The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLCON 2023

VkFFT and beyond – a platform for runtime GPU code generation

Dmitrii Tolmachev, ETH Zurich

April 18-20, 2023 | University of Cambridge, UK

iwocl.org

Presentation plan

- Release of VkFFT version 1.3 and the new platform for runtime GPU code generation.
- How VkFFT works on the inside and how to optimize GPU code.
- Future of VkFFT.
- Platform showcase: finite difference solver.
- Finite difference solver benchmarks for different accuracy orders and precision on Nvidia A100, AMD MI250 and Nvidia 2080.

Fast Fourier Transform: theory

Discrete Fourier Transform (DFT) is defined as:

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

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

1. Perform bit-reversal permutation to reorder data.
2. Perform N_1 DFTs of size N_2.
3. Perform O(N) multiplications by twiddle factors - complex roots of unity defined by the radix.
4. Perform N_2 DFTs of size N_1.

What is VkFFT?

- 1D/2D/3D systems, forward and inverse directions of FFT.
- Support for big FFT dimension sizes: (2^32, 2^32, 2^32).
- Radix-2/3/4/5/7/8/11/13 FFTs and composite radix kernels.
- Rader’s and Bluestein's FFT algorithms for all other sequences.
- Single, double and half-precision support.
- Complex to complex (C2C), real to complex (R2C), complex to real (C2R) transformations.
- Real to real (R2R) Discrete Cosine Transformations of types I, II, III and IV.
- Convolutions and cross-correlations.
- Native zero padding to model open systems.
- Works on Nvidia, AMD, Intel, Apple and mobile GPUs.
- Works on Windows, Linux and macOS.
- VkFFT supports Vulkan, CUDA, HIP, OpenCL, Level Zero and Metal as backends.

VkFFT version 1.3.0

- Addresses the issues of the single header approach – code is once again fully reorganized (Still in C).
- Removes a big portion of the code duplication.
- Standardizes the code generation – no more hardlined sprintf code paths.
- Implements a new platform for code generation.

VkFFT version 1.3.0: platform structure

Application manager

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

Plan manager

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

Code manager

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

Level 2 kernels

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

Level 1 kernels

Simple routines: matrix-vector multiplication, FFT, pre- and post-processing, read/write blocks.

Level 0 kernels

Memory management, basic math functions inlining, subgroup functions, API-dependent definitions.

VkFFT version 1.3.0: container abstractions

- Code generator now operates on special data containers (implemented as C-unions), that can hold integer/float/complex numbers either as known during the plan creation values or as strings of variable names.
- **Example:** MUL(A,B,C) operation that performs A=B*C.
  1. If all containers have known values, A can be precomputed during plan creation.
  2. If A , B and C are, for example, register names, we print to the kernel an operation of multiplication.
- Each operation like MUL has its own representation for all supported APIs.
- Not a full compiler, but a simple tool that unifies syntaxis of different APIs and supports JIT optimizations. Like what VkFFT is already, but cleaner.

Buffer layout

Registers

Indexes

Read data

Preprocessing

FFT loop

Postprocessing

Write data

OpenCL API

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1))) void VkFFT_main (__global
float2* inputs, __global float2* outputs) {
__local float2 sdata[18];

float2 temp_0;
temp_0.x = 0.0f;
temp_0.y = 0.0f;
float2 temp_1;
temp_1.x = 0.0f;
temp_1.y = 0.0f;
float2 w;
w.x = 0.0f;
w.y = 0.0f;
float2 loc_0;
loc_0.x = 0.0f;
loc_0.y = 0.0f;
unsigned int tempInt;
tempInt = 0;
unsigned int tempInt2;
tempInt2 = 0;
unsigned int shiftX;
shiftX = 0;
unsigned int shiftY;
shiftY = 0;
unsigned int shiftZ;
shiftZ = 0;
unsigned int stageInvocationID;
stageInvocationID = 0;
unsigned int blockInvocationID;
blockInvocationID = 0;
unsigned int sdataID;
sdataID = 0;
unsigned int combinedID;
combinedID = 0;
unsigned int inoutID;
inoutID = 0;
unsigned int inoutID_x;
inoutID_x = 0;
unsigned int inoutID_y;
inoutID_y = 0;
float angle;
angle = 0.0f;
shiftX = get_group_id(0);
shiftZ = 0;
shiftZ = shiftZ + 0;
inoutID = 2 * shiftY;
inoutID = inoutID + shiftZ;
inoutID = inoutID + get_local_id(0);
combinedID = get_local_id(0) + 0;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
temp_0 = inputs[inoutID];
combinedID = get_local_id(0) + 1;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
inoutID = inoutID + 1;
temp_1 = inputs[inoutID];
stageInvocationID = get_local_id(0) + 0;
stageInvocationID = stageInvocationID % 1;
angle = stageInvocationID * -3.14159265358979312e+00f;
w.x = 1.00000000000000000e+00f;
w.y = 0.00000000000000000e+00f;
loc_0.x = temp_1.x * w.x - temp_1.y * w.y;
loc_0.y = temp_1.x * w.y + temp_1.y * w.x;
temp_1.x = temp_0.x - loc_0.x;
temp_1.y = temp_0.y - loc_0.y;
temp_0.x = temp_0.x + loc_0.x;
temp_0.y = temp_0.y + loc_0.y;
inoutID = 2 * shiftY;
inoutID = inoutID + shiftZ;
inoutID = inoutID + get_local_id(0);
combinedID = get_local_id(0) + 0;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
outputs[inoutID] = temp_0;
combinedID = get_local_id(0) + 1;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
inoutID = inoutID + 1;
outputs[inoutID] = temp_1;
}
```

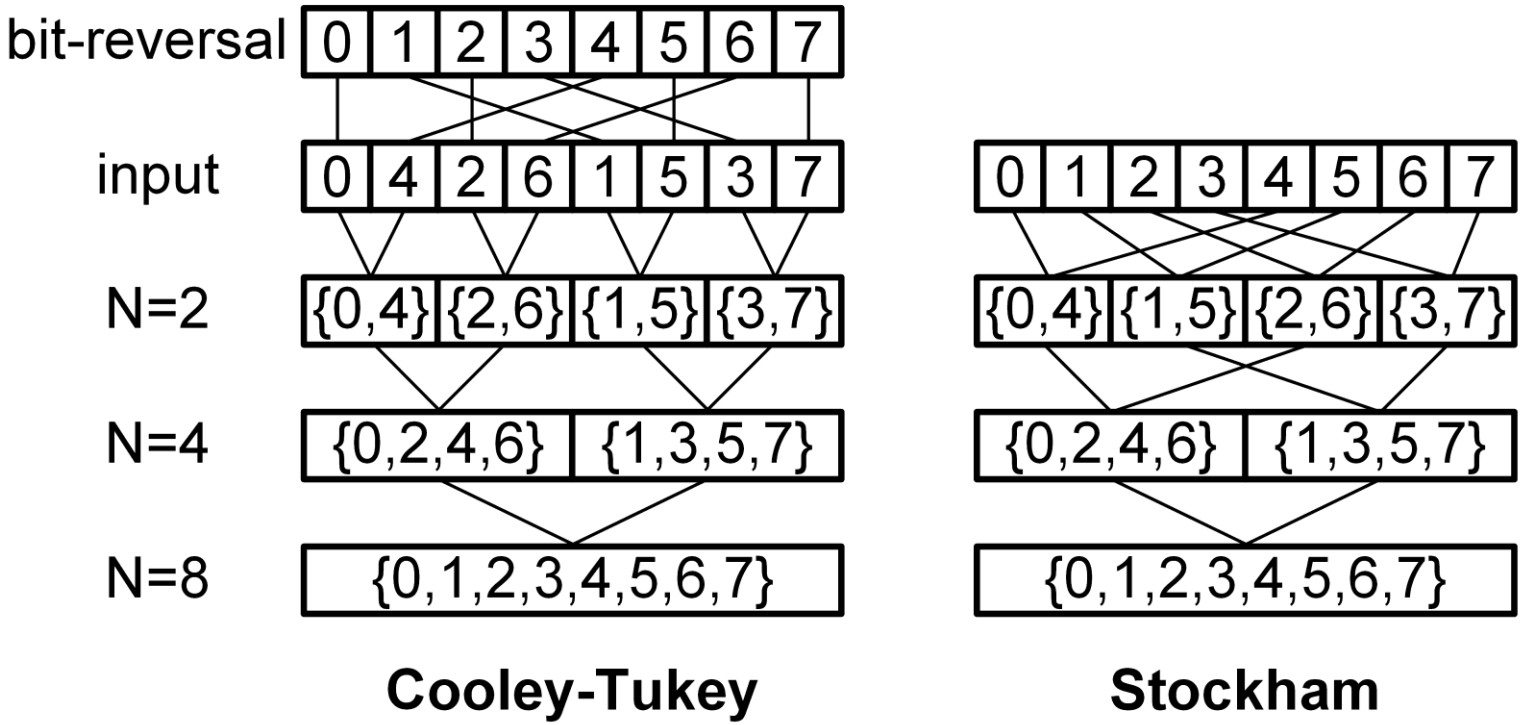
CUDA API

```
extern __shared__ float shared[];
extern "C" __global__ void __launch_bounds__(1) VkFFT_main (float2* inputs, float2* outputs) {
float2* sdata = (float2*)shared;

float2 temp_0;
temp_0.x = 0.0f;
temp_0.y = 0.0f;
float2 temp_1;
temp_1.x = 0.0f;
temp_1.y = 0.0f;
float2 w;
w.x = 0.0f;
w.y = 0.0f;
float2 loc_0;
loc_0.x = 0.0f;
loc_0.y = 0.0f;
unsigned int tempInt;
tempInt = 0;
unsigned int tempInt2;
tempInt2 = 0;
unsigned int shiftX;
shiftX = 0;
unsigned int shiftY;
shiftY = 0;
unsigned int shiftZ;
shiftZ = 0;
unsigned int stageInvocationID;
stageInvocationID = 0;
unsigned int blockInvocationID;
blockInvocationID = 0;
unsigned int sdataID;
sdataID = 0;
unsigned int combinedID;
combinedID = 0;
unsigned int inoutID;
inoutID = 0;
unsigned int inoutID_x;
inoutID_x = 0;
unsigned int inoutID_y;
inoutID_y = 0;
float angle;
angle = 0.0f;
shiftX = blockIdx.x;
shiftZ = 0;
shiftZ = shiftZ + 0;
inoutID = 2 * shiftY;
inoutID = inoutID + shiftZ;
inoutID = inoutID + threadIdx.x;
combinedID = threadIdx.x + 0;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
temp_0 = inputs[inoutID];
combinedID = threadIdx.x + 1;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
inoutID = inoutID + 1;
temp_1 = inputs[inoutID];
stageInvocationID = threadIdx.x + 0;
stageInvocationID = stageInvocationID % 1;
angle = stageInvocationID * -3.14159265358979312e+00f;
w.x = 1.00000000000000000e+00f;
w.y = 0.00000000000000000e+00f;
loc_0.x = temp_1.x * w.x - temp_1.y * w.y;
loc_0.y = temp_1.x * w.y + temp_1.y * w.x;
temp_1.x = temp_0.x - loc_0.x;
temp_1.y = temp_0.y - loc_0.y;
temp_0.x = temp_0.x + loc_0.x;
temp_0.y = temp_0.y + loc_0.y;
inoutID = 2 * shiftY;
inoutID = inoutID + shiftZ;
inoutID = inoutID + threadIdx.x;
combinedID = threadIdx.x + 0;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
outputs[inoutID] = temp_0;
combinedID = threadIdx.x + 1;
inoutID_x = combinedID % 2;
inoutID_y = combinedID / 2;
inoutID_y = inoutID_y + shiftY;
inoutID = inoutID + 1;
outputs[inoutID] = temp_1;
}
```

VkFFT dive-in: main FFT loop

- All algorithms in VkFFT are based on the Stockham version of the Fast Fourier Transform.
- In the Cooley-Tukey algorithm, digit-reversal is performed once - before or after all steps of the recursive algorithm, while in the Stockham algorithm the digit-reversals are done after each small radix (2-13) FT.
- This can be considered as the core of VkFFT and the place where most compute power is used.



VkFFT dive-in: main FFT loop

- VkFFT follows the conception of 1 thread – 1 small radix (2-13) Fourier Transform (FT).
- **Example:** FFT of length 35 = (R5 x 7) * (R7 x 5). This notation means:
  - 7 threads perform radix-5 (R5) FT.
  - 7 threads write to shared memory, synchronize, 5 threads read from shared memory (2 inactive).
  - 5 threads perform radix-7 (R7) FT (2 threads are inactive).
- In this example the kernel configuration will have 7 threads with 7 complex variables (in registers).

VkFFT dive-in: main FFT loop

- VkFFT has 6 primes implemented as a single-thread small FTs - 2, 3, 5, 7, 11 and 13.
  VkFFT also has single-thread FFT codelets for small prime-multiplications: 4, 6, 8, 9, 10, 12, 14, 15, 16 and 32.
- Bigger primes can also be implemented as a single-thread FT, but this reduces total number of threads per kernel and occupancy as a result.

- **Problem:** sizes like 26 = (R2 x 13) * (R13 x 2) will have a huge discrepancy between number of threads and used registers – there is a need for 13 threads with 2 complex registers to do radix-2 FTs, but only 2 threads with 13 registers to do radix-13.
- **Solution:** implement code that can have one thread do more than one small radix FT. Here, an FFT of length 26 can be done with 2 threads, each doing 7 FTs of length 2 (2*7 = 14 complex registers, one extra FT) and 1 FT of length 13. Or it can be done with one thread with 26 registers – there is a room for fine-tuning.

VkFFT dive-in: main FFT loop

We define the following control parameters:

- **min_registers_per_thread** value – the lowest number of registers needed in all small radix kernels.
- **registers_per_thread** value – the max number of registers needed across all small radix kernels.
- Each prime has the number of registers required for it on the range: **[min_registers_per_thread, registers_per_thread]**.
- It is hand-written for all combinations of 2, 3, 5, 7, 11 and 13 (total 64) to have similar low gap between **min_registers_per_thread** and **registers_per_thread** to have max threads working at every step.

- **VkFFTGetRegistersPerThread** function does this in the **vkFFT_Scheduler.h** file.

VkFFT dive-in: main FFT loop

- **Example:** 240 = 2 * 2 * 2 * 2 * 3 * 5.
  - **min_registers_per_thread = 10** – used for radix-5
  - **registers_per_thread = 12** – used for radix-2 and radix-3
  - **Result:** FFT of 240 is split as ([R2 x 6] x 10) * (R12 x 10) * (R10 x 12), as VkFFT has custom kernels for 10 and 12. The stage that performs just radix-2 FT, performs 6 of them in 12 registers.

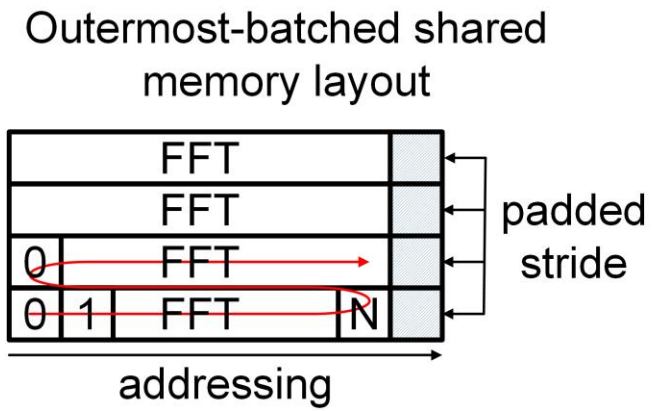
- Total number of threads per FFT is determined as the FFT length divided by **min_registers_per_thread** and rounded up.
  - FFT of length 240 will use 24 threads.

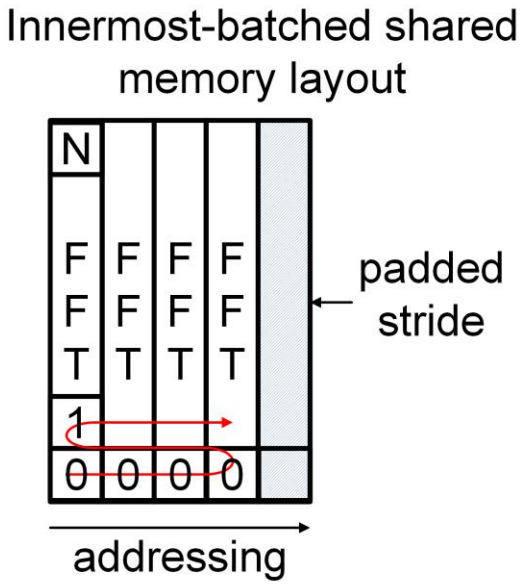
- **Problem:** FFT of length 240 needs 24 threads, which is smaller than the SIMD size of most modern architectures (32 or 64).
- **Solution:** have kernel do 5 FFTs of length 240. This way we have 120 threads and almost full warp utilization.

VkFFT dive-in: main FFT loop

- VkFFT can perform batching in an outermost or in an innermost dimension in the shared memory. This can solve potential shared memory bank conflicts with the common approach of having non power of two strides.

- Overall, thread block structure is 2D in the main FFT loop: threads required per FFT in one dimension and number of FFTs in the other.



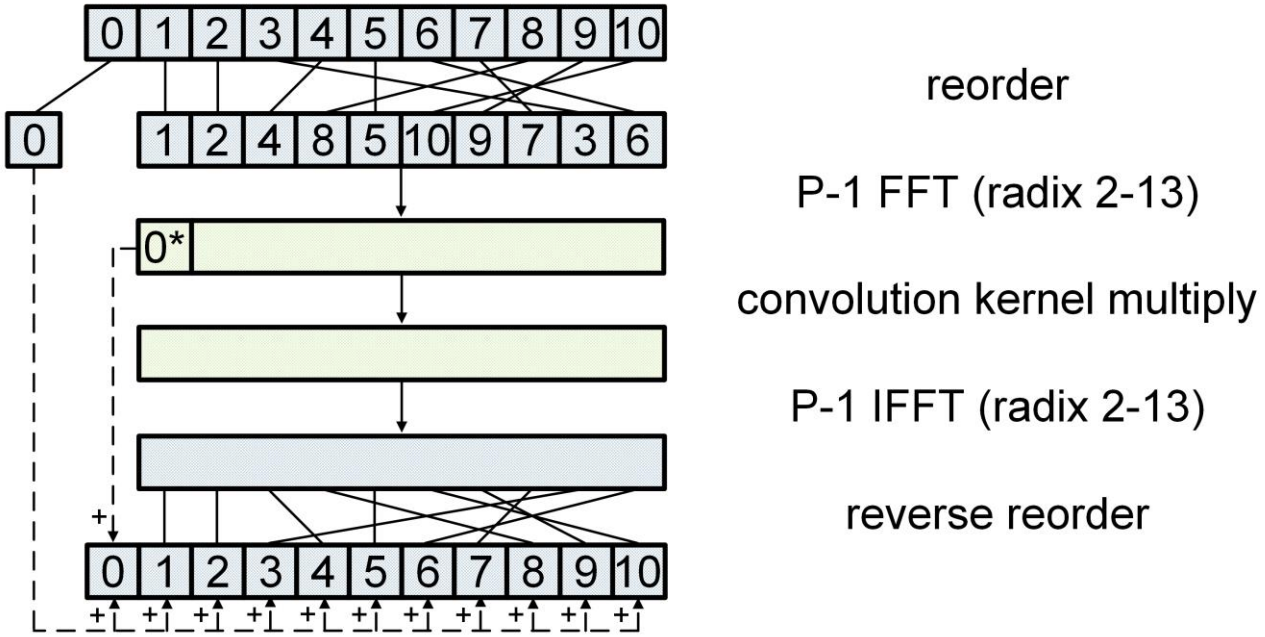


threads_x – FFT
threads_y – batch

threads_x – batch
threads_y – FFT

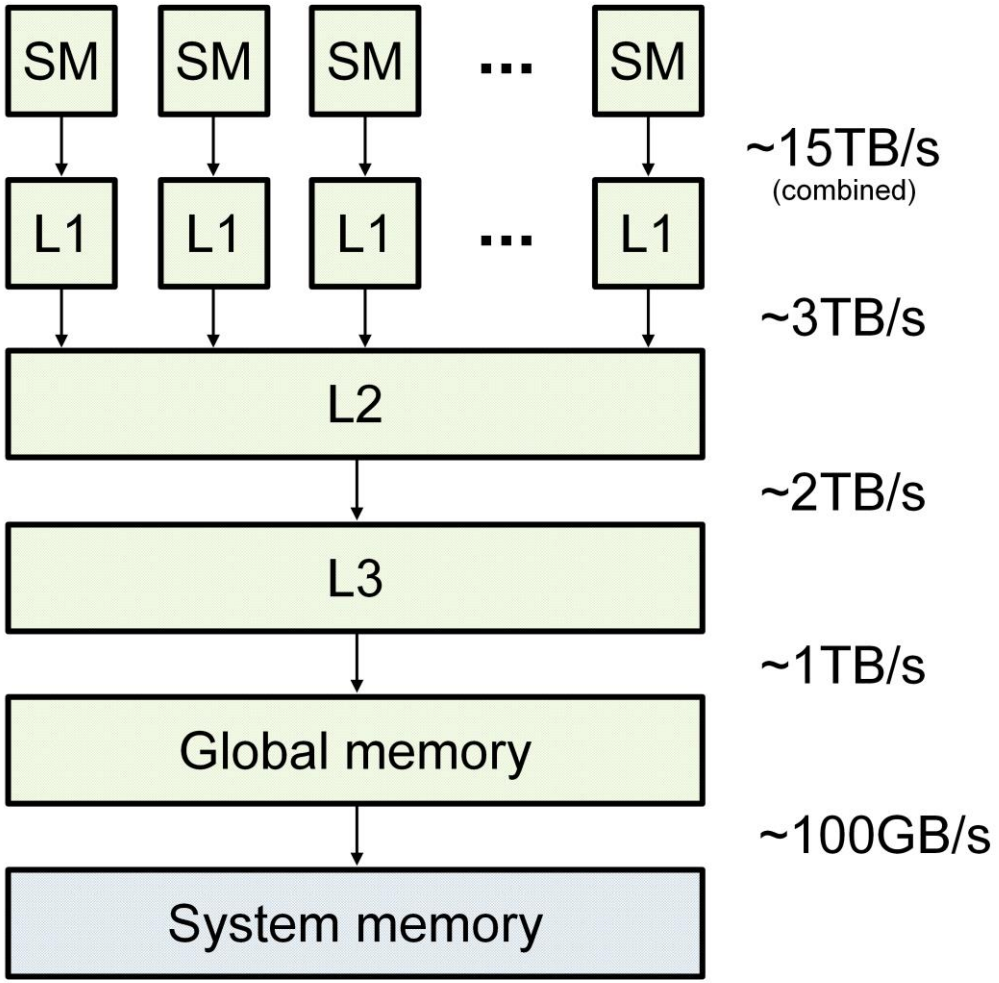
VkFFT dive-in: main FFT loop

- Special mention: primes like 17, 19, etc.
- VkFFT has Rader's FFT algorithm, which allows to view these primes FFTs as P-1 length FFT + IFFT.
- **Example:** 272 = (R17* x 16) * (R16 x 17) = [(R16 x 16) * (R16 x 16)] * (R16 x 17).
- **Example:** 116 = (R29* x 4) * (R4 x 29) = [{(R4 x 28) * (R7 x 16)} * {(R4 x 28) * (R7 x 16)}] * (R4 x 29).



VkFFT dive-in: read/write stage

- Just as important as the main FFT loop stage – FFT is a memory bound problem.

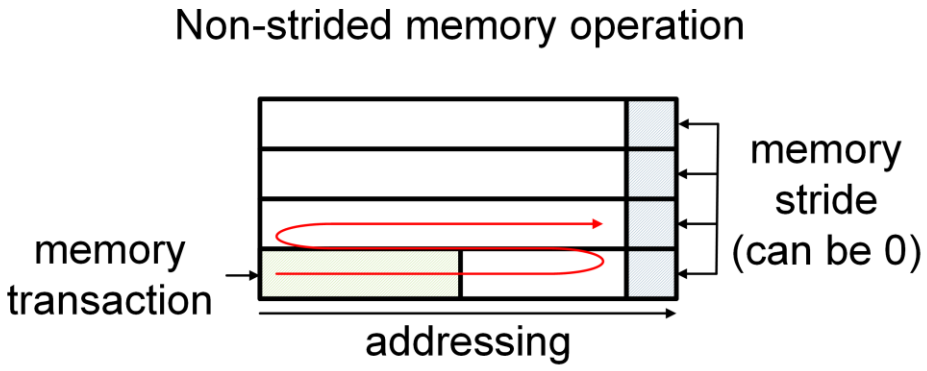


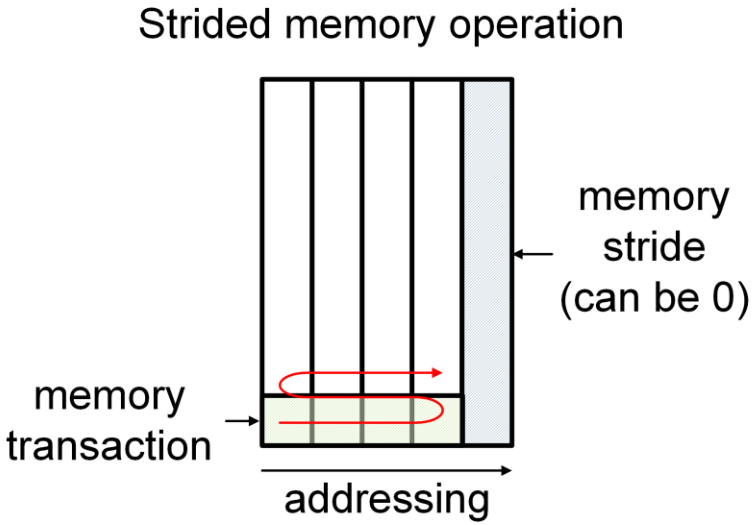
VkFFT dive-in: read/write stage

- VkFFT kernels use all available threads to read/write data as continuously as possible – memory coalescing is the priority. Instead of the 2D approach of the main FFT-loop, they are viewed as a 1D pool:

$$combinedID = threadID.x + threadID.y * blockSize.x$$

- There are two types of read/write operations – non-strided and strided. First works with the continuous data in memory. Second operates on small chunks of data (needed for coalescing) but from distant addresses.





VkFFT dive-in: read/write stage

- All memory operations are abstracted with the help of VkFFT containers, just like math operations.
- If the layout of the memory operations is the same as the first/last FFT operation (each thread reads data that it will use), read/write can be performed directly to registers, otherwise data is written to shared memory.
- Zero-padding is done here as a conditional check.

- In VkFFT 1.3, read/write stage has been generalized and optimized – before there was separate code written for all C2C/R2C/C2R/R2R transforms – total 10k LOC. Now it is only 1.5k LOC.

- **Problem:** extremely distant (>100MB) coalesced memory accesses can hit the same pin of the memory controller and become serialized.
- **Solution:** the only workaround is to coalesce more FFTs per kernel (even potentially increasing memory transferred by having more uploads)

VkFFT dive-in: pre- and post-processing

- In version 1.3, all pre- and post-processing blocks have been rewritten to follow strict rules imposed by the read/write stage.
- They can work either with the continuous 1D adressing of read/write stage or with the 2D approach of the main FFT loop. If the pre/post-processing requires shared memory communication, it can switch from one addressing mode to another, potentially saving on shared memory data transfers.

- If pre/post-processing is not shared memory localized, it can be done as a separate plan with separate dispatch and code generator.
- **Example:** even R2C/C2R decomposition for big sequences.

VkFFT potential future – direct binary generation?

- For FMA operations, the code of VkFFT kernels is already looking similar to Nvidia SASS / AMD ISA, so generating them directly may be a potential way to go in the future.
- VkFFT tries to calculate addresses with as few instructions as possible – but at times it backfires.

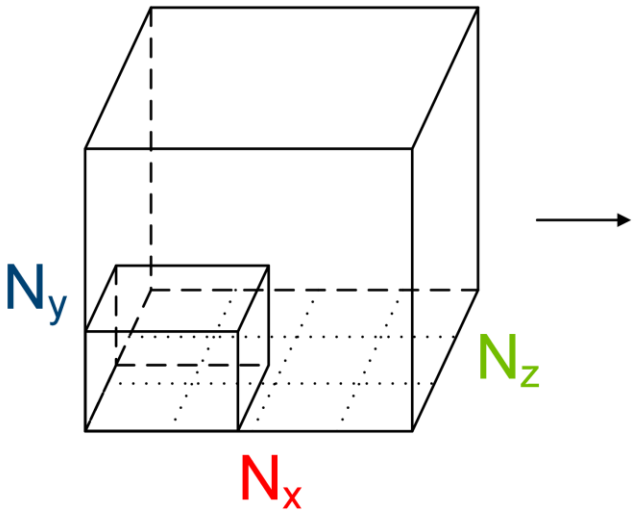
- **Problem:** compiler (NVCC or HIPCC, for example) understands that kernel has the same addresses during the read stage and during the write stage and keeps them for the whole execution time in registers, increasing the number of registers used.

- **Problem:** NVRTC, HIPRTC and other JIT compilers are slow. The sequence of VkFFT sprintf's is fast. With direct binary generation it will be possible to generate optimized binaries with no overhead.

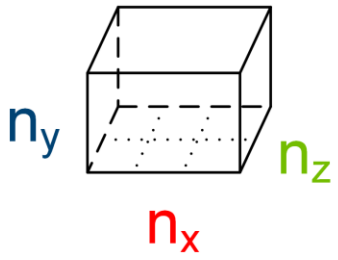
Platform showcase: finite difference solver

- To demonstrate that the platform is able to use the hardware at the lowest level, a finite differences solver that uses no shared memory for communications has been implemented.
- A single working element is called a logic block. It has a singular SIMD size/warp size/wave size group of threads dedicated to it.

Input system



Logic block



n_x = SIMD size multiple

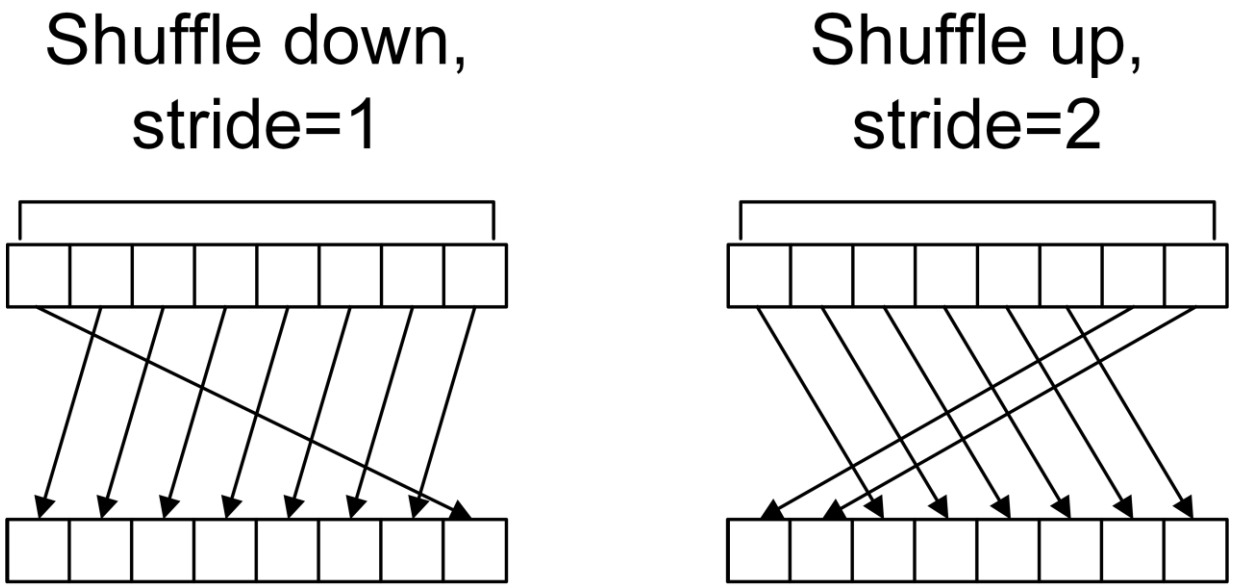
$n_y * n_z * (n_x / \text{SIMD size})$ = registers per thread

$[N_x/n_x]$, $[N_y/n_y]$, $[N_z/n_z]$ = number of logic blocks

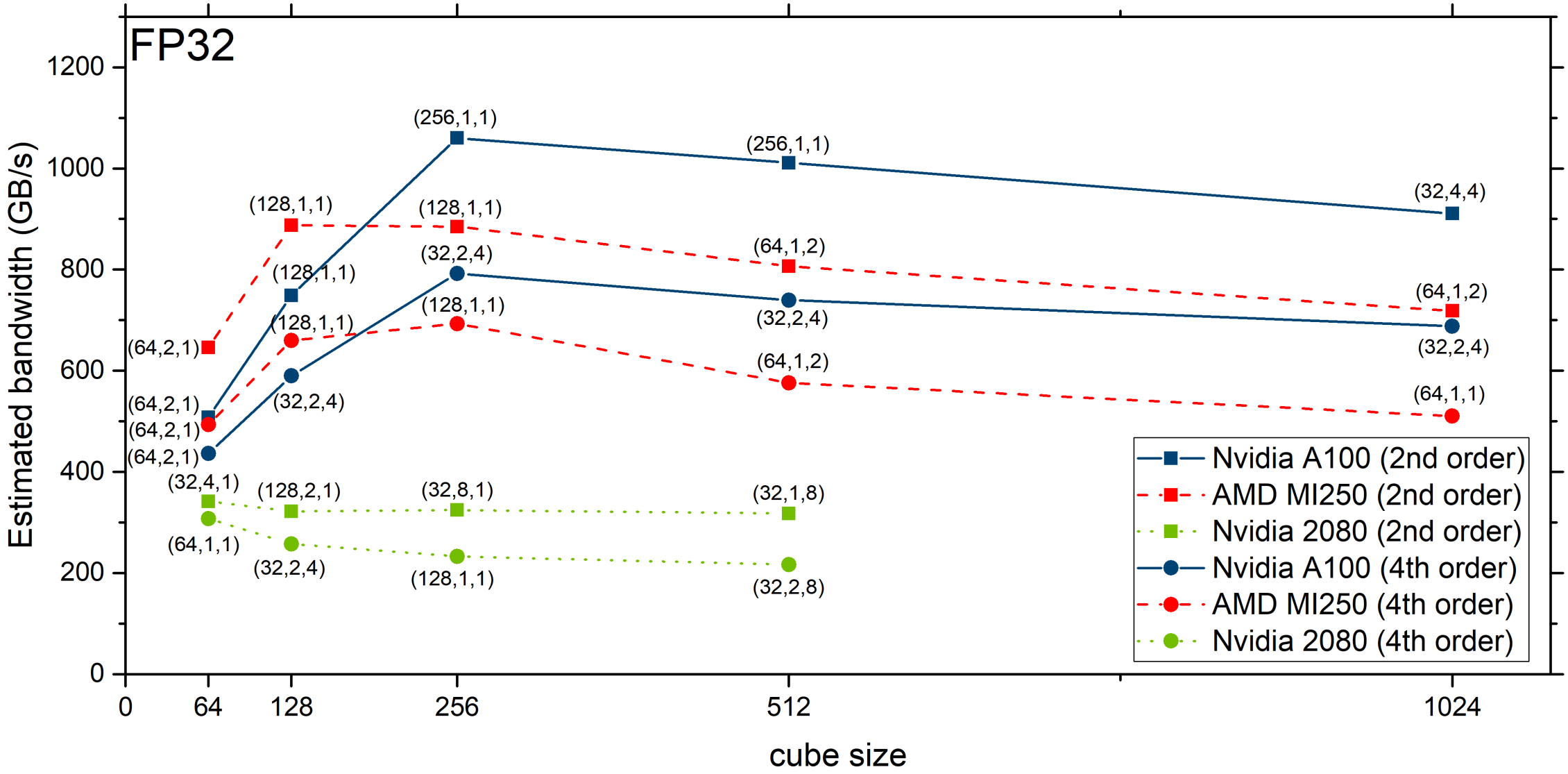
Platform showcase: finite difference solver

- What is unusual: low number of threads per kernel – only SIMD size number of threads.
- We have no synchronizations inside – low number of threads is sufficient to cover latencies.

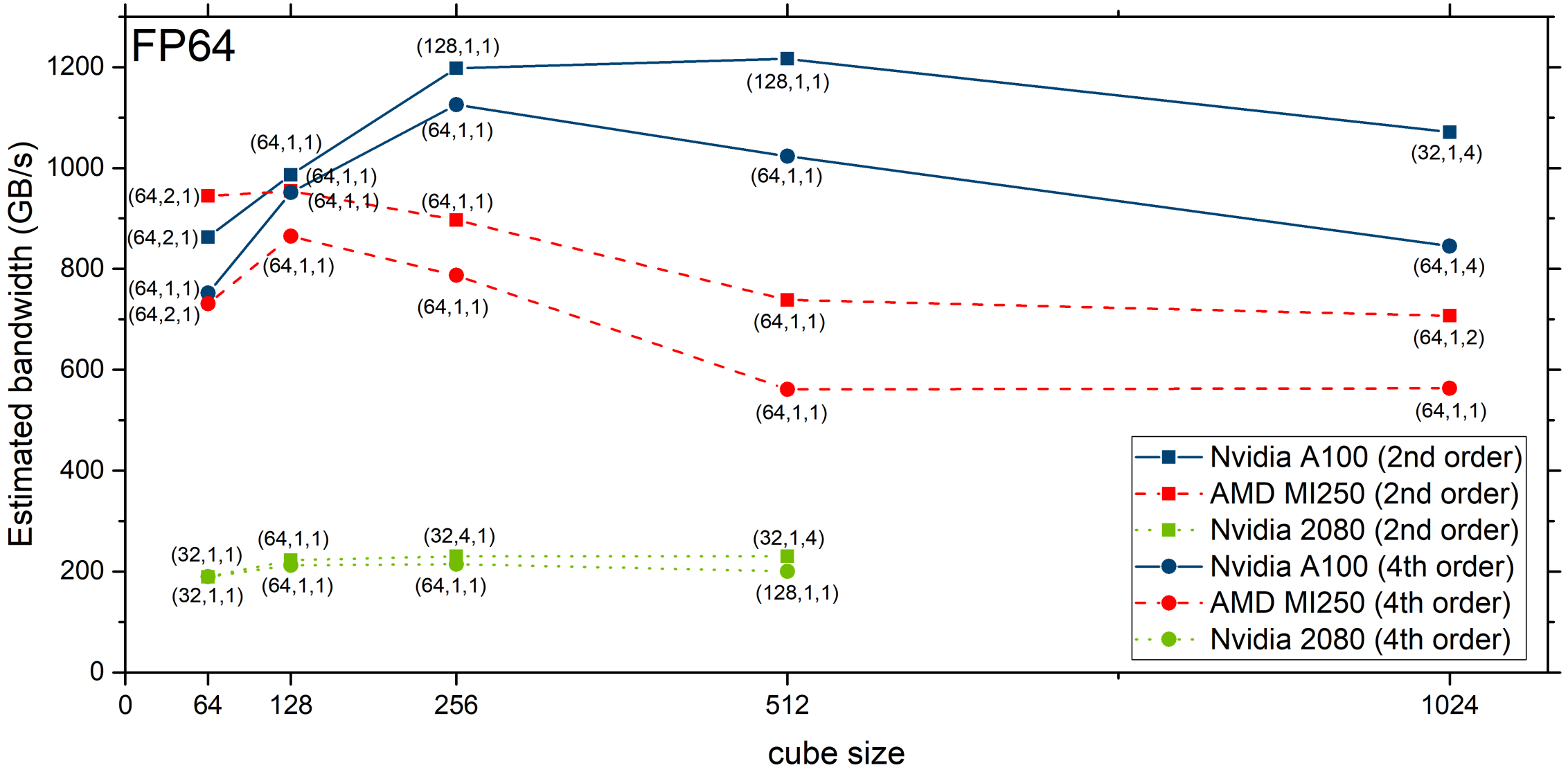
- Configurable size of logic block allows for fine-tuning of halo effects impact on data transfers.
- Within the logic block the only data transfers are subgroup shuffles along the x-axis. All y- and z-axis data resides in registers of a single thread.



Platform showcase: second derivative operation



Platform showcase: finite difference solver



Closing words

- A big redesign of VkFFT has been released on GitHub: https://github.com/DTolm/VkFFT/tree/develop.
- Gave insight into how VkFFT works under the hood.
- The paper on VkFFT has been published: https://ieeexplore.ieee.org/document/10036080.

- Presented the platform of code generation based on the VkFFT design.
- Demonstrated how low-level optimized finite difference solver can be implemented in this platform.

Dmitrii Tolmachev
PhD Student
dtolm96@gmail.com
dmitrii.tolmachev@erdw.ethz.ch

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

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