Latest Advancements on Parallel and Distributed FFT Computation on NVIDIA GPUs

Miguel Ferrer Avila; Josh Romero; Łukasz Ligowski; Filippo Spiga;

SIAM Conference on Parallel Processing for Scientific Computing 2024

Agenda

- The NVIDIA FFT Ecosystem
- cuFFT: Just-In-Time, Link-Time Optimized Kernels
- cuFFTDx: Math Device eXtensions for FFTs
- cuFFTMp: Awesome Scalability
- cuDecomp: Adaptive Pencil Decomposition Library
- NVPL FFT: Beyond the GPU
- Conclusions, Acknowledgements and Contact

The NVIDIA FFT Ecosystem

One Transform, Many Flavors

User C/C++ code

LTO callback

NVPL FFT

cuFFT

LTO EA

cuFFTMp

cuDecomp

cuFFTDx

Grace node

GPU node

GPU node

GPU node

- **NVPL FFT**: Part of the NVIDIA NVPL package
- **cuFFT**: Part of the CUDA Toolkit and NVIDIA HPC SDK
- **cuFFT LTO EA**: Stand-alone preview binary
- **cuFFTMp**: Part of the NVIDIA HPC SDK
- **cuDecomp**: Available as OSS on the NVIDIA Github
- **cuFFTDx**: Part of the Device eXtensions package

Call for feedback!

cuFFT: Just-In-Time, Link-Time Optimized Kernels

Combinatorial explosion

- Why Link-Time Optimization?
- Offline (pre)-compiled kernels:

O { C2C, R2C, C2R X double, float, half… X Cooley-Tukey, Rader, Bluestein… X 32-bit index, 64-bit index X callback, normalization, zero pad… X GPU arch }

- JIT LTO kernels:

O { C2C, R2C, C2R + double, float, half… + Cooley-Tukey, Rader, Bluestein… + 32-bit index, 64-bit index + callback, normalization, zero pad… + GPU arch }

- Users can provide their own LTO pieces:

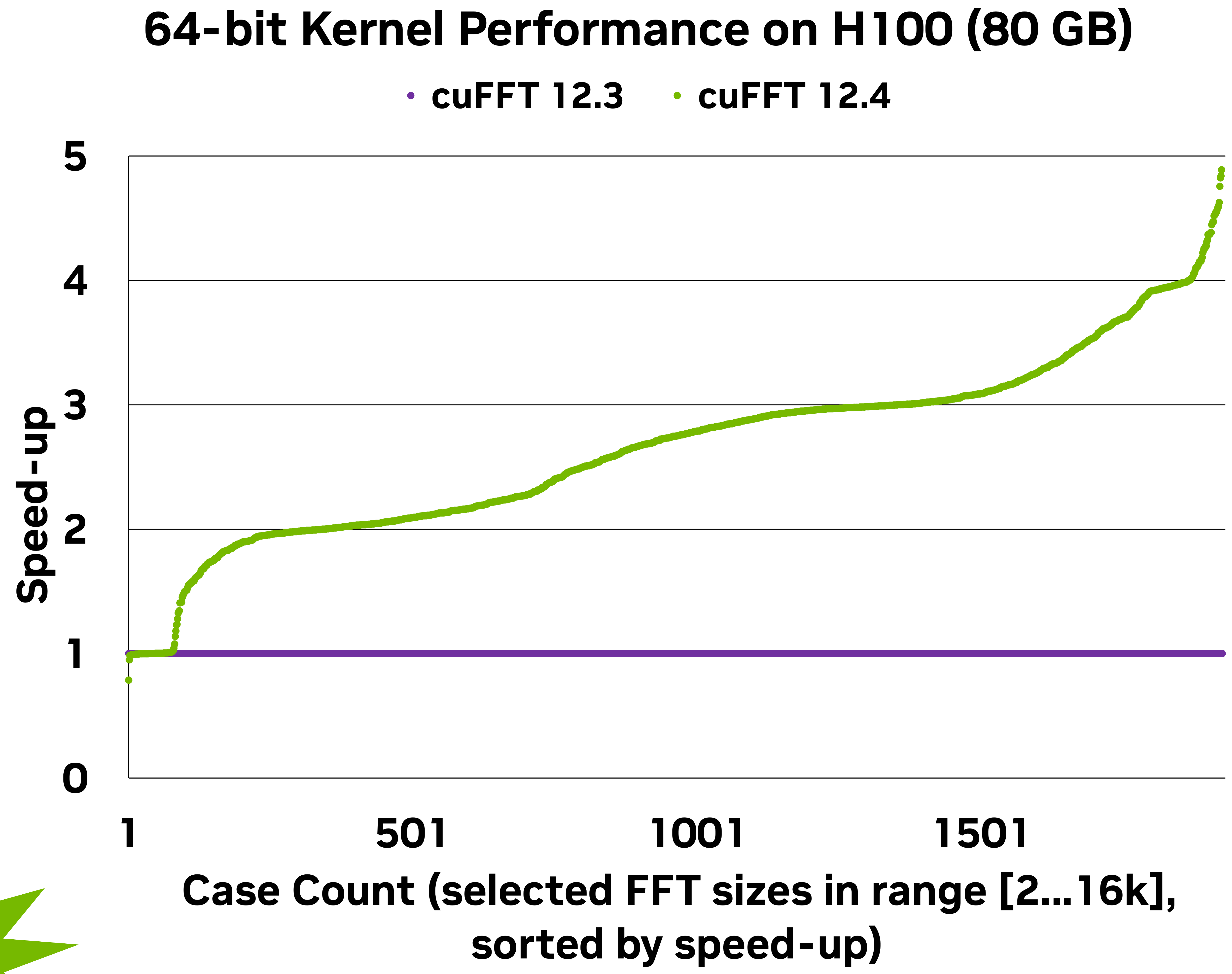
User-defined convolution

cuFFT: Just-In-Time, Link-Time Optimized Kernels

Runtime optimization and fusion with user code

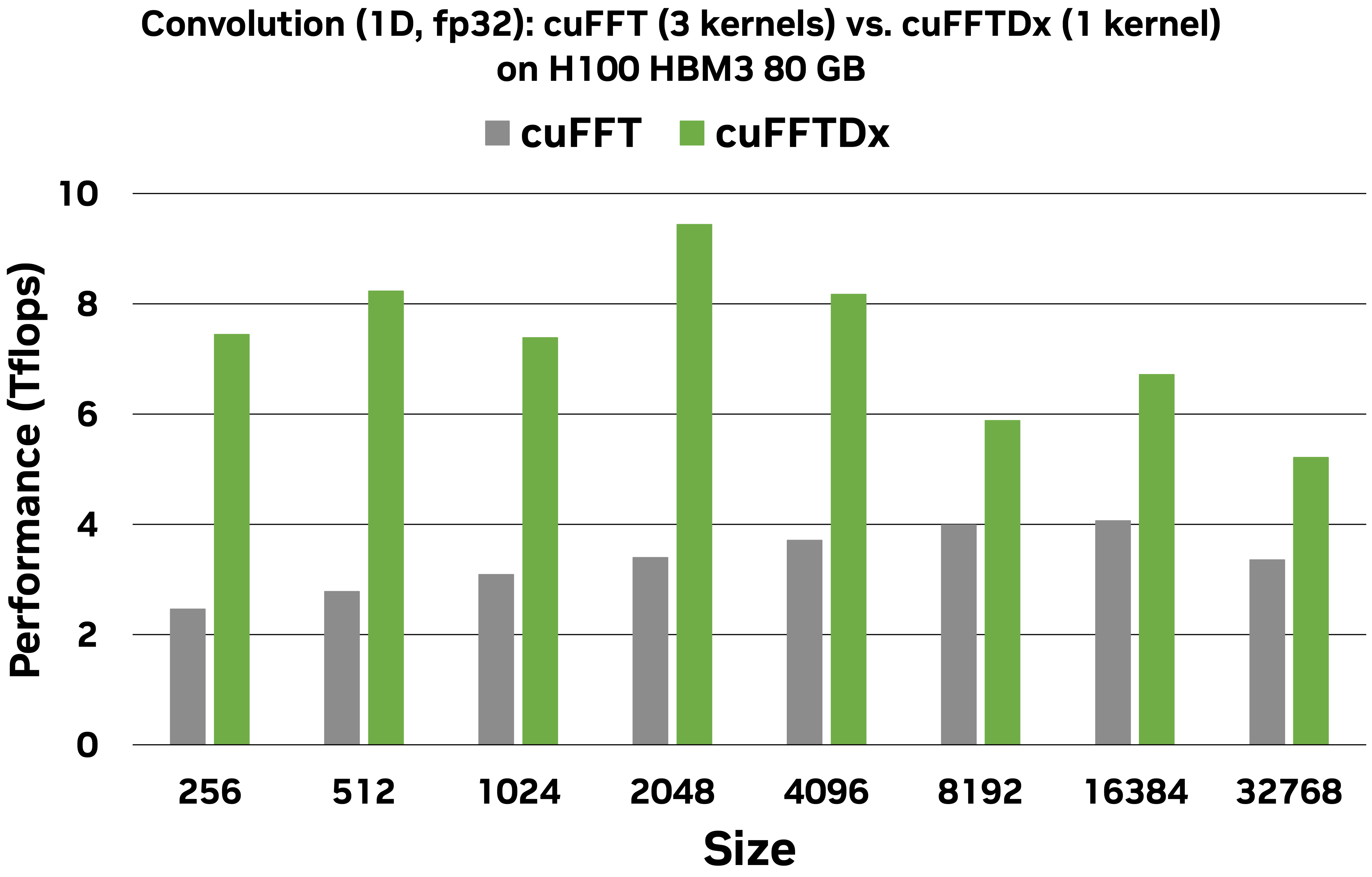
- cuFFT + LTO available in CUDA Toolkit 12.4
  - 64-bit index kernels
- Soon:
  - Improved performance of R2C / C2R, and other kernels
  - Normalization
  - Zero padding
  - Mixed precision (read/write float, compute double) ?
- cuFFT + LTO user callbacks available as preview in cuFFT LTO EA
  - LTO callbacks available on Windows
- Considerations:
  - Opt-in LTO kernels via `cufftSetPlanProperty`
  - User LTO code:
    - Provide function name to cuFFT via API

Call for feedback!



cuFFTDx: Math Device eXtensions for FFTs

Build-Your-Own FFT kernels



- cuFFTDx is a C++, header-only library that enables inlining performant FFT computation in user kernels
- The FFT block can be configured to the user kernel requirements
- Write your own convolution in a single kernel!
  - Example: Hyena convolution operator for LLM
- Best suited for cases where the data fits in shared memory / registers
- Designed to work with other Device Extension libraries

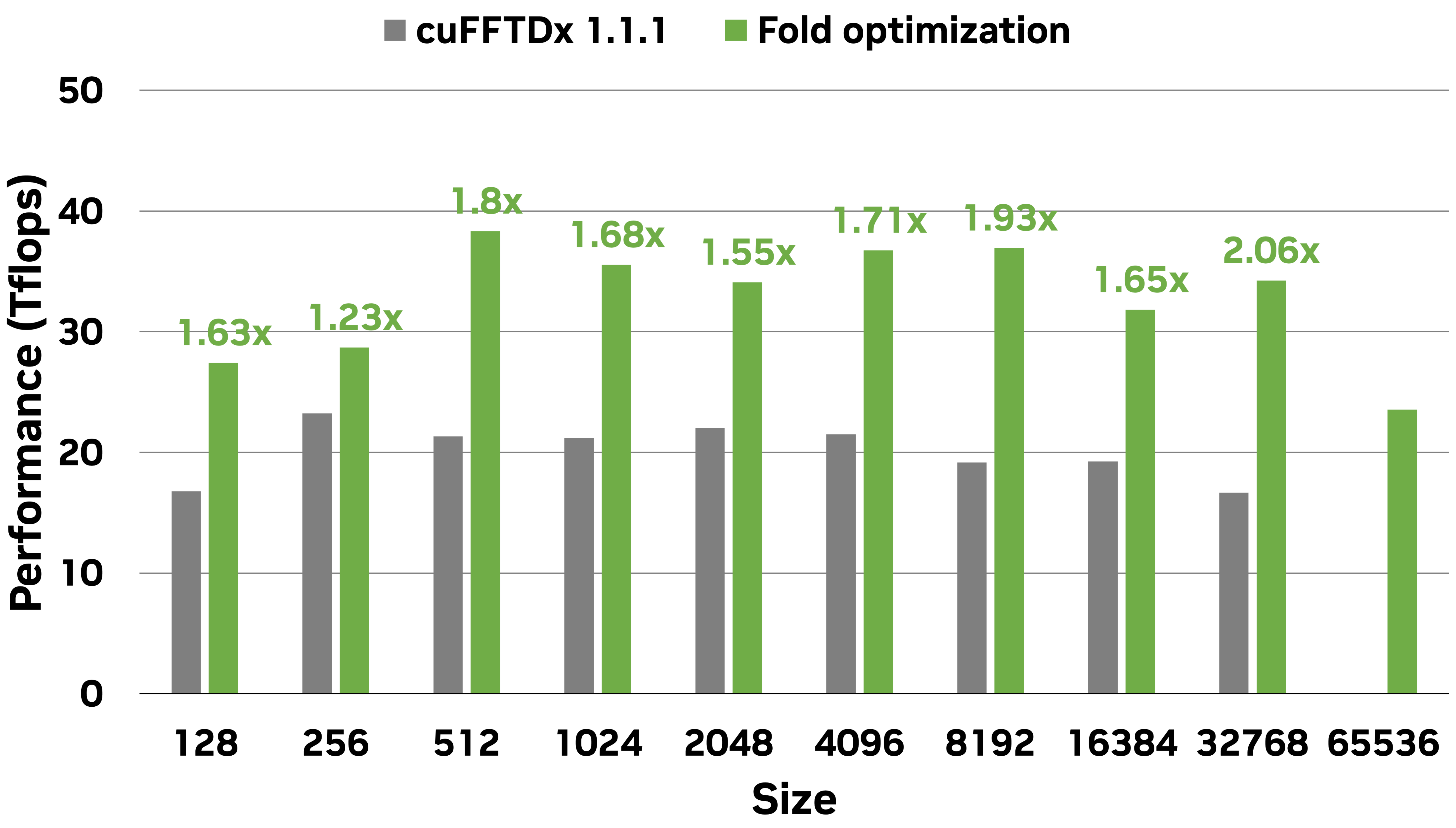
Call for feedback!

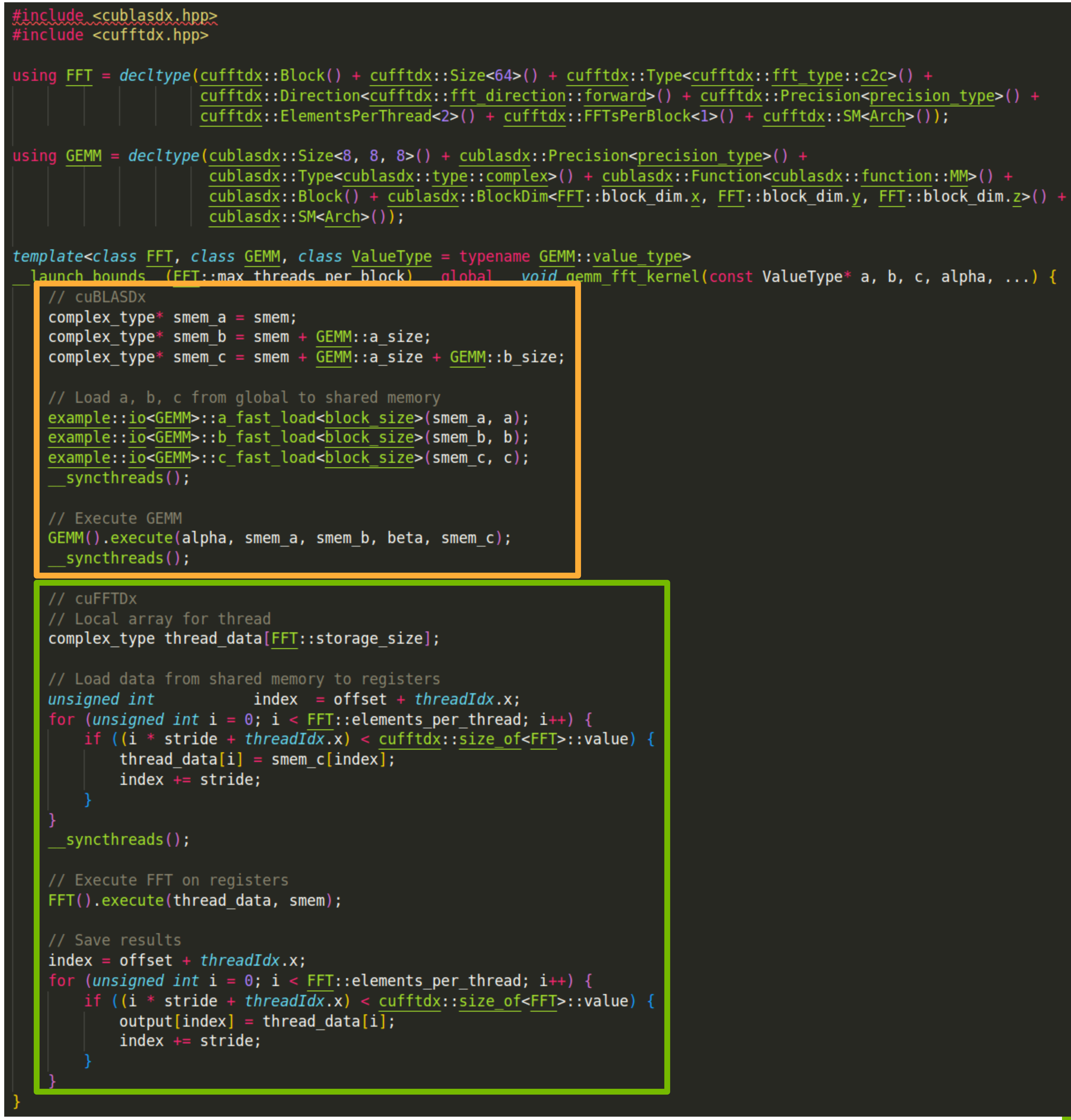
cuFFTDx + cuBLASDx

Build-Your-Own Math kernels

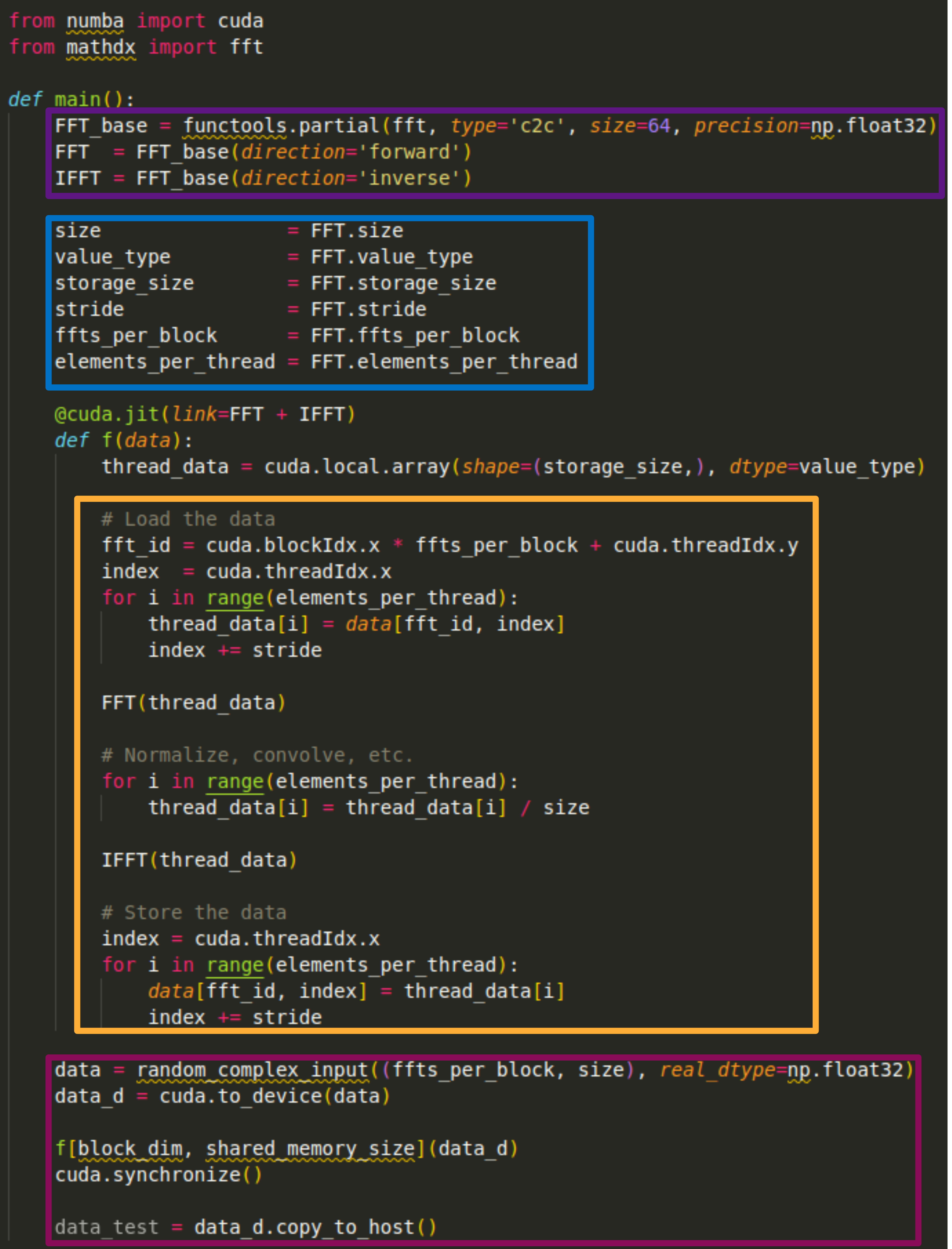
- New features:
  - Interoperability with cuBLASDx (available January 2024)
  - Support for larger R2C / C2R FFT sizes (soon)
    - Up to 64k for single- and half-precision
    - Up to 32k for double-precision
  - Improved performance in R2C / C2R (soon)

cuFFTDx 1D R2C fp32 fold optimization on H100 HBM3 80 GB





```
#include <cublasdx.hpp>
#include <cufftdx.hpp>

using FFT = decltype(cufftdx::Block() + cufftdx::Size<64>() + cufftdx::Type<cufftdx::fft_type::c2c>() +
                     cufftdx::Direction<cufftdx::fft_direction::forward>() + cufftdx::Precision<precision_type>() +
                     cufftdx::ElementsPerThread<2>() + cufftdx::FFTsPerBlock<1>() + cufftdx::SM<Arch>());

using GEMM = decltype(cublasdx::Size<8, 8, 8>() + cublasdx::Precision<precision_type>() +
                      cublasdx::Type<cublasdx::type::complex>() + cublasdx::Function<cublasdx::function::MM>() +
                      cublasdx::Block() + cublasdx::BlockDim<FFT::block_dim.x, FFT::block_dim.y, FFT::block_dim.z>() +
                      cublasdx::SM<Arch>());

template<class FFT, class GEMM, class ValueType = typename GEMM::value_type>
__launch_bounds__(FFT::max_threads_per_block) __global__ void gemm_fft_kernel(const ValueType* a, b, c, alpha, ...) {
    // cuBLASDx
    complex_type* smem_a = smem;
    complex_type* smem_b = smem + GEMM::a_size;
    complex_type* smem_c = smem + GEMM::a_size + GEMM::b_size;

    // Load a, b, c from global to shared memory
    example::io<GEMM>::a_fast_load<block_size>(smem_a, a);
    example::io<GEMM>::b_fast_load<block_size>(smem_b, b);
    example::io<GEMM>::c_fast_load<block_size>(smem_c, c);
    __syncthreads();

    // Execute GEMM
    GEMM().execute(alpha, smem_a, smem_b, beta, smem_c);
    __syncthreads();

    // cuFFTDx
    // Local array for thread
    complex_type thread_data[FFT::storage_size];

    // Load data from shared memory to registers
    unsigned int           index  = offset + threadIdx.x;
    for (unsigned int i = 0; i < FFT::elements_per_thread; i++) {
        if ((i * stride + threadIdx.x) < cufftdx::size_of<FFT>::value) {
            thread_data[i] = smem_c[index];
            index += stride;
        }
    }
    __syncthreads();

    // Execute FFT on registers
    FFT().execute(thread_data, smem);

    // Save results
    index = offset + threadIdx.x;
    for (unsigned int i = 0; i < FFT::elements_per_thread; i++) {
        if ((i * stride + threadIdx.x) < cufftdx::size_of<FFT>::value) {
            output[index] = thread_data[i];
            index += stride;
        }
    }
}
```



```python
from numba import cuda
from mathdx import fft

def main():
    FFT_base = functools.partial(fft, type='c2c', size=64, precision=np.float32)
    FFT  = FFT_base(direction='forward')
    IFFT = FFT_base(direction='inverse')

    size                = FFT.size
    value_type          = FFT.value_type
    storage_size        = FFT.storage_size
    stride              = FFT.stride
    ffts_per_block      = FFT.ffts_per_block
    elements_per_thread = FFT.elements_per_thread

    @cuda.jit(link=FFT + IFFT)
    def f(data):
        thread_data = cuda.local.array(shape=(storage_size,), dtype=value_type)

        # Load the data
        fft_id = cuda.blockIdx.x * ffts_per_block + cuda.threadIdx.y
        index  = cuda.threadIdx.x
        for i in range(elements_per_thread):
            thread_data[i] = data[fft_id, index]
            index += stride

        FFT(thread_data)

        # Normalize, convolve, etc.
        for i in range(elements_per_thread):
            thread_data[i] = thread_data[i] / size

        IFFT(thread_data)

        # Store the data
        index = cuda.threadIdx.x
        for i in range(elements_per_thread):
            data[fft_id, index] = thread_data[i]
            index += stride

    data = random_complex_input((ffts_per_block, size), real_dtype=np.float32)
    data_d = cuda.to_device(data)

    f[block_dim, shared_memory_size](data_d)
    cuda.synchronize()

    data_test = data_d.copy_to_host()
```

cuFFTDx: Math Device eXtensions for FFTs

Beyond C++

- Can we leverage the same functionality on a higher-level?
  - For example, faster kernel prototyping with python

- In this hypothetical example:
  - We create an FFT object from cuFFTDx
  - We query the FFT object properties
  - We write a forward + normalization + inverse kernel, using the FFT object properties
  - We run the kernel

- Can we combine cuFFTDx and a tool like Numba to enable runtime finalized Python kernels?

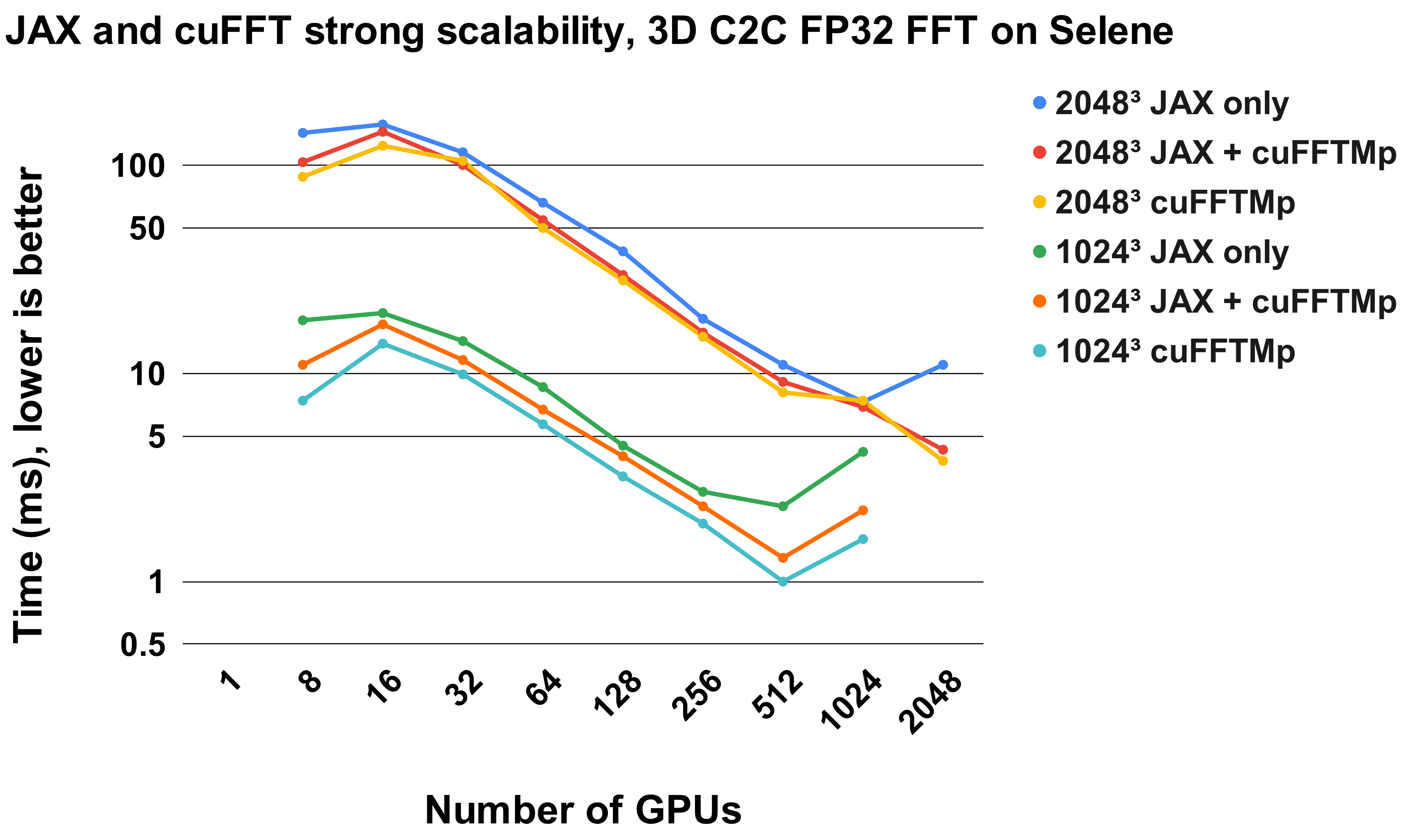
- The resulting kernel should have comparable performance to an equivalent CUDA C++ kernel?

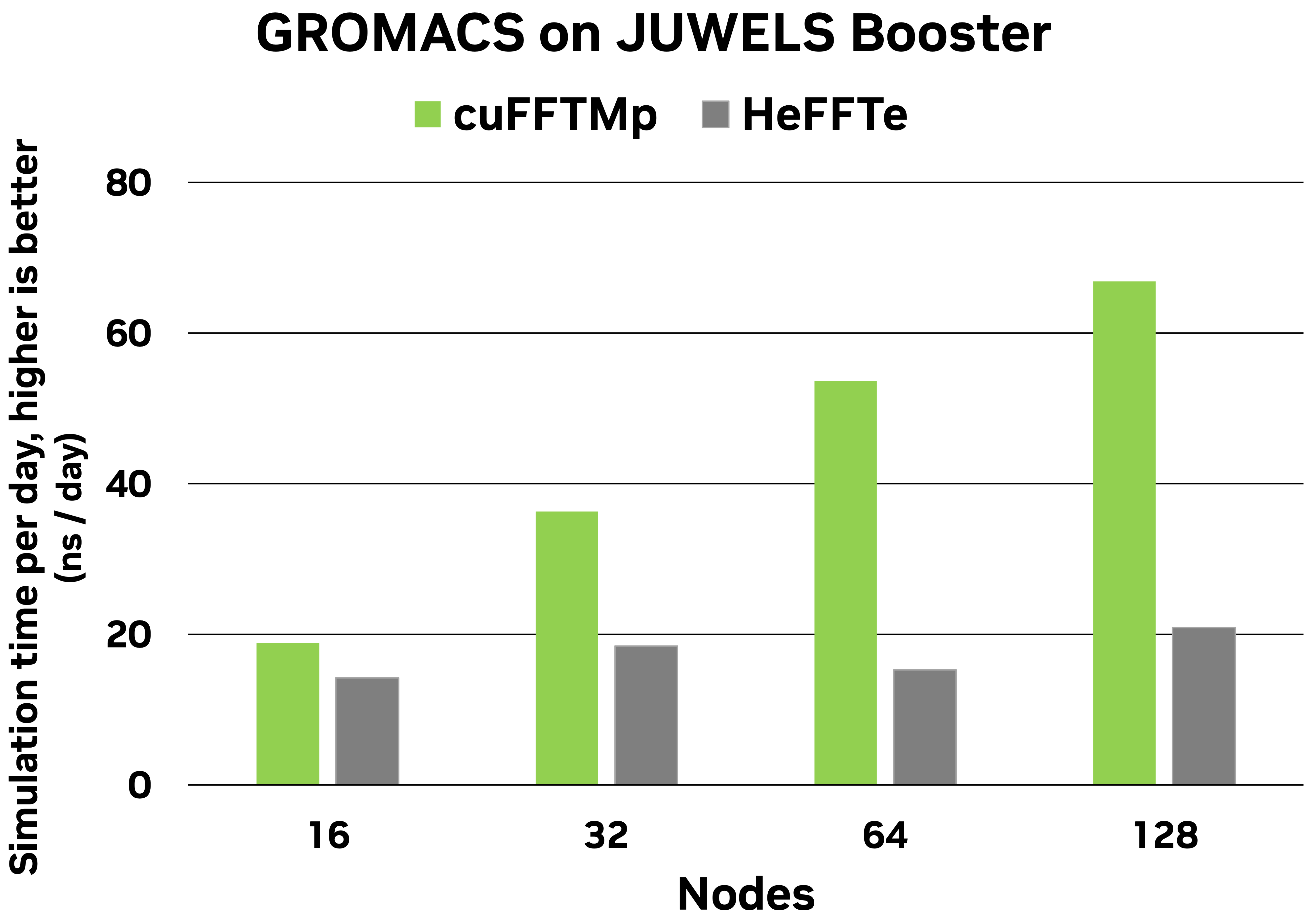
Call for feedback!

cuFFTMp: Awesome Scalability

Distributed FFTs at Speed-Of-Light

- cuFFTMp is a distributed-memory FFT library, currently shipped as EA preview in the NVIDIA HPC SDK
- cuFFTMp supports 2D and 3D FFTs, using slab (1D) and pencil (2D) data decompositions with arbitrary block sizes
- MPI-compatible interface, optimized for single node and multi-node.
- General release later this year with support for minor-version compatibility





cuFFTMp: Awesome Scalability

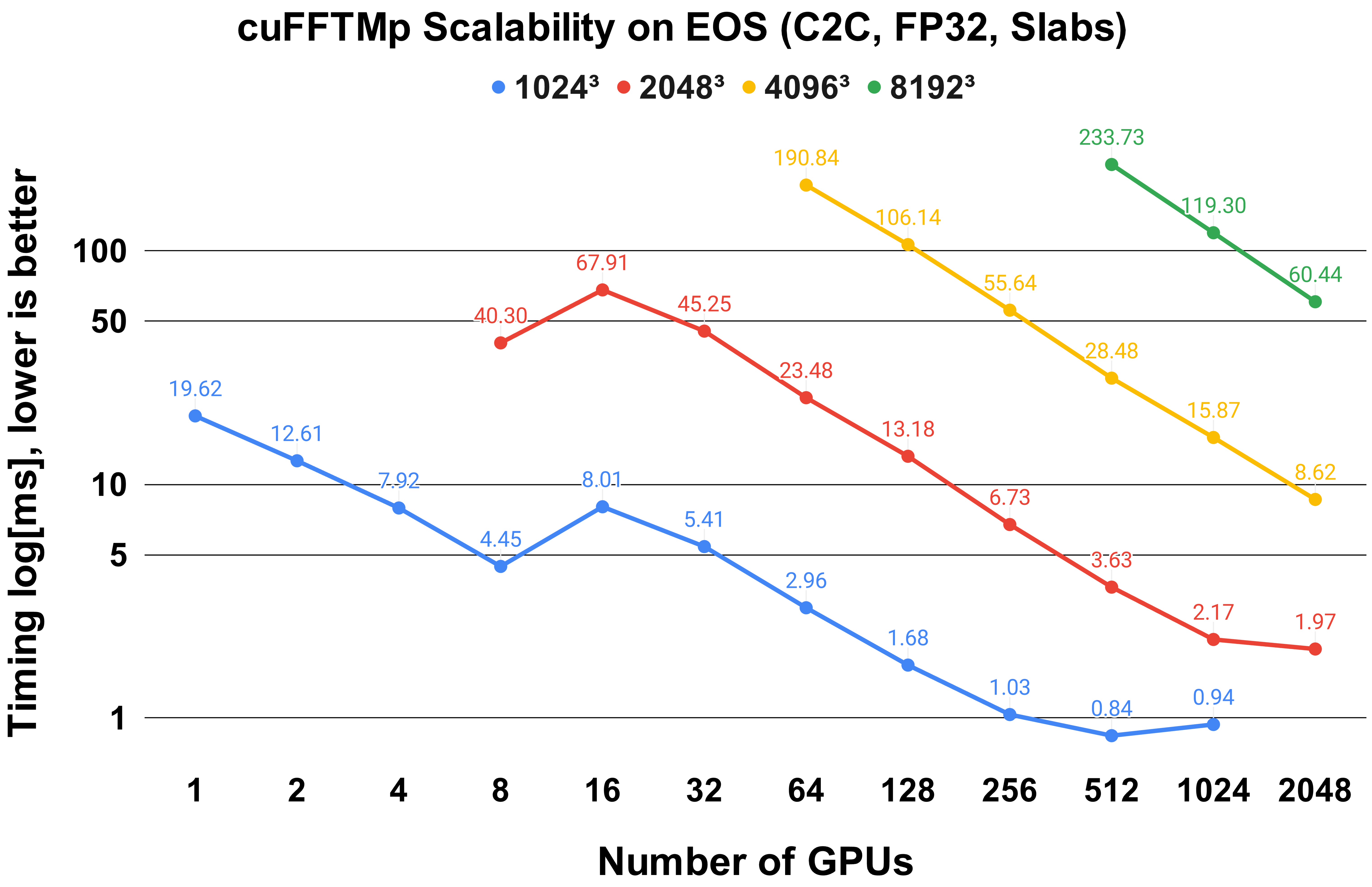
Strong scalability powered by NVSHMEM

- cuFFTMp leverages NVSHMEM to achieve strong scalability and low-latency

- NVSHMEM: PGAS library for NVIDIA clusters
  - Based on OpenSHMEM
  - Asynchronous, low-overhead comms initiated by CUDA threads
  - MPI, OpenSHMEM interoperability

- Should we take it one step further?
  - NVSHMEM + cuFFTDx kernels…
  - … managed and run by a python application?

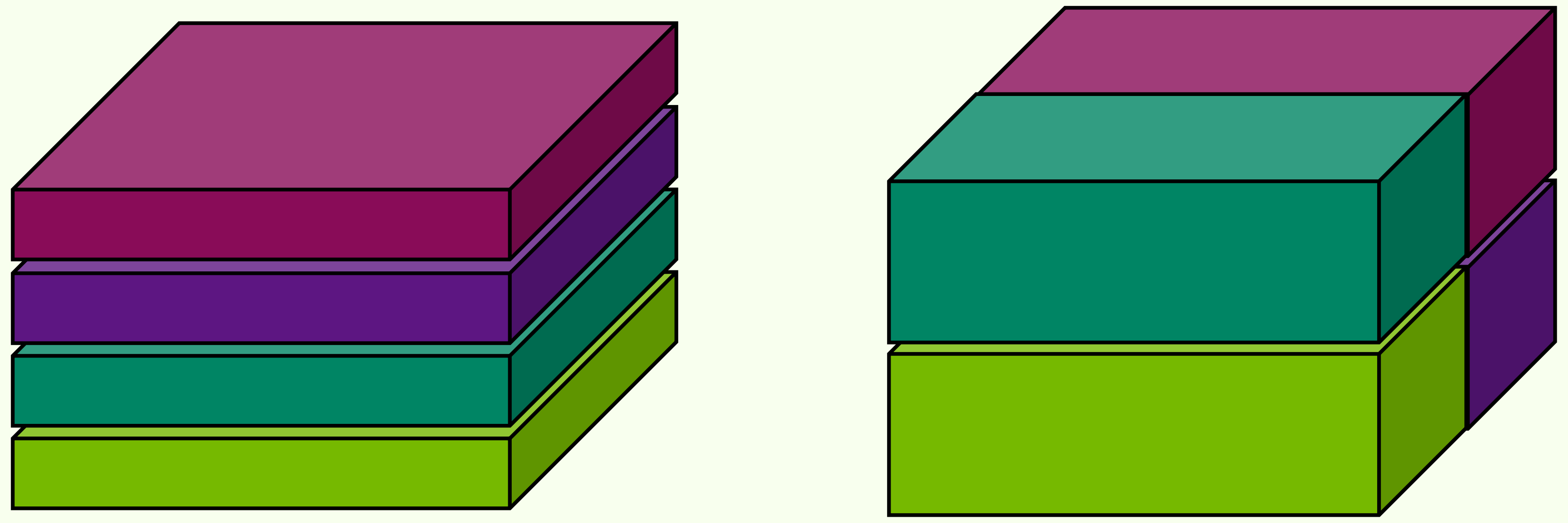
Call for feedback!



cuDecomp: Adaptive Pencil Decomposition Library

Build your own performant distributed FFTs!

- cuDecomp: slab / pencil decomposition for global transposition beyond FFTs (e.g. tridiagonal solvers)
  - Inspired by 2DECOMP&FFT library, but GPU-centric with C/C++ and Fortran support
  - Runtime auto-tuning of data distribution schema and communication backend (MPI, NCCL, NVSHMEM)



Selects best 2D process distribution

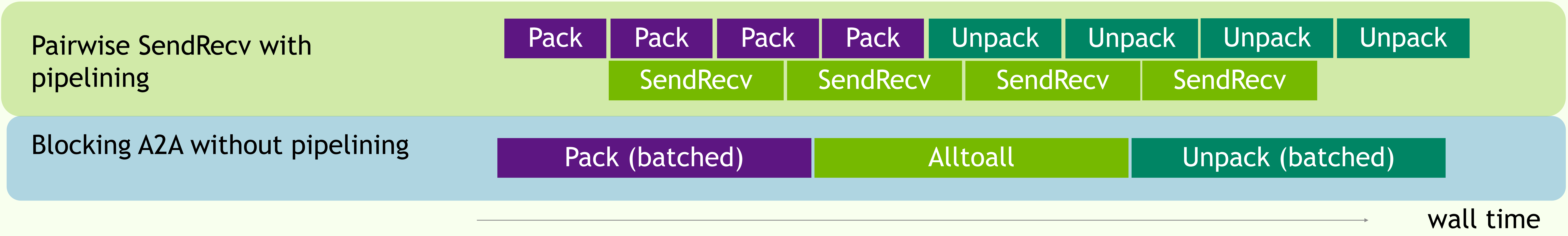
NCCL

NVSHMEM

MPI

OpenMPI
Spectrum MPI
Cray MPICH

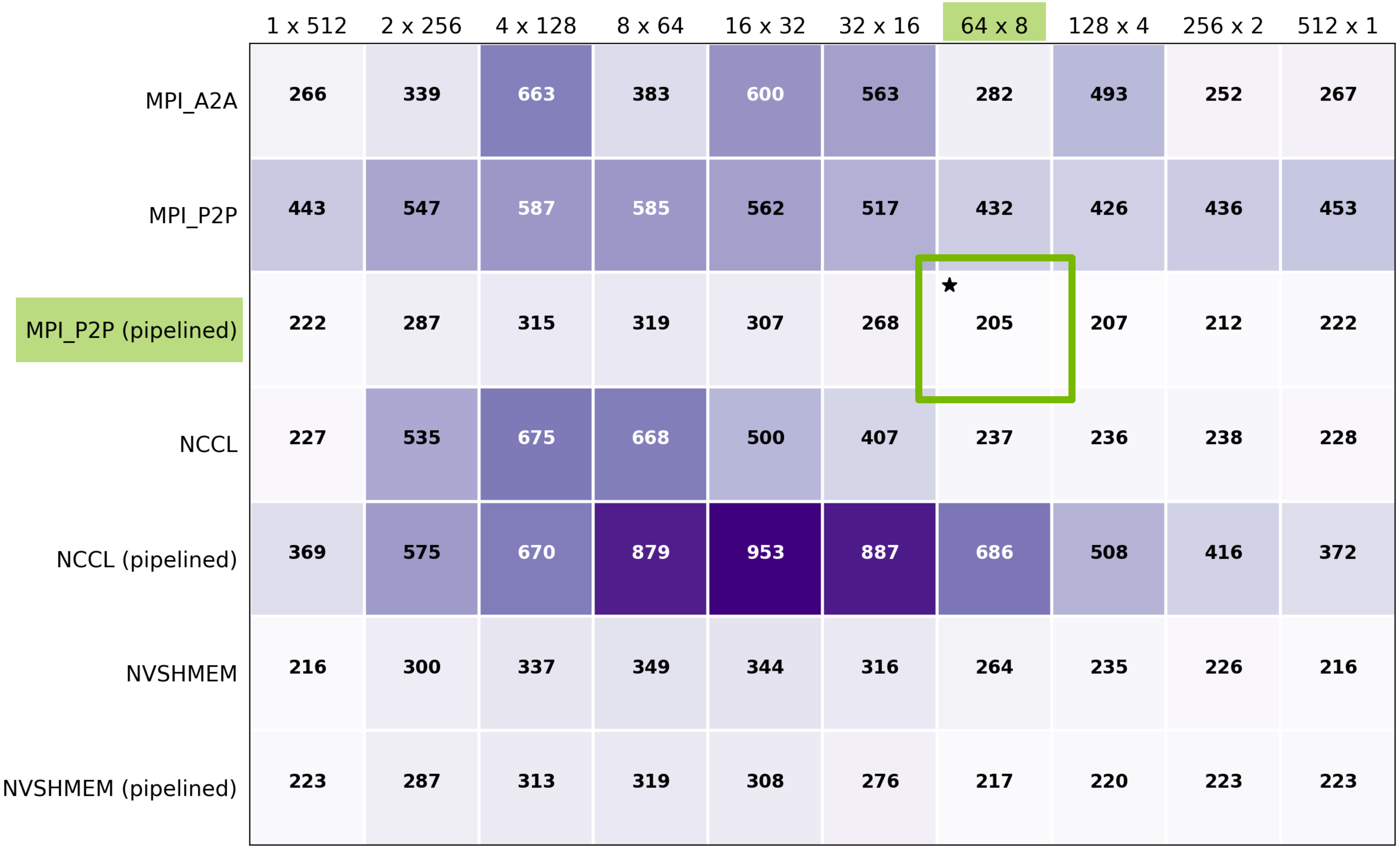
Selects best communication library



Multiple high-level implementations available

cuDecomp: Autotuning Sample Results

4096 x 8192 x 8192 Grid, Single Complex Precision



	1 x 512	2 x 256	4 x 128	8 x 64	16 x 32	32 x 16	64 x 8	128 x 4	256 x 2	512 x 1
MPI_A2A	266	339	663	383	600	563	282	493	252	267
MPI_P2P	443	547	587	585	562	517	432	426	436	453
MPI_P2P (pipelined)	222	287	315	319	307	268	★ 205	207	212	222
NCCL	227	535	675	668	500	407	237	236	238	228
NCCL (pipelined)	369	575	670	879	953	887	686	508	416	372
NVSHMEM	216	300	337	349	344	316	264	235	226	216
NVSHMEM (pipelined)	223	287	313	319	308	276	217	220	223	223

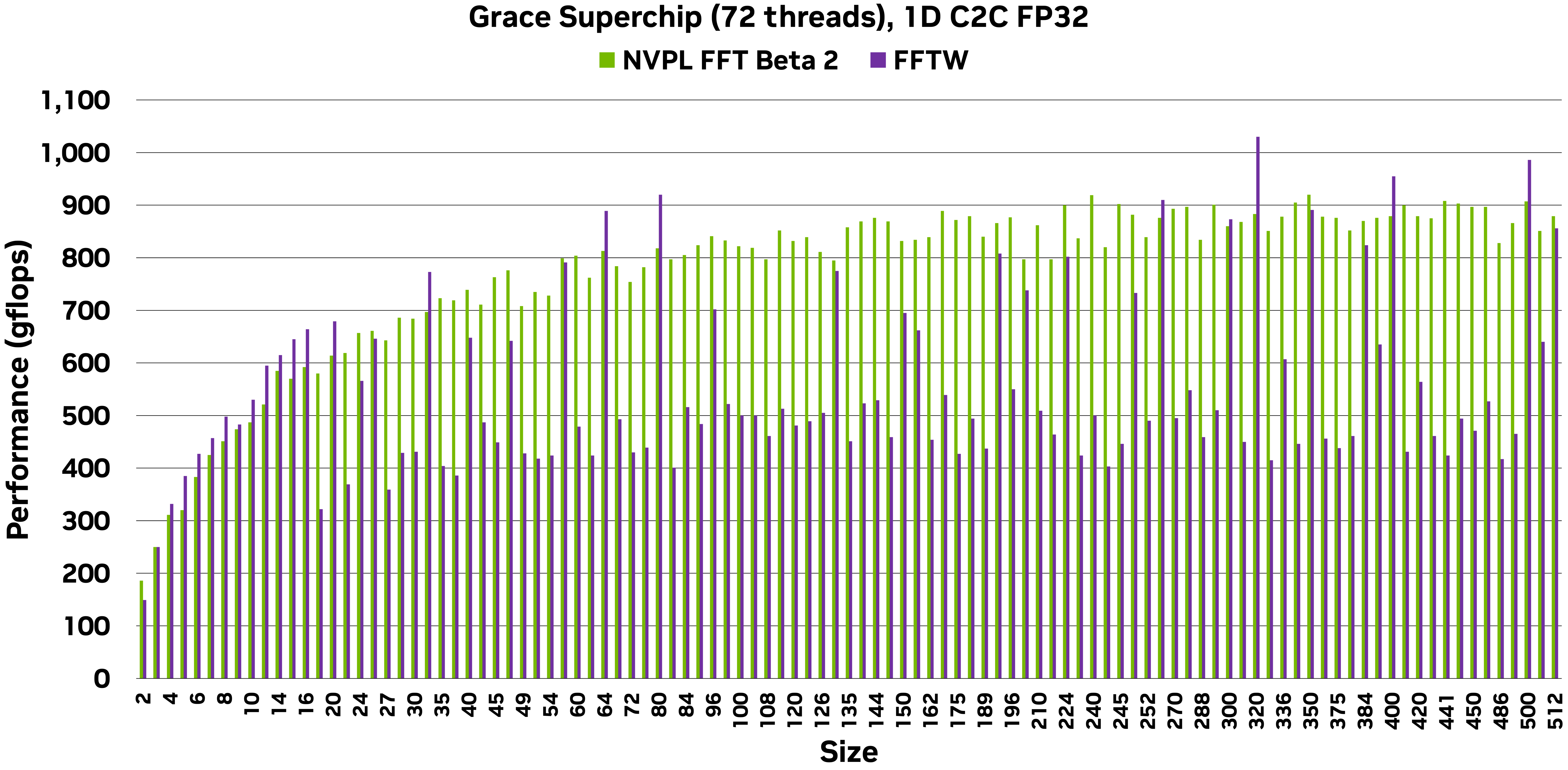
Avg. Transpose Trial Time [ms]

Eos (64 Node)

NVPL FFT: Beyond the GPU

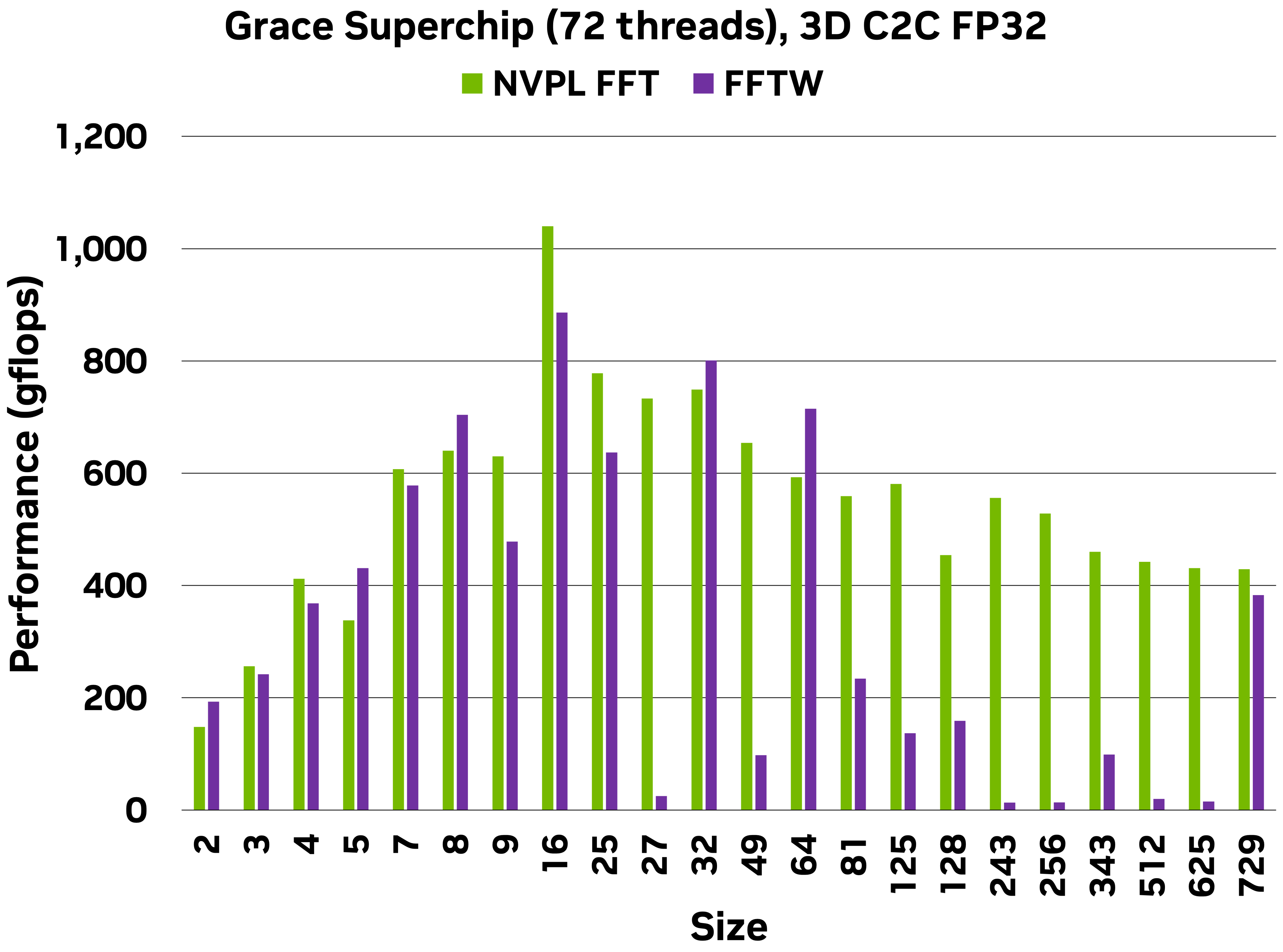
Beta 2 performance results on Grace Superchip

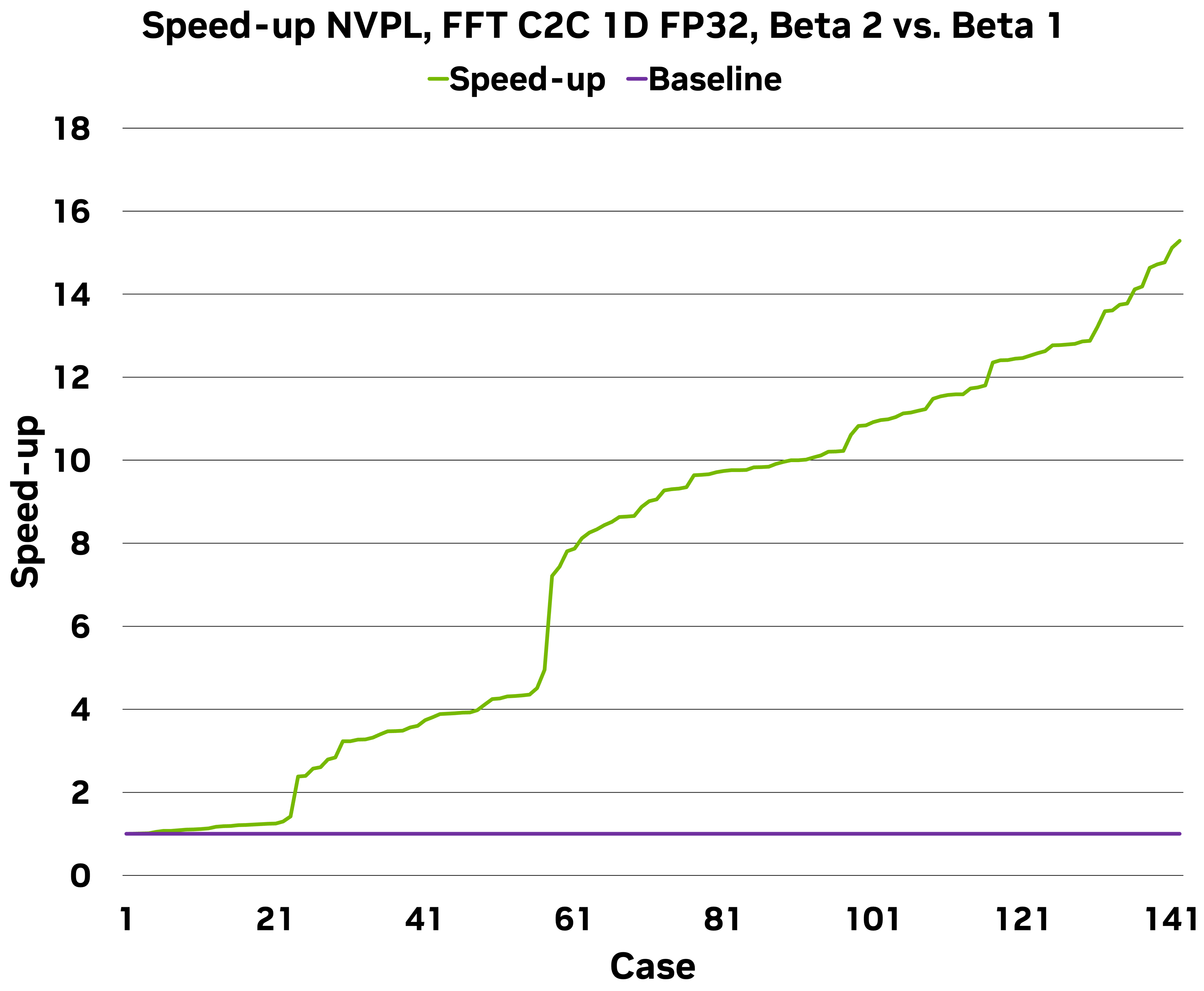
- NVPL FFT is an FFT library optimized for ARM CPUs, targeting the Grace Superchip
- Single-thread and multi-thread routines (based on GNU OpenMP)
- Beta 2 coming soon, with improved performance



NVPL FFT: Beyond the GPU

Beta 2 performance results on Grace Superchip





- Performance optimization is ongoing
  - Test your app, tell us where to improve
- Please share your use cases with us!

Call for feedback!

Roadmap

Building our roadmap based on our users' needs

	Available	time →		
cuFFT	64-bit LTO	32-bit LTO R2C/C2R perf	Normalization Zero padding	Mixed-precision, LTO customization
cuFFT LTO EA	LTO callbacks			
cuFFTDx	+ cuBLASDx	R2C / C2R performance	High-level API?	Multi-node?
cuFFTMp	Support for Grace-Hopper	Minor-version compat (GA)	Improved performance via LTO	
cuDecomp	0.3.1	QoL build improvements	New NCCL features	
NVPL FFT	Beta 2: improved performance	Improved performance	Guru API	

* Roadmap is tentative and subject to change

Acknowledgments and Contact

We need your feedback

Call for feedback!

- We need your input:
  - Use cases (problem characteristics, platforms, needs)
  - Feedback on our existing products and/or previews

- Contact us:
  - Łukasz Ligowski (FFT Engineering Manager), lligowski@nvidia.com
  - Miguel Ferrer Avila (FFT Library Lead), mferreravila@nvidia.com
  - Jakub Szuppe (Device eXtensions Lead), jszuppe@nvidia.com
  - Josh Romero (cuDecomp Lead), joshr@nvidia.com
  - Arthy Sundaram (CUDA Math Product Manager), asundaram@nvidia.com
  - Filippo Spiga (HPC Developer Relations Manager), fspiga@nvidia.com

- Acknowledgements and thank you:
  - cuFFT Team and NVIDIA
  - Prof. Daisuke Takahashi

Thank you!