THE USE OF THE CUFFTDX LIBRARY FOR PERFORMANCE OPTIMIZATION OF FOURIER-TRANSFORM BASED GPU ALGORITHMS

Bálint Tóth

University of Pannonia

Department of Electrical Engineering and Information Systems

OVERVIEW

- cuFFT and its limits
- Introduction to cuFFT Device Extensions
- C++ template metaprogramming
- C++ expression templates
- Features and configuration of cuFFTDx
- Example: Welch’s periodogram

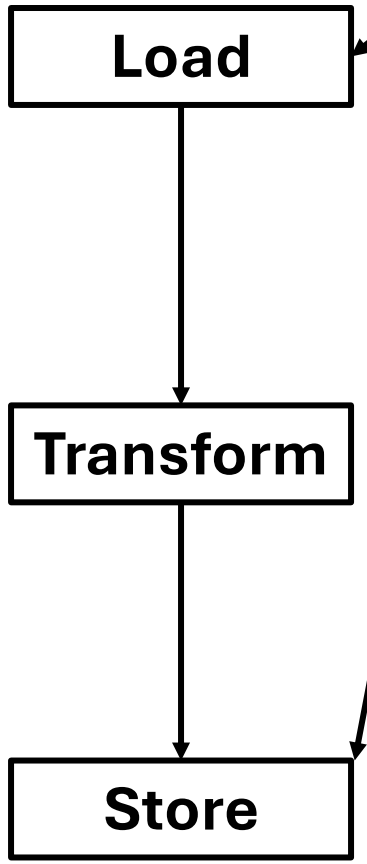
INTRODUCTION

- Fourier transform on the GPU
  - **Option A: use cuFFT[1]**
  - Option B: write it yourself
- cuFFT basics
  - Off-load some computation to the GPU
  - Ideal for already existing host-side code
- cuFFT workflow:
  - Configure transform (plan)
  - Copy data to GPU
  - Execute transform
  - Copy data back to host

```
cufftHandle plan;
cufftCreate(&plan);
cufftPlan1d(&plan,batch_size,
       CUFFT_R2C, batches);
cufftSetStream(plan, stream);

/* Copy data to device */

cufftExecR2C(plan,signal,spectrum);

/* Copy data back to host */
```

LIMITATIONS OF CUFFT

- Strictly host-side API
- Simple operations before & after transform:
  - Custom kernels: inefficient
- **Device functions[2]**
  - Only runs on load and store
  - Obstructs computation flow
  - Logic is still host side




```
__device__ void callback(
    void* dataOut,
    size_t offset,
    cufftComplex element,
    void* callerInfo,
    void *sharedPtr) {
    /* load / store op. */
}
```

```
__device__
cufftCallbackLoadR d_callback
  = callback;

cufftCallbackLoadR h_callback;

cudaMemcpyFromSymbol(
    &h_callback,
    d_callback,
    sizeof(h_callback));

cufftXtSetCallback(
    plan,
    (void **)&h_callback,
    CUFFT_CB_LD_REAL, 0);
```

CUFFT DEVICE EXTENSIONS (CUFFTDX[3])

- Part NVIDIA MathDx
  - cuFFTDx
  - cuBLASDx
- In-place FFT callable from kernel code
- Header only!

```
template<typename FFT>
__global__ void block_fft_kernel(/*...*/) {
    /* Copy data to registers */
    FFT().execute(thread_data,
        shared_mem, workspace);
}

/* ... */
using FFT = decltype(Size<128>()
                   + Type<fft_type::c2c>()
                   + FFTsPerBlock<1>()
                   + ElementsPerThread<8>()
                   + Block());
/* ... */

block_fft_kernel<FFT>
    <<<1, FFT::block_dim,
       FFT::shmem_size>>>(/*...*/);
```

TEMPLATE METAPROGRAMMING

- Generics: parameterize types compile time
- Turing complete, embedded language

```
template <typename T>
struct Array
{T* data; /*...*/};
```

Template declaration

```
template <>
struct Array<bool>
{unsigned char* data;};
```

Partial template specialization

```
Array<int> int_array;
```

```
struct Array
{int* data; /*...*/};
```

Template specialization

```
template <bool P, typename T, typename F>
struct conditional;

template <typename T,typename F>
struct conditional<true, T, F>
{ using type = T; };

template <typename T,typename F>
struct conditional<false, T, F>
{ using type = F; };
```

Conditional: a simple implementation of an ‚if' statement using partial template specialization

Factorial computation using a recursive meta function

```
template <int n>
struct factorial
{ static constexpr int value = n * factorial<n-1>::value; };

template <>
struct factorial<0>
{ static constexpr int value = 1; };
```

EXPRESSION TEMPLATES[4]

- Lazy evaluation in an eagerly-evaluated language
- Decouple an expression's definition from its evaluation

```
float A = 1.0f, B = 2.0f;
float x = 1.0;
func(A*x + B);
```

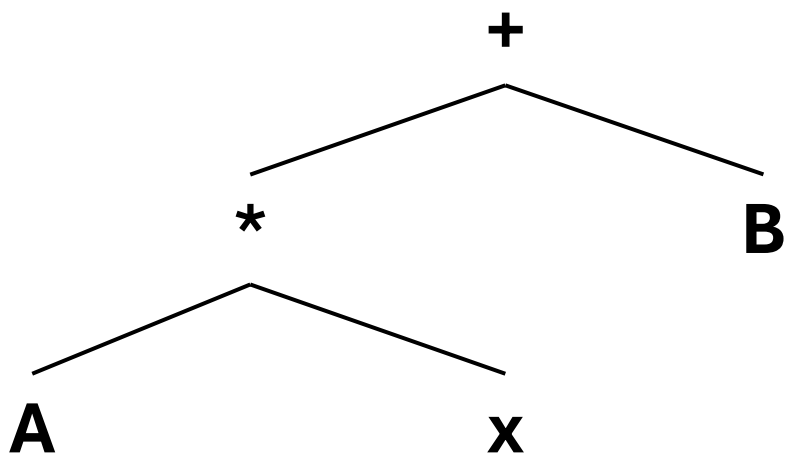
Eager evaluation:

1. Evaluate *tmp* = *A*x* + *B*
2. Call *func(tmp)*

```
Constant A(1.0f);
Constant B(1.0f);
Variable<float> x;

func(A*x + B);

/* ... */

template <typename Exp>
void func(const Exp& exp) {
    // custom operations
    exp(1.0f);
}
```

Lazy evaluation:

1. Construct expression tree:



2. Call *func()* with the expression
3. The function decides when to evaluate the expression

```
template <typename T>
class Constant {
private:
    T value;
public:
    using type = T;

    auto operator()(type const& var)
    const {
        return value;
    }
};
```

Leaf nodes of the expression tree

```
template <typename T,
          typename L,
          typename R,
          template <typename> typename Op>
class BinOp {
private:
    Op<T> op; L l; R r;
public:
    using type = T;

    auto operator ()(type const& var)
    const {
        return op(l(var),r(var));
    }
};
```

Junction nodes of the expression tree

```
template <typename L, typename R>
auto operator +(L const& l, R const& r) /* return type omitted */ {
    using type = decltype(typename L::type() + typename R::type());
    return BinOp(l, r, std::plus<type> {});
}
```

```
template <typename L, typename R>
auto operator *(L const& l, R const& r) /* return type omitted */ {
    using type = decltype(typename L::type() * typename R::type());
    return BinOp(l, r, std::multiplies<type> {});
}
```

Operators for expression tree construction

cuFFTDx - Operators

```
decltype(Size<128>()
 + Type<fft_type::c2c>()
 + FFTsPerBlock<1>()
 + ElementsPerThread<8>()
 + Block()
);
```

- Modified expression templates
- Dynamically build FFT description from types
  - Description operators: FFT parameters
  - Execution operators: operation mode selection
  - Operators „added” together

```
enum class fft_operator {
    size,
    type,
    /* ... */
};
```

Type class for all operators

```
template <int Value>
struct SizeOperator {
    using type = int;
    static constexpr int value = Value;
};
```

Contents of an operator

cuFFTDx - Traits

- Simple meta function to specify that a type conforms to some
- Standard traits available in the <type_traits> standard header
- Useful for error checking
- Parameter query functions in cuFFTDx

```
template <int Value>
struct is_operator<fft_operator,
                   fft_operator::size,
                   SizeOperator<Value>>
{ static constexpr bool value = true; };
```

Operator implementing the `is_operator` trait

cuFFTDX – Description & Execution

- Description holds a recipe for a transform
- Execution handles the internal implementation of the transform

```
template <typename... Operators>
struct fft_description {
    /* fft_operator_wrapper is a variadic template list */
    using description_type = fft_operator_wrapper<Operators...>;

    using this_size = get_or_default<
        fft_operator,
        fft_operator::size,
        description_type,
        SizeOperator<2>>::type;
    static constexpr auto size = this_size::value;
    /* ... */
};
```

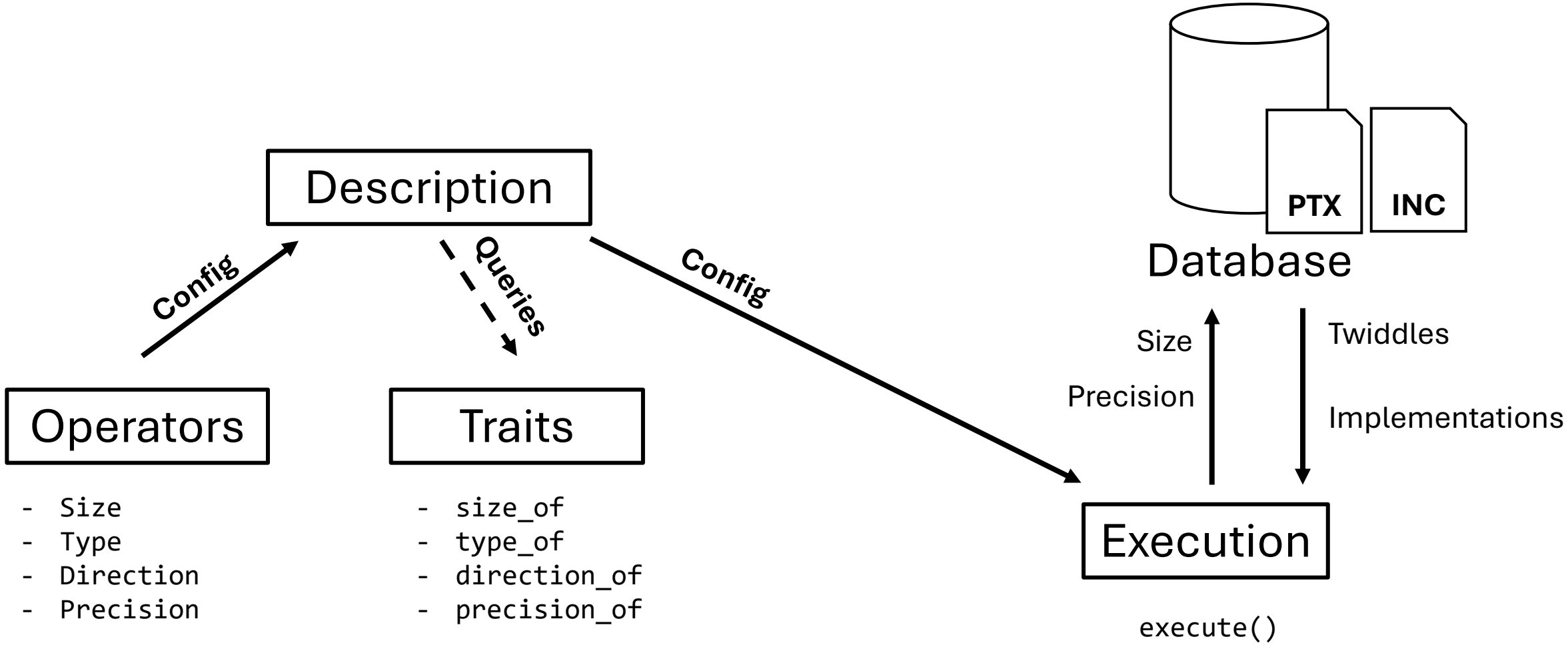
```
template <typename... O1s, typename... O2s>
auto operator +(fft_description<O1s...> const&, fft_description<O2s...> const&)
        -> fft_description<O1s..., O2s...> {
    return fft_description<O1s..., O2s...>();
}
```

Operator to assemble the description

```
template <typename OperatorTypeClass, OperatorTypeClass OperatorType,
    typename Description, typename Default = void>
struct get_or_default {
private:
    using get_type = get_t<OperatorTypeClass, OperatorType, Description>;
public:
    using type = typename std::conditional<
        std::is_void<get_type>::value,
        Default, get_type>::type;
};
```

Meta function to query a variadic argument list of operators and match the requested operator type

cuFFTDx – Internal structure



cuFFTDx - Features

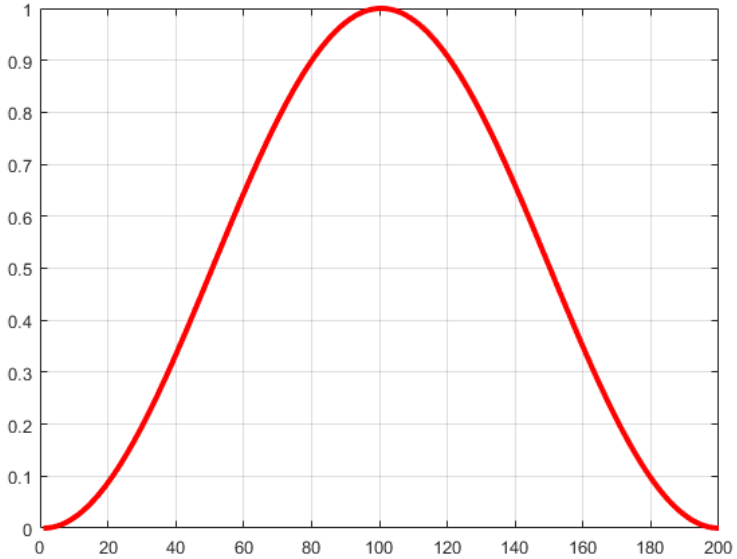
- Description operators:
  - `Size<unsigned int>`
  - `Direction<forward | inverse>`
  - `Type<c2c | r2c | c2r>`
  - `Precision<half | float | double>`
  - `SM<unsigned int>`
- Execution modes:
  - `Block`
  - `Thread`
- Traits:
  - `size_of<Description>::value`
  - `type_of<Description>::value`
  - `direction_of<Description>::value`
  - `precision_of<Description>::type`
  - `is_fft<Description>::value`
  - etc.

cuFFTDx - Limitations

- Header only library
  - Longer compile time
  - Small programmer's error results in ~100 lines of error message
- Pre-built FFT implementations
  - Longest input sequence: 32768 (float precision, Ampere architecture)
- Data layout in kernel has to follow cuFFTDx
  - Shared memory
  - Registers
  - Block and Thread mapping
- Parameters cannot change during runtime

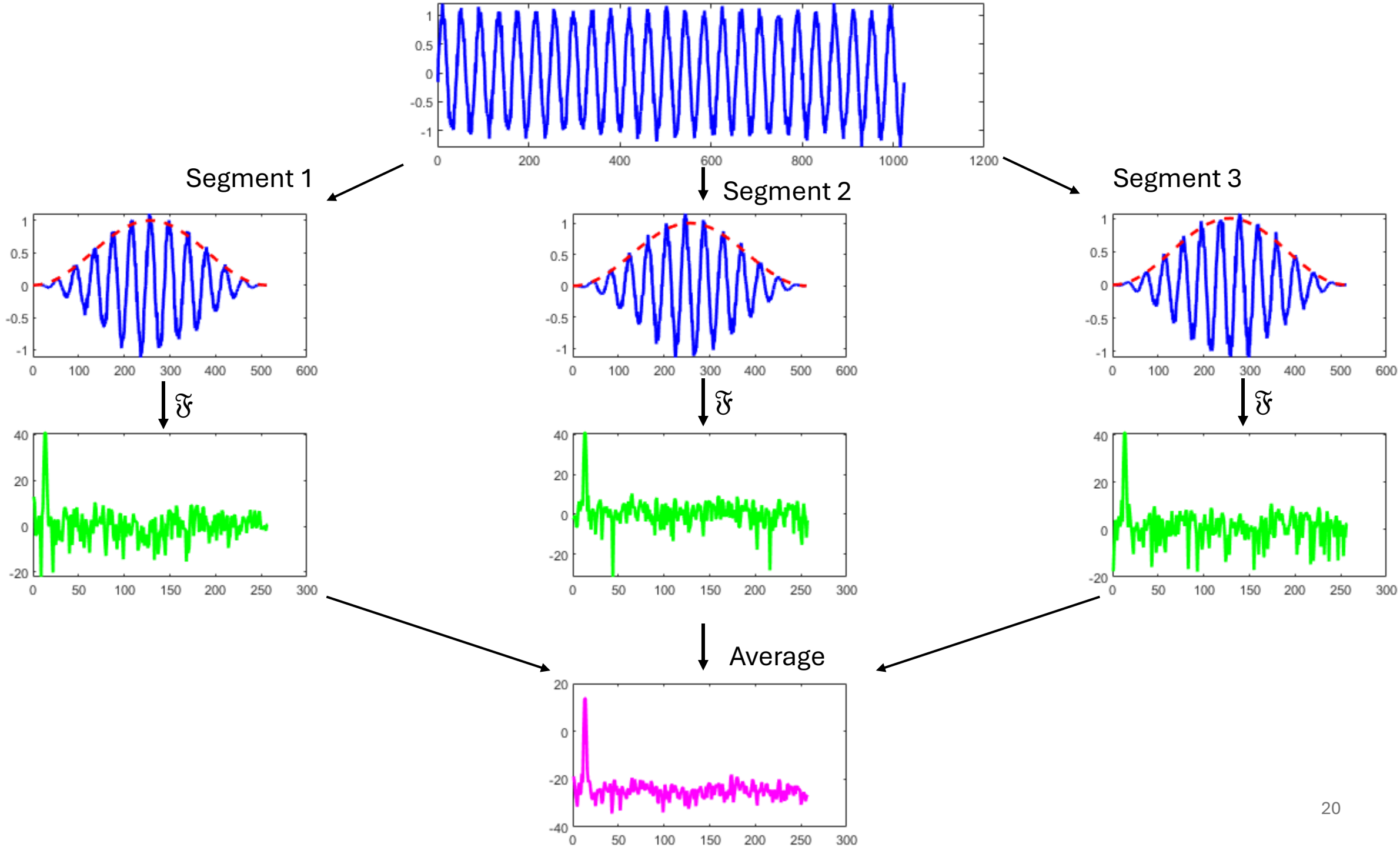
EXAMPLE: WELCH'S METHOD FOR PSD ESTIMATION[5]

- Modification of the classic periodogram
  - Periodogram is sensitive to noise
  - Welch's method is more robust to noise
- Overlapping segments
- Window functions
- **Multiple independent Fourier-transforms**
- Segment spectra averaged



$$P_x(l) = \frac{1}{K}\sum_{k=1}^{K}\frac{L}{U}|X_k|^2 \qquad X_k = \mathfrak{F}\{x_k\}$$

x_k: segment
N: signal length
K: number of segments
L: segment length


Segment 1
Segment 2
Segment 3
ℱ
ℱ
ℱ
Average

BASIC CUFFT

Configuration

```
cufftHandle plan;
cufftCreate(&plan);
cufftPlan1d(&plan,nfft,CUFFT_R2C,
    seg_num*channels);
cufftSetStream(plan, main_stream);
```

```
distribute<<<...>>>(signal,N,/*...*/);
cudaStreamSynchronize(main_stream);
```

```
cufftExecR2C(plan, fft_buffer, pxx);
cudaStreamSynchronize(main_stream);
```

```
average<<<...>>>(d_pxx, /*...*/);
cudaStreamSynchronize(main_stream);
```

```
__global__ void distribute(/*...*/) {
    buffer[global_id]
        = signal[channel_id]
        * window(x);
}
```

Windowing

```
__global__ void average(/*...*/) {
    atomicAdd(&(result[fft_id]) ,
      fft_buffer[fft_id]/segments));
}
```

Averaging

cuFFTDx

```
using FFT = decltype(Size <4096>()
                   + Type <fft_type::r2c>()
                   + Direction <fft_direction::forward>()
                   + Block() );
```

Configuration

```
template <typename FFT>
__global__ void multigrid_welch_dx(/*...*/) {
```

```
   for (unsigned i=0;i<FFT::elements_per_thread;i++)
        reinterpret_cast<real_type*>(thread_data)[i]
            = data[index] * window(x);
```

Windowing

```
    FFT().execute(/*...*/);
```

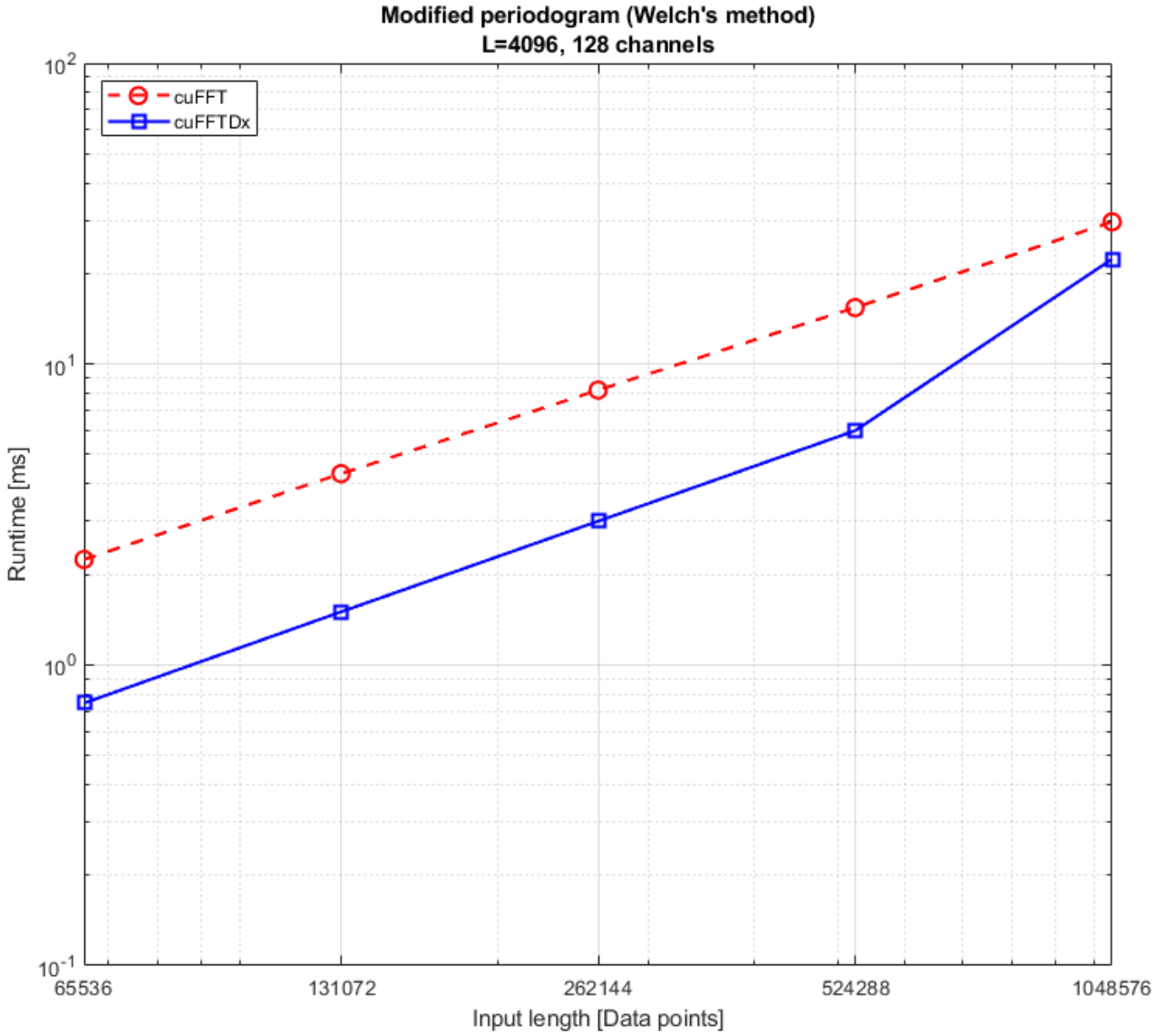
```
    for (unsigned i=0;i<FFT::elements_per_thread/2+1;i++)
        atomicAdd(&(result[index]),
                  thread_data[i]/block.group_dim().x);
```

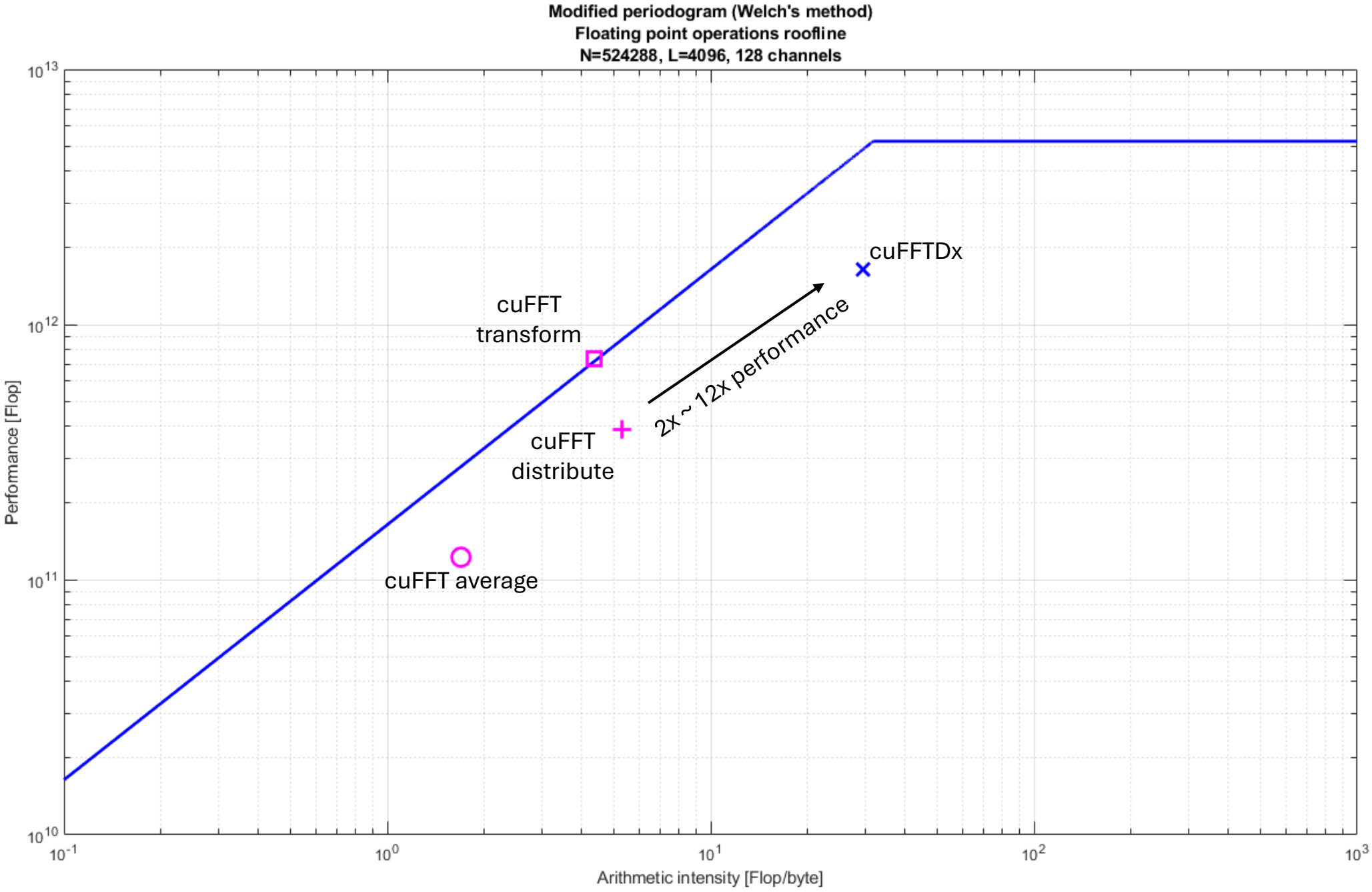
Averaging

```
}
```

RESULTS

- NVIDIA RTX 3050 Mobile
- Length from 65536 to 1048576
- 128 channels
- L=4096, 50% overlap
- Near-constant 2x speedup




Modified periodogram (Welch's method)
Floating point operations roofline
N=524288, L=4096, 128 channels
Perfomance [Flop]
Arithmetic intensity [Flop/byte]
cuFFT
transform
cuFFT
distribute
cuFFT average
cuFFTDx
2x ~ 12x performance

SUMMARY

- cuFFTDx
  - FFT that can be called from GPU code
  - Transform is configured compile time with Template Metaprogramming
  - Limited input length, longer compile time
- Demonstration example
  - Speedup: 2x compared to cuFFT implementation
  - Performance: over 1 TFlop/s sustained
  - Eliminated the need for low-performance kernels
  - Room for improvements
- Further experiments:
  - Use of Expression Templates in CUDA kernels (window functions)
  - Other DSP algorithms

REFERENCES & FURTHER READING

1. https://docs.nvidia.com/cuda/cufft/
2. https://developer.nvidia.com/blog/cuda-pro-tip-use-cufft-callbacks-custom-data-processing/
3. https://docs.nvidia.com/cuda/cufftdx/index.html
4. Veldhuizen, T. Expression templates. C++ Report, 1995, 7.5: 26-31.
5. P. Welch, “The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms,” IEEE Transactions on Audio and Electroacoustics, vol. 15, pp. 70–73, 1967.

More on expression templates and TMP:

- Langer, A., & Kreft, K. C++ Expression Templates Do you want high-performance, readable expressions? Expression templates do it all. C/C++ Users Journal, 2003, 27-34. (https://angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm)
- https://en.cppreference.com/w/cpp/header/type_traits