uint64_t registerBoostNonPow2;  // Specify if register
 ↪  overutilization should be used on non power of 2 sequences
 ↪  (0 - off, 1 - on)
uint64_t registerBoost4Step;    // Specify if register file
 ↪  overutilization should be used in big sequences (>2^14),
 ↪  same definition as registerBoost. Default 1
//not used techniques:
uint64_t swapTo3Stage4Step;     // Specify at which power of 2
 ↪  to switch from 2 upload to 3 upload 4-step FFT, in case if
 ↪  making max sequence size lower than coalesced sequence
 ↪  helps to combat TLB misses. Default 0 - disabled. Must be
 ↪  at least 17
uint64_t performHalfBandwidthBoost;     // Try to reduce
 ↪  coalsesced number by a factor of 2 to get bigger sequence
 ↪  in one upload
uint64_t devicePageSize;    // In KB, the size of a page on the
 ↪  GPU. Setting to 0 disables local buffer split in pages
uint64_t localPageSize;     // In KB, the size to split page
 ↪  into if sequence spans multiple devicePageSize pages

// Automatically filled based on device info (still can be
 ↪  reconfigured by user):
uint64_t maxComputeWorkGroupCount[3];   //
 ↪  maxComputeWorkGroupCount from VkPhysicalDeviceLimits
```

```
uint64_t maxComputeWorkGroupSize[3];    //
 ↪  maxComputeWorkGroupCount from VkPhysicalDeviceLimits
uint64_t maxThreadsNum;     // Max number of threads from
 ↪  VkPhysicalDeviceLimits
uint64_t sharedMemorySizeStatic;    // Available for static
 ↪  allocation shared memory size, in bytes
uint64_t sharedMemorySize;  // Available for allocation shared
 ↪  memory size, in bytes
uint64_t sharedMemorySizePow2;  // Power of 2 which is less or
 ↪  equal to sharedMemorySize, in bytes
uint64_t warpSize;  // Number of threads per warp/wavefront.
uint64_t halfThreads;   // Intel fix
uint64_t allocateTempBuffer;    // Buffer allocated by app
 ↪  automatically if needed to reorder Four step algorithm.
 ↪  Parameter to check if it has been allocated
uint64_t reorderFourStep;   // Unshuffle Four step algorithm.
 ↪  Requires tempbuffer allocation (0 - off, 1 - on). Default
 ↪  1.
int64_t maxCodeLength;  // Specify how big can be buffer used
 ↪  for code generation (in char). Default 1000000 chars.
int64_t maxTempLength;  // Specify how big can be buffer used
 ↪  for intermediate string sprintfs be (in char). Default 5000
 ↪  chars. If code segfaults for some reason - try increasing
 ↪  this number.
#if(VKFFT_BACKEND==0) //Vulkan API
VkDeviceMemory tempBufferDeviceMemory;  // Filled at app
 ↪  creation
VkCommandBuffer* commandBuffer;     // Filled at app execution
VkMemoryBarrier* memory_barrier;    // Filled at app creation
#elif(VKFFT_BACKEND==1) //CUDA API
cudaEvent_t* stream_event;  // Filled at app creation
uint64_t streamCounter;     // Filled at app creation
uint64_t streamID;  // Filled at app creation
#elif(VKFFT_BACKEND==2) //HIP API
hipEvent_t* stream_event;   // Filled at app creation
uint64_t streamCounter;     // Filled at app creation
uint64_t streamID;  // Filled at app creation
#elif(VKFFT_BACKEND==3) //OpenCL API
cl_command_queue* commandQueue;     // Filled at app creation
#endif
} VkFFTConfiguration;
```

3.3.1 Driver API parameters

In order to work, VkFFT needs some structures that are provided by the driver. They are backend API-dependent. VkFFT will return corresponding VkFFTResult if one of these structures are not provided (value equal to zero) unless it is stated that there is a default value assigned. VkFFT will not modify provided values directly.

Vulkan API will need the following information:

- VkPhysicalDevice* physicalDevice - Pointer to Vulkan physical device, obtained from vkEnumeratePhysicalDevices()
- VkDevice* device - Pointer to Vulkan device, created with vkCreateDevice()
- VkQueue* queue - Pointer to Vulkan queue, created with vkGetDeviceQueue()
- VkCommandPool* commandPool - Pointer to Vulkan command pool, created with vkCreateCommandPool()
- VkFence* fence - Pointer to Vulkan fence, created with vkCreateFence()
- uint64_t isCompilerInitialized - Specify if glslang compiler has been intialized before (0 - off, 1 - on). Default 0 - VkFFT will call glslang_initialize_process() at initializeVkFFT() and glslang_finialize_process() at deleteVkFFT() calls.

CUDA API will need the following information:

- CUdevice* device - Pointer to CUDA device, obtained from cuDeviceGet()
- cudaStream_t* stream - Pointer to streams (can be more than 1), where to execute the kernels. Default 0. Streams must be associated with the provided device. There is no real benefit in having more than one, however.
- uint64_t num_streams - Try to submit CUDA kernels in multiple streams for asynchronous execution. Default 1

HIP API will need the following information:

- hipDevice_t* device - Pointer to HIP device, obtained from hipDeviceGet()
- hipStream_t* stream - Pointer to streams (can be more than 1), where to execute the kernels. Default 0. Streams must be associated with the provided device. There is no real benefit in having more than one, however.
- uint64_t num_streams - Try to submit HIP kernels in multiple streams for asynchronous execution. Default 1

OpenCL API will need the following information:

- cl_platform_id* platform - Pointer to OpenCL platform, obtained from clGetPlatformIDs()
- cl_device_id* device - Pointer to OpenCL device, obtained from clGetDeviceIDs()
- cl_context* context - Pointer to OpenCL context, obtained from clCreateContext()

3.3.2 Memory management parameters

There are five buffer types user can provide to VkFFT:

- the main buffer (buffer)
- temporary buffer used for calculations requiring out-of-place writes (tempBuffer)
- separate input buffer, from which initial read is performed (inputBuffer)
- separate output buffer, to which final write is performed (outputBuffer)
- kernel buffer, used for calculation of convolutions and cross-correlations (kernel)

These buffers must be passed by a pointer: in Vulkan API they are provided as VkBuffer*, in CUDA and HIP they are provided as void**, in OpenCL, they are provided as cl_mem*. Even though the underlying structure (VkBuffer, void*, cl_mem) is not a memory but just a number that the driver can use to access corresponding allocated memory on the GPU, passing them by a pointer allows for the user to query multiple GPU allocated buffers for VkFFT to use. Currently, it is only supported in Vulkan API - each of five buffer types can be made out of multiple separate memory allocations. For example, it is possible to combine multiple small unused at the point of FFT calculation buffers to form a tempBuffer. This option also allows Vulkan API to overcome the limit of 4GB for a single memory allocation - due to the fact that Vulkan can only use 32-bit numbers for addressing (other APIs support 64-bit addressing).

To use the buffers other than the main buffer, the user has to specify this in configuration at the application creation stage (set to zero by default, optional parameters):

- uint64_t userTempBuffer - enables manual temporary buffer allocation (otherwise it is managed by VkFFT)
- uint64_t isInputFormatted - specifies that initial read is performed from a separate buffer (inputBuffer)
- uint64_t isOutputFormatted - specifies that final write is performed to a separate buffer (outputBuffer)
- uint64_t performConvolution - enables convolution calculations, which requires precomputed kernel (kernel)

Buffer sizes (bufferSize/tempBufferSize/inputBufferSize/outputBufferSize/kernelSize) are provided as a uint64_t pointer to an array, where each element corresponds to the buffer size of the buffer with the same placement in the buffer array. Buffer sizes have to be provided in Vulkan API (due to the stricter memory management model and multiple buffer support) and are optional in other backends (they can be useful to determine when to switch for 64-bit addressing).

Buffer number (bufferNum/tempBufferNum/inputBufferNum/outputBufferNum/kernelNum) corresponds to how many elements are in the buffer and buffer size array. By default it is set to 1 and is not required to be provided by the user. Non-Vulkan backends currently don't support values other than default. Optional parameter.

Buffer offset (bufferOffset/ tempBufferOffset/ inputBufferOffset/ outputBufferOffset/ kernelOffset) specifies offset from the start of the buffer sequence. It must be specified in bytes and must be divisible by the number type size used in the corresponding array (otherwise, the offset will be truncated). It is provided as a single uint64_t value. Optional parameters.

User can provide custom dimension strides for buffer/inputBuffer/outputBuffer buffers - uint64_t[3] array. Strides are specified in elements used in the array (not bytes). The first element corresponds to the stride between elements in the H direction, the second corresponds to the D direction and the third to C (or N, if the number of elements in C is 1). The first axis is assumed to be non-strided. Must be at least of the same size as default strides, otherwise the behavior is undefined. Optional parameters.

uint64_t inverseReturnToInputBuffer - an option that allows setting the final output buffer of the inverse transform to the same buffer, initial read of forward transform is performed from (inputBuffer, if isInputFormatted enabled). Optional parameter.

3.3.3 General FFT parameters

This section describes part of the configuration structure responsible for FFT specification.

uint64_t FFTdim - dimensionality of the transform (1, 2 or 3). Required parameter.

uint64_t size[3] - WHD dimensions of the transform. Required parameter.

uint64_t numberBatches - N parameter of the transform. By default, it is set to 1. Optional parameter.

uint64_t performR2C - perform R2C/C2R decomposition. performDCT must be set to 0. Default 0, set to 1 to enable. Optional parameter.

uint64_t performDCT - perform DCT transformation. performR2C must be set to 0. Default 0, set to X for DCT-X (currently supported X: 2, 3 and 4). OpenCL API has some problems with barrier management in DCT-IV currently and can produce incorrect results. Optional parameter.

uint64_t normalize - enabling this parameter will make the inverse transform divide the result by the FFT length. Default 0, set to 1 to enable. Optional parameter.

3.3.4 Precision parameters (and some things that can affect it):

uint64_t doublePrecision - perform calculations in double precision. Default 0, set to 1 to enable. In Vulkan/OpenCL your device must support double-precision functionality. Optional parameter.

uint64_t doublePrecisionFloatMemory - perform calculations in double precision, but all intermediate and final storage in float. Input/Output/main buffers must have single-precision layout. doublePrecision must be set to 0. This option increases precision, but not that much to be recommended for actual use. Default 0, set to 1 to enable. In Vulkan/OpenCL your device must support double-precision functionality. Experimental feature. Optional parameter.

uint64_t halfPrecision - half-precision in VkFFT is implemented only as memory optimization. All calculations are done in single precision (similar way as doublePrecisionFloatMemory works for double and single precision). Default 0, set to 1 to enable. Works only in Vulkan API now, experimental feature (half precision seems to have bad precision for the first FFT element). Optional parameter.

uint64_t halfPrecisionMemoryOnly - another way of performing half-precision in VkFFT, it will use half-precision only for initial and final memory storage in input/output buffer. Input/Output have to be allocated as half, buffer/tempBuffer have to be allocated as float (out-of-place mode only). Specify isInputFormatted and isOutputFormatted to use. So, for example, intermediate storage between axes FFTs in the multidimensional case will be done in single precision, as opposed to half-precision in the base halfPrecision case. halfPrecision must be set to 1. Default 0, set to 1 to enable. Works only in Vulkan API now, experimental feature. Optional parameter.

uint64_t useLUT - switches from calculating sines and cosines (via special function units in single precision or as a polynomial approximation in double precision) to using precomputed Look-Up Tables. Default 0 in single precision, 1 in double precision, set to 1 to enable. Set to 1 by default for Intel GPUs. If you have issues with single-precision accuracy on your GPU, try enabling this parameter (mobile GPUs may be affected). Optional parameter.

3.3.5 Advanced parameters (code will work fine without using them)

uint64_t omitDimension[3] - parameter, that disables the FFT calculation for a particular axis (WHD). Note, that omitted dimensions still need to be included in FFTdim and size. This parameter simply works as a switch during execution - by not executing the particular dimension code. It doesn't work with the non-strided axis (W) of R2C/C2R mode. It doesn't work with convolution calculations. Default 0, set to 1 to enable. Optional parameter.

uint64_t useUint64 - forces VkFFT to use 64-bit addressing in generated kernels. It is automatically enabled if the estimated buffer size is more than 4GB. Doesn't work with the Vulkan backend. By default, it is set to 0. Optional parameter.

uint64_t coalescedMemory - number of bytes to coalesce per one transaction. For Nvidia and AMD is equal to 32, Intel is equal to 64. Going to work regardless, but if specified by the user correctly, the performance will be higher. Default 64 for other GPUs. For half-precision should be multiplied by two. Should be a power of two. Optional parameter.

uint64_t numSharedBanks - configure the number of shared banks on the target GPU. Default 32. Minor performance boost as it solves shared memory conflicts for the power of two systems. Optional parameter.

uint64_t aimThreads - try to aim all kernels at this amount of threads. Gains/losses are not predictable, just a parameter to play with (it is not guaranteed that the target kernel will use that many threads). Default 128. Optional parameter.

uint64_t useUint64 - forces 64-bit addressing in generated kernels. Should be enabled automatically for systems spanning more than 4GB, but it is better to have an option to force it as a failsafe. Doesn't work in Vulkan API (use multiple buffer binding). Default 0, set to 1 to enable. Optional parameter.

uint64_t fixMaxRadixBluestein - controls the padding of sequences in Bluestein convolution. If specified, the padded sequence will be made of up to fixMaxRadixBluestein primes. Default: 2 for up to 1048576 combined dimension FFT system, 7 after. Min = 2, Max = 13. Optional parameter.

uint64_t disableMergeSequencesR2C - disable the optimization that performs merging of two real sequences to reduce calculations (in R2C/C2R and R2R). If enabled, calculations will be performed by simply setting the imaginary component to zero. Default 0, set to 1 to enable. Optional parameter.

uint64_t disableReorderFourStep - disables unshuffling of the Four Step FFT algorithm (last transposition of data). With this option enabled, tempBuffer will not be needed (unless it is required by Bluestein's multi-upload FFT algorithm). Default 0, set to 1 to enable. Automatically enabled for convolution calculations and Bluestein's algorithm. Optional parameter.

uint64_t makeForwardPlanOnly - generate code only for forward FFT. Default 0, set to 1 to enable. Mutually exclusive with makeInversePlanOnly. Optional parameter.

uint64_t makeInversePlanOnly - generate code only for inverse FFT. Default 0, set to 1 to enable. Mutually exclusive with makeForwardPlan. Optional parameter.

uint64_t considerAllAxesStrided - will create a plan for a non-strided axis similar to a strided axis (used with disableReorderFourStep to get the same layout for Bluestein kernel). Default 0, set to 1 to enable. Optional parameter.

uint64_t keepShaderCode - debugging option, will keep shader code and print all executed shaders during the plan execution in order. Default 0, set to 1 to enable. Optional parameter.

uint64_t printMemoryLayout - debugging option, will print order of buffers used in kernels. Default 0, set to 1 to enable. Optional parameter.

3.3.6 Zero padding parameters

uint64_t performZeropadding[3] - do not read/write some data/perform computations if some part of the sequence is known to have zeros. Set separately for each axis (WHD). If enabled, all 1D sequences in this direction will be considered padded (independent of other zero-padded axes). Default 0, set to 1 to enable. Optional parameter.

uint64_t fft_zeropad_left[3] - specify start boundary of zero block in the system for each axis. Default 0, set to the value between 0 and size[X]-1. Optional parameter.

uint64_t fft_zeropad_right[3] - specify end boundary of zero block in the system for each axis. Default 0, set to the value between fft_zeropad_left[X] and size[X]-1. Optional parameter.

uint64_t frequencyZeroPadding - enables zero padding of the frequency domain, so the first read of inverse FFT will consider the parts of the system from fft_zeropad_left to fft_zeropad_right as zero. Default 0 - spatial zero padding, set to 1 to enable. Optional parameter.

3.3.7 Convolution parameters

uint64_t performConvolution - main parameter that enables convolutions in the application. If enabled, you must specify kernel buffer, number of kernel buffers and kernel sizes (in Vulkan API). Disables reordering of the Four Step FFT algorithm. Default 0, set to 1 to enable. Optional parameter.

uint64_t conjugateConvolution - default 0, set to 1 to enable enables conjugation of the sequence FFT is currently done on, 2 to enable conjugation of the convolution kernel. Optional parameter.

uint64_t crossPowerSpectrumNormalization - normalize the FFT * kernel multiplication in frequency domain. Default 0, set to 1 to enable. Optional parameter.

uint64_t coordinateFeatures - max coordinate (C), or dimension of the features vector. In matrix convolution - the size of the vector. The main purpose is to support Matrix-Vector convolutions. Use numberBatches parameter in tasks, not requiring two separate coordinate-like enumerations of data. Default 1. Optional parameter.

uint64_t matrixConvolution - set to 2 to perform 2x2, set to 3 to perform 3x3 matrix-vector convolution. Matrix-vector convolution is a form of point-wise multiplication in the Fourier space, used by the convolution theorem, where multiplication takes the form of Matrix-vector multiplication. Overrides coordinateFeatures during execution. Default 0. Optional parameter.

uint64_t symmetricKernel - specify if kernel in 2x2 or 3x3 matrix convolution is symmetric. You need to store data as xx, xy, yy (upper-triangular) if enabled and as xx, xy, yx, yy (along rows then along columns, from left to right) if disabled. Default 0, set to 1 to enable. Optional parameter.

uint64_t numberKernels - specify how many kernels were initialized before performing one input/multiple output convolutions. Overwrites numberBatches (N). Only used in convolution step and the following inverse transforms. Default 1. Optional parameter.

uint64_t kernelConvolution - specify if this application is used to create kernel for convolution, so it has the same properties/memory layout. performConvolution has to be set to 0 for the kernel creation. Default 0, set to 1 to enable. Optional parameter, but it is a required parameter for kernel generation.

3.3.8 Register overutilization

Only works in C2C mode, without convolution support. Enabled in Vulkan and OpenCL APIs only (it works in other APIs, but worse). Experimental feature.

uint64_t registerBoost - specify if the register file size is bigger than shared memory and can be used to extend it X times (on Nvidia 256KB register file can be used instead of 32KB of shared memory, set this constant to 4 to emulate 128KB of shared memory). Default 1 - no overutilization. In Vulkan and OpenCL it is set to 4 on Nvidia GPUs, to 2 if the driver shows 64KB or more of shared memory on AMD, to 2 if the driver shows less than 64KB of shared memory on AMD, to 1 if the driver shows 64KB or more of shared memory on Intel, to 2 if the driver shows less than 64KB of shared memory on Intel. Optional parameter.

uint64_t registerBoostNonPow2 - specify if register overutilization should be used on non-power of 2 sequences. Default 0, set to 1 to enable. Optional parameter.

uint64_t registerBoost4Step - specify if register file overutilization should be used in big sequences (>2^14), same definition as registerBoost. Default 1. Optional parameter.

3.3.9 Extra advanced parameters (filled automatically)

uint64_t maxComputeWorkGroupCount[3] - how many workgroups can be launched at one dispatch. Automatically derived from the driver, can be artificially lowered. Then VkFFT will perform a logical split and extension of the number of workgroups to cover the required range.

uint64_t maxComputeWorkGroupSize[3] - max dimensions of the workgroup. Automatically derived from the driver. Can be modified if there are some issues with the driver (as there were with ROCm 4.0, when it returned 1024 for maxComputeWorkGroupSize and actually supported only up to 256 threads).

uint64_t maxThreadsNum - max number of threads per block. Similar to maxComputeWorkGroupSize, but aggregated. Automatically derived from the driver.

uint64_t sharedMemorySizeStatic - available for static allocation shared memory size, in bytes. Automatically derived from the driver. Can be controlled by the user, if desired.

uint64_t sharedMemorySize - available for allocation shared memory size, in bytes. VkFFT uses dynamic shared memory in CUDA/HIP as it allows for bigger allocations. Automatically derived from the driver. Can be controlled by the user, if desired.

uint64_t sharedMemorySizePow2 - the power of 2 which is less or equal to sharedMemorySize, in bytes. Automatically computed.

uint64_t warpSize - number of threads per warp/wavefront. Automatically derived from the driver, but can be modified (can increase performance, though unpredictable as defaults have good values). Must be a power of two.

uint64_t halfThreads - Intel GPU fix, tries to reduce the amount of dispatched threads in half to solve performance degradation in the Four Step FFT algorithm. Default 0 for other

GPUs, try enabling it if performance degrades in the Four Step FFT algorithm for your GPU as well.

int64_t maxCodeLength - specify how big can the buffer used for code generation be (in char). Default 1000000 chars.

int64_t maxTempLength - specify how big can the buffer used for intermediate string sprintf's be (in char). Default 5000 chars. If code segfaults for some reason - try increasing this number.

4 VkFFT Benchmark/Precision Suite and utils_VkFFT helper routines

The only licensed (MIT) part of the VkFFT repository is the VkFFT header file - core library. Other files are either external helper libraries (half, glslang, with their respective licenses) or unlicensed code that is intended for simple copy-pasting (benchmark_scripts, utils_VkFFT.h). It is the easiest way to understand how to use VkFFT by taking the provided scripts and tinker them to the particular task. The current version of the benchmark and precision verification suite has the following codes available:

- user_benchmark_VkFFT - generalization of the main configuration parameters that can be used to launch simplest in-place transforms for the most important supported functionality
- Sample 0 - FFT + iFFT C2C benchmark 1D batched in single precision
- Sample 1 - FFT + iFFT C2C benchmark 1D batched in double precision
- Sample 2 - FFT + iFFT C2C benchmark 1D batched in half precision
- Sample 3 - FFT + iFFT C2C multidimensional benchmark in single precision
- Sample 4 - FFT + iFFT C2C multidimensional benchmark in single precision, native zeropadding
- Sample 5 - FFT + iFFT C2C benchmark 1D batched in single precision, no reshuffling
- Sample 6 - FFT + iFFT R2C / C2R benchmark, in-place.
- Sample 7 - FFT + iFFT C2C Bluestein benchmark in single precision
- Sample 8 - FFT + iFFT C2C Bluestein benchmark in double precision
- Sample 10 - multiple buffers (4 by default) split version of benchmark 0
- Sample 11 - VkFFT / xFFT / FFTW C2C precision test in single precision (xFFT can be cuFFT or rocFFT)
- Sample 12 - VkFFT / xFFT / FFTW C2C precision test in double precision (xFFT can be cuFFT or rocFFT)
- Sample 13 - VkFFT / cuFFT / FFTW C2C precision test in half precision

- Sample 14 - VkFFT / FFTW C2C radix 3 / 5 / 7 / 11 / 13 / Bluestein precision test in single precision
- Sample 15 - VkFFT / xFFT / FFTW R2C+C2R precision test in single precision, out-of-place. (xFFT can be cuFFT or rocFFT)
- Sample 16 - VkFFT / FFTW R2R DCT-II, III and IV precision test in single precision
- Sample 17 - VkFFT / FFTW R2R DCT-II, III and IV precision test in double precision
- Sample 18 - VkFFT / FFTW C2C radix 3 / 5 / 7 / 11 / 13 / Bluestein precision test in double precision
- Sample 50 - convolution example with identity kernel
- Sample 51 - zero padding convolution example with identity kernel
- Sample 52 - batched convolution example with identity kernel
- Sample 100 - VkFFT FFT + iFFT R2R DCT multidimensional benchmark in single precision
- Sample 101 - VkFFT FFT + iFFT R2R DCT multidimensional benchmark in double precision
- Sample 1000 - FFT + iFFT C2C benchmark 1D batched in single precision: all supported systems from 2 to 4096
- Sample 1001 - FFT + iFFT C2C benchmark 1D batched in single precision: all supported systems from 2 to 4096
- Sample 1003 - FFT + iFFT C2C benchmark 1D batched in single precision: all supported systems from 2 to 4096

4.1 utils_VkFFT helper routines

Launching even the simplest Vulkan application can be a non-trivial task. To help with this, utils_VkFFT contains the routines that can help to create the simplest Vulkan application, allocate memory, record command buffers and launch them. Code has some comments explaining what is going on at each step. It also has some useful struct defines (like vkGPU) that keep the most important handles used in Vulkan Compute. This section may be expanded in the future to the proper step-by-step guide on Vulkan Compute simple application creation. I also encourage to check https://github.com/DTolm/VulkanComputeSamples-Transposition repository for another example of a compute algorithm (matrix transposition) implemented with Vulkan API.

utils_VkFFT also has a routine that prints the list of available devices.

vkGPU struct has the following definition:

```
typedef struct {
#if(VKFFT_BACKEND==0) //Vulkan API
```

```
VkInstance instance; //a connection between the application and the Vulkan library
VkPhysicalDevice physicalDevice; //a handle for the graphics card used in the application
VkPhysicalDeviceProperties physicalDeviceProperties; //bastic device properties
VkPhysicalDeviceMemoryProperties physicalDeviceMemoryProperties; //basic memory properties of the device
VkDevice device; //a logical device, interacting with physical device
VkDebugUtilsMessengerEXT debugMessenger; //extension for debugging
uint64_t queueFamilyIndex; //if multiple queues are available, specify the used one
VkQueue queue; //a place, where all operations are submitted
VkCommandPool commandPool; //an opaque objects that command buffer memory is allocated from
VkFence fence; //a vkGPU->fence used to synchronize dispatches
std::vector<const char*> enabledDeviceExtensions;
uint64_t enableValidationLayers;
#elif(VKFFT_BACKEND==1) //CUDA API
CUdevice device;
CUcontext context;
#elif(VKFFT_BACKEND==2) //HIP API
hipDevice_t device;
hipCtx_t context;
#elif(VKFFT_BACKEND==3) //OpenCL API
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue commandQueue;
#endif
uint64_t device_id; //an id of a device, reported by devices_list call
} VkGPU;
```

5 VkFFT Code Examples

This section will provide some simple pseudocode for VkFFT usage, which will once again outline important steps required to launch FFT with VkFFT. More information (and fully working code) can be found in this folder of the VkFFT repository:

/benchmark_samples/vkFFT_scripts/src/

5.1 Driver initializations

Before launching VkFFT, do not forget to do all necessary driver initializations. The following code specifies them for all the supported backends, though the final implementation may be different depending on the particular user's configuration.

```
#if(VKFFT_BACKEND==0) //Vulkan API
VkResult res = VK_SUCCESS;
//create instance - a connection between the application and
 ↪  the Vulkan library
res = createInstance(vkGPU, sample_id);
if (res != 0) {
    //printf("Instance creation failed, error code: %" PRIu64
     ↪  "\n", res);
    return VKFFT_ERROR_FAILED_TO_CREATE_INSTANCE;
}
//set up the debugging messenger
res = setupDebugMessenger(vkGPU);
if (res != 0) {
//printf("Debug messenger creation failed, error code: %"
 ↪  PRIu64 "\n", res);
    return VKFFT_ERROR_FAILED_TO_SETUP_DEBUG_MESSENGER;
}
//check if there are GPUs that support Vulkan and select one
res = findPhysicalDevice(vkGPU);
if (res != 0) {
//printf("Physical device not found, error code: %" PRIu64
 ↪  "\n", res);
    return VKFFT_ERROR_FAILED_TO_FIND_PHYSICAL_DEVICE;
}
//create logical device representation
res = createDevice(vkGPU, sample_id);
if (res != 0) {
//printf("Device creation failed, error code: %" PRIu64 "\n",
 ↪  res);
    return VKFFT_ERROR_FAILED_TO_CREATE_DEVICE;
}
//create fence for synchronization
res = createFence(vkGPU);
if (res != 0) {
//printf("Fence creation failed, error code: %" PRIu64 "\n",
 ↪  res);
    return VKFFT_ERROR_FAILED_TO_CREATE_FENCE;
}
//create a place, command buffer memory is allocated from
```

```
res = createCommandPool(vkGPU);
if (res != 0) {
    //printf("Fence creation failed, error code: %" PRIu64
     ↪  "\n", res);
    return VKFFT_ERROR_FAILED_TO_CREATE_COMMAND_POOL;
}
vkGetPhysicalDeviceProperties(vkGPU->physicalDevice,
 ↪  &vkGPU->physicalDeviceProperties);
vkGetPhysicalDeviceMemoryProperties(vkGPU->physicalDevice,
 ↪  &vkGPU->physicalDeviceMemoryProperties);
glslang_initialize_process();
//compiler can be initialized before VkFFT

#elif(VKFFT_BACKEND==1) //CUDA API
CUresult res = CUDA_SUCCESS;
cudaError_t res2 = cudaSuccess;
res = cuInit(0);
if (res != CUDA_SUCCESS) return
 ↪  VKFFT_ERROR_FAILED_TO_INITIALIZE;
res2 = cudaSetDevice((int)vkGPU->device_id);
if (res2 != cudaSuccess) return
 ↪  VKFFT_ERROR_FAILED_TO_SET_DEVICE_ID;
res = cuDeviceGet(&vkGPU->device, (int)vkGPU->device_id);
if (res != CUDA_SUCCESS) return
 ↪  VKFFT_ERROR_FAILED_TO_GET_DEVICE;
res = cuCtxCreate(&vkGPU->context, 0, (int)vkGPU->device);
if (res != CUDA_SUCCESS) return
 ↪  VKFFT_ERROR_FAILED_TO_CREATE_CONTEXT;
#elif(VKFFT_BACKEND==2) //HIP API
hipError_t res = hipSuccess;
res = hipInit(0);
if (res != hipSuccess) return VKFFT_ERROR_FAILED_TO_INITIALIZE;
res = hipSetDevice((int)vkGPU->device_id);
if (res != hipSuccess) return
 ↪  VKFFT_ERROR_FAILED_TO_SET_DEVICE_ID;
res = hipDeviceGet(&vkGPU->device, (int)vkGPU->device_id);
if (res != hipSuccess) return VKFFT_ERROR_FAILED_TO_GET_DEVICE;
res = hipCtxCreate(&vkGPU->context, 0, (int)vkGPU->device);
if (res != hipSuccess) return
 ↪  VKFFT_ERROR_FAILED_TO_CREATE_CONTEXT;
#elif(VKFFT_BACKEND==3) //OpenCL API
cl_int res = CL_SUCCESS;
cl_uint numPlatforms;
res = clGetPlatformIDs(0, 0, &numPlatforms);
```

```
if (res != CL_SUCCESS) return VKFFT_ERROR_FAILED_TO_INITIALIZE;
cl_platform_id* platforms =
 ↪  (cl_platform_id*)malloc(sizeof(cl_platform_id) *
 ↪  numPlatforms);
if (!platforms) return VKFFT_ERROR_MALLOC_FAILED;
res = clGetPlatformIDs(numPlatforms, platforms, 0);
if (res != CL_SUCCESS) return VKFFT_ERROR_FAILED_TO_INITIALIZE;
uint64_t k = 0;
for (uint64_t j = 0; j < numPlatforms; j++) {
    cl_uint numDevices;
    res = clGetDeviceIDs(platforms[j], CL_DEVICE_TYPE_ALL, 0,
     ↪  0, &numDevices);
    cl_device_id* deviceList =
     ↪  (cl_device_id*)malloc(sizeof(cl_device_id) *
     ↪  numDevices);
    if (!deviceList) return VKFFT_ERROR_MALLOC_FAILED;
    res = clGetDeviceIDs(platforms[j], CL_DEVICE_TYPE_ALL,
     ↪  numDevices, deviceList, 0);
    if (res != CL_SUCCESS) return
     ↪  VKFFT_ERROR_FAILED_TO_GET_DEVICE;
    for (uint64_t i = 0; i < numDevices; i++) {
        if (k == vkGPU->device_id) {
            vkGPU->platform = platforms[j];
            vkGPU->device = deviceList[i];
            vkGPU->context = clCreateContext(NULL, 1,
             ↪  &vkGPU->device, NULL, NULL, &res);
            if (res != CL_SUCCESS) return
             ↪  VKFFT_ERROR_FAILED_TO_CREATE_CONTEXT;
            cl_command_queue commandQueue =
             ↪  clCreateCommandQueue(vkGPU->context,
             ↪  vkGPU->device, 0, &res);
            if (res != CL_SUCCESS) return
             ↪  VKFFT_ERROR_FAILED_TO_CREATE_COMMAND_QUEUE;
            vkGPU->commandQueue = commandQueue;
            k++;
        }
        else {
            k++;
        }
    }
    free(deviceList);
}
free(platforms);
#endif
```

5.2 Simple FFT application example: 1D (one dimensional) C2C (complex to complex) FP32 (single precision) FFT

This example performs the simplest case of FFT. It shows all the necessary fields that the user must fill during the configuration and the submission process. Other samples will build on this one, as driver parameters initialization and code execution commands are the same for all configurations (except for the launch parameters that can be configured after application creation).

```
//zero-initialize configuration + FFT application
VkFFTConfiguration configuration = {};
VkFFTApplication app = {};

configuration.FFTdim = 1; //FFT dimension, 1D, 2D or 3D
configuration.size[0] = Nx; //FFT size
uint64_t bufferSize = (uint64_t)sizeof(float) * 2 *
 ↪  configuration.size[0];

//Device management + code submission
configuration.device = &vkGPU->device;

#if(VKFFT_BACKEND==0) //Vulkan API
configuration.queue = &vkGPU->queue;
configuration.fence = &vkGPU->fence;
configuration.commandPool = &vkGPU->commandPool;
configuration.physicalDevice = &vkGPU->physicalDevice;
 ↪
configuration.isCompilerInitialized = isCompilerInitialized;
 ↪  //glslang compiler can be initialized before VkFFT plan
 ↪  creation. if not, VkFFT will create and destroy one after
 ↪  initialization
#elif(VKFFT_BACKEND==3) //OpenCL API
configuration.platform = &vkGPU->platform;
configuration.context = &vkGPU->context;
#endif

allocateBuffer(buffer, bufferSize); //Pseudocode for buffer
 ↪  allocation, differs between APIs
transferDataFromCPU(buffer, cpu_buffer); //Pseudocode for data
 ↪  transfer from CPU to GPU, differs between APIs

#if(VKFFT_BACKEND==0) //Vulkan API needs bufferSize at
 ↪  initialization
configuration.bufferSize = &bufferSize;
#endif
```

```
VkFFTResult resFFT = initializeVkFFT(&app, configuration);

VkFFTLaunchParams launchParams = {};
launchParams.buffer = &buffer;
#if(VKFFT_BACKEND==0) //Vulkan API
launchParams.commandBuffer = &commandBuffer;
#elif(VKFFT_BACKEND==3) //OpenCL API
launchParams.commandQueue = &commandQueue;
#endif
resFFT = VkFFTAppend(app, -1, &launchParams);

//add synchronization relevant to your API - vkWaitFor-
 ↪ Fences/cudaDeviceSynchronize/hipDeviceSynchronize/clFinish
transferDataToCPU(cpu_buffer, buffer); //Pseudocode for data
 ↪ transfer from GPU to CPU, differs between APIs

freeBuffer(buffer, bufferSize); //Pseudocode for buffer
 ↪ deallocation, differs between APIs

deleteVkFFT(&app);
```

5.3 Advanced FFT application example: ND, C2C/R2C/R2R, different precisions, batched FFT

This example shows how to configure the main parameters of interest in the VkFFT library: multidimensional case, different types of transforms, different precision, perform batched transforms.

In the code below X, Y and Z are the dimensions of FFT, B - number of batches, R2C - real to complex mode 0 or 1 (on/off), DCT - 0, 2, 3 or 4 (off/DCT type), P - precision (0 - single, 1 - double, 2 - half).

```
//zero-initialize configuration + FFT application
VkFFTConfiguration configuration = {};
VkFFTApplication app = {};

configuration.FFTdim = 1; //FFT dimension, 1D, 2D or 3D
configuration.size[0] = X;
configuration.size[1] = Y;
configuration.size[2] = Z;
if (Y > 1) configuration.FFTdim++;
if (Z > 1) configuration.FFTdim++;
configuration.numberBatches = B;
configuration.performR2C = R2C;
```

```
configuration.performDCT = DCT;
if (P == 1) configuration.doublePrecision = 1;
if (P == 2) configuration.halfPrecision = 1;

uint64_t bufferSize = 0;

if (R2C) {
    bufferSize = (uint64_t)(storageComplexSize / 2) *
     ↪  (configuration.size[0] + 2) * configuration.size[1] *
     ↪  configuration.size[2] * configuration.numberBatches;
}
else {
    if (DCT) {
        bufferSize = (uint64_t)(storageComplexSize / 2) *
         ↪  configuration.size[0] * configuration.size[1] *
         ↪  configuration.size[2] *
         ↪  configuration.numberBatches;
    }
    else {
        bufferSize = (uint64_t)storageComplexSize *
         ↪  configuration.size[0] * configuration.size[1] *
         ↪  configuration.size[2] *
         ↪  configuration.numberBatches;
    }
} // storageComplexSize - 4/8/16 for FP16/FP32/FP64
 ↪  respectively.

//Device management + code submission - code is identical to
 ↪  the previous example
```

5.4 Advanced FFT application example: out-of-place R2C FFT with custom strides

In this example, VkFFT is configured to calculate a 3D out-of-place R2C FFT of a system with custom strides. VkFFT reads data from the inputBuffer and produces the result in the buffer.

```
//zero-initialize configuration + FFT application
VkFFTConfiguration configuration = {};
VkFFTApplication app = {};

configuration.FFTdim = 3; //FFT dimension, 1D, 2D or 3D
configuration.size[0] = Nx;
configuration.size[1] = Ny;
```

```
configuration.size[2] = Nz;

configuration.performR2C = 1;

//out-of-place - we need to specify that input buffer is
↪  separate from the main buffer
configuration.isInputFormatted = 1;
configuration.inputBufferStride[0] = configuration.size[0];
configuration.inputBufferStride[1] =
↪  configuration.inputBufferStride[0] *
↪  configuration.size[1];
configuration.inputBufferStride[2] =
↪  configuration.inputBufferStride[1] *
↪  configuration.size[2];

configuration.bufferStride[0] = (uint64_t)
↪  (configuration.size[0] / 2) + 1;
configuration.bufferStride[1] = configuration.bufferStride[0]
↪  * configuration.size[1];
configuration.bufferStride[2] = configuration.bufferStride[1]*
↪  configuration.size[2];

uint64_t inputBufferSize = (uint64_t)sizeof(float) *
↪  configuration.size[0] * configuration.size[1] *
↪  configuration.size[2];

uint64_t bufferSize = (uint64_t)sizeof(float) * 2 *
↪  (configuration.size[0]/2+1) * configuration.size[1] *
↪  configuration.size[2];

//Device management + code submission - code is identical to
↪  the first example, except that you need to allocate two
↪  buffers (and provide them in the launch configuration).
```

5.5 Advanced FFT application example: 3D zero-padded FFT

In this example, VkFFT is configured to calculate a 3D FFT of a system. The meaningful data is located in the first octant of the buffer, the rest is padded with zeros. This configuration removes the circular part of the convolution and allows modelling of open systems.

```
//zero-initialize configuration + FFT application
VkFFTConfiguration configuration = {};
VkFFTApplication app = {};
```

```
configuration.FFTdim = 3; //FFT dimension, 1D, 2D or 3D
configuration.size[0] = Nx;
configuration.size[1] = Ny;
configuration.size[2] = Nz;

configuration.performZeropadding[0] = 1; //Perform padding
 ↪  with zeros on GPU. Still need to properly align input data
 ↪  (no need to fill padding area with meaningful data) but
 ↪  this will increase performance due to the lower amount of
 ↪  the memory reads/writes and omitting sequences only
 ↪  consisting of zeros.
configuration.performZeropadding[1] = 1;
configuration.performZeropadding[2] = 1;
configuration.fft_zeropad_left[0] =
 ↪  (uint64_t)ceil(configuration.size[0] / 2.0);
configuration.fft_zeropad_right[0] = configuration.size[0];
configuration.fft_zeropad_left[1] =
 ↪  (uint64_t)ceil(configuration.size[1] / 2.0);
configuration.fft_zeropad_right[1] = configuration.size[1];
configuration.fft_zeropad_left[2] =
 ↪  (uint64_t)ceil(configuration.size[2] / 2.0);
configuration.fft_zeropad_right[2] = configuration.size[2];

uint64_t bufferSize = (uint64_t)storageComplexSize *
 ↪  configuration.size[0] * configuration.size[1] *
 ↪  configuration.size[2];


//Device management + code submission - code is identical to
 ↪  the first example
```

5.6 Convolution application example: 3x3 matrix-vector convolution in 1D

In this example, VkFFT is configured to calculate a kernel, represented by a 3x3 matrix and a system, represented by a 3D vector. Their convolution is a matrix-vector multiplication in the frequency domain.

```
//zero-initialize configuration + FFT application, we need two
 ↪  - one for kernel calculation
VkFFTConfiguration kernel_configuration = {};
VkFFTConfiguration convolution_configuration = {};
VkFFTApplication app_kernel = {};
VkFFTApplication app_convolution = {};
```

```
kernel_configuration.FFTdim = 1; //FFT dimension, 1D, 2D or 3D
kernel_configuration.size[0] = Nx; //FFT size

uint64_t bufferSize = (uint64_t)sizeof(float) * 2 *
 ↪  kernel_configuration.size[0];

//configure kernel
kernel_configuration.kernelConvolution = 1; //specify if this
 ↪  plan is used to create kernel for convolution
kernel_configuration.coordinateFeatures = 9; //Specify
 ↪  dimensionality of the input feature vector (default 1).
 ↪  Each component is stored not as a vector, but as a separate
 ↪  system and padded on it's own according to other options
 ↪  (i.e. for x*y system of 3-vector, first x*y elements
 ↪  correspond to the first dimension, then goes x*y for the
 ↪  second, etc).
//coordinateFeatures number is an important constant for
 ↪  convolution. If we perform 1x1 convolution, it is equal to
 ↪  number of features, but matrixConvolution should be equal
 ↪  to 1. For matrix convolution, it must be equal to
 ↪  matrixConvolution parameter. If we perform 2x2 convolution,
 ↪  it is equal to 3 for symmetric kernel (stored as xx, xy,
 ↪  yy) and 4 for nonsymmetric (stored as xx, xy, yx, yy).
 ↪  Similarly, 6 (stored as xx, xy, xz, yy, yz, zz) and 9
 ↪  (stored as xx, xy, xz, yx, yy, yz, zx, zy, zz) for 3x3
 ↪  convolutions.
kernel_configuration.normalize = 1;

//Initialize app_kernel and perform a single forward FFT like
 ↪  in examples before. You pass kernel as a buffer for the
 ↪  preparation stage.

convolution_configuration = kernel_configuration;
convolution_configuration.kernelConvolution = 0;
convolution_configuration.performConvolution = 1;
convolution_configuration.symmetricKernel = 0;//Specify if
 ↪  convolution kernel is symmetric. In this case we only pass
 ↪  upper triangle part of it in the form of: (xx, xy, yy) for
 ↪  2d and (xx, xy, xz, yy, yz, zz) for 3d.
convolution_configuration.matrixConvolution = 3;//we do matrix
 ↪  convolution, so kernel is 9 numbers (3x3), but vector
 ↪  dimension is 3
```

```
convolution_configuration.coordinateFeatures = 3;//equal to
 ↪  matrixConvolution size

//Initialize app_convolution and perform a single forward FFT
 ↪  like in examples before. You pass kernel as kernel and
 ↪  system to be convolved with it as buffer
```

5.7 Convolution application example: R2C cross-correlation between two sets of N images

In this example, VkFFT is configured to calculate a kernel, represented by three 2D vectors (RGB values of a pixel) and a system, also represented by three 2D vectors. There are N kernels and N systems. Their cross-correlation is a conjugate convolution in the frequency domain. Images are usually stored as real, not complex numbers, so code uses R2C optimization as well.

```
//zero-initialize configuration + FFT application, we need two
 ↪  - one for kernel calculation
VkFFTConfiguration kernel_configuration = {};
VkFFTConfiguration convolution_configuration = {};
VkFFTApplication app_kernel = {};
VkFFTApplication app_convolution = {};


kernel_configuration.FFTdim = 2; //FFT dimension, 1D, 2D or 3D
kernel_configuration.size[0] = Nx;
kernel_configuration.size[1] = Ny;
kernel_configuration.coordinateFeatures = 3;
kernel_configuration.numberBatches = N;
kernel_configuration.performR2C = 1;
kernel_configuration.normalize = 1;

uint64_t bufferSize = (uint64_t)sizeof(float) * 2 *
 ↪  (kernel_configuration.size[0]/2+1) *
 ↪  kernel_configuration.size[1] *
 ↪  kernel_configuration.coordinateFeatures *
 ↪  kernel_configuration.numberBatches;

kernel_configuration.kernelConvolution = 1; //specify if this
 ↪  plan is used to create kernel for convolution

//Initialize app_kernel and perform a single forward FFT like
 ↪  in examples before. Pad in-place R2C system like this:
```

```
for (uint64_t n = 0; n < kernel_configuration.numberBatches;
 ↪  n++) {
    for (uint64_t c = 0; c <
     ↪  kernel_configuration.coordinateFeatures; c++) {
        for (uint64_t j = 0; j < kernel_configuration.size[1];
         ↪  j++) {
            for (uint64_t i = 0; i <
             ↪  kernel_configuration.size[0]; i++) {
                kernel_padded_GPU[i + j * 2 *
                 ↪  (kernel_configuration.size[0]/2 + 1) + c *
                 ↪  2 * (kernel_configuration.size[0]/2 + 1) *
                 ↪  kernel_configuration.size[1] + n * 2 *
                 ↪  (kernel_configuration.size[0]/2 + 1) *
                 ↪  kernel_configuration.size[1] *
                 ↪  kernel_configuration.coordinateFeatures] =
                 ↪  kernel_input[i + j *
                 ↪  kernel_configuration.size[0] + c *
                 ↪  kernel_configuration.size[0] *
                 ↪  kernel_configuration.size[1] + n *
                 ↪  kernel_configuration.size[0] *
                 ↪  kernel_configuration.size[1] *
                 ↪  kernel_configuration.coordinateFeatures];
            }
        }
    }
}
convolution_configuration = kernel_configuration;
convolution_configuration.kernelConvolution = 0;
convolution_configuration.performConvolution = 1;
convolution_configuration.conjugateConvolution = 1;

//Initialize app_convolution and perform a single forward FFT
 ↪  like in examples before. Pad the system in the same way as
 ↪  the kernel
```