Leveraging MLIR for Loop Vectorization and GPU Porting of FFT Libraries

Yifei He, Artur Podobas and Stefano Markidis

KTH Royal Institute of Technology

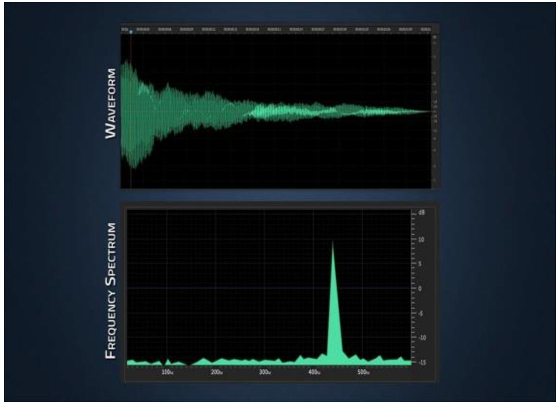
Outline

- Why new FFT library?
- FFT DSL and dialect
- Challeges: In FFT computation, how do we utilize MLIR/LLVM to achieve:
  - **Sparsity**
  - **Complex number handling**
  - **Parallellization**
- Progress & Plans

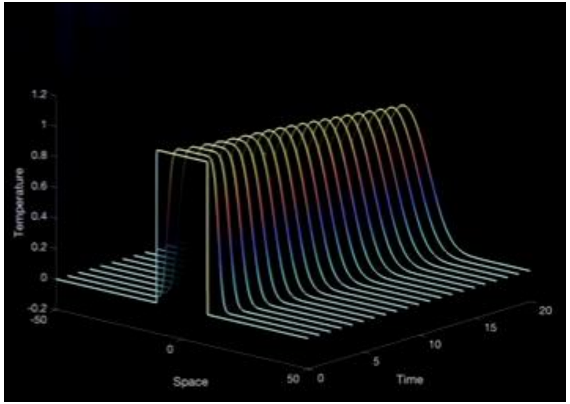
Motivation: Importance of FFT

- **Applications**

Signal processing



Partial Differential Equations(PDE)



- **Libraries for FFT:**

FFTW SPIRAL heFFTe Intel oneAPI Math Kernel Library CUFFT

Motivation:

- **Hand-written Libraries:**
  - Requires significant efforts for performance tuning
  - Hard to adapt to new hardwares

- **Compiler-based libraries like FFTW:**
  - Lack of portability over heterogeneous hardware (modern hardware features)
  - Cannot utilize the evolving compiler community
    - > MLIR/LLVM is more adaptive to search/learn based methods
  - Emit C level code, lack of control on low level compilation

FFT Algorithm in matrix-formalism

$\mathcal{O}(n^2)$

$$DFT_{N_{m,n}} = (\omega_N)^{mn}, \quad \text{where} \quad \omega_N = \exp(-2\pi i/N) \quad \text{for} \quad 0 \leq m,n < N.$$

$$\mathrm{DFT_N} = (\mathrm{DFT_K} \otimes \mathrm{I_M})\, \mathrm{D^N_M} (\mathrm{I_K} \otimes \mathrm{DFT_M})\, \Pi^{\mathrm{N}}_{\mathrm{K}} \quad \text{with} \quad N = MK$$

$\mathcal{O}(n \log n)$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & 1 & \\ & 1 & & 1 \\ 1 & & -1 & \\ & 1 & & -1 \end{bmatrix}}_{\mathrm{DFT_2} \otimes \mathrm{I_2}} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & -i \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix}}_{\mathrm{I_2} \otimes \mathrm{DFT_2}} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix}$$

FFTc DSL: Declarative representation of FFT tensor Algorithm

Fourier transform

Diagonal matrix (twiddles)

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{D}_4^2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \Pi_4^2$$

Kronecker product

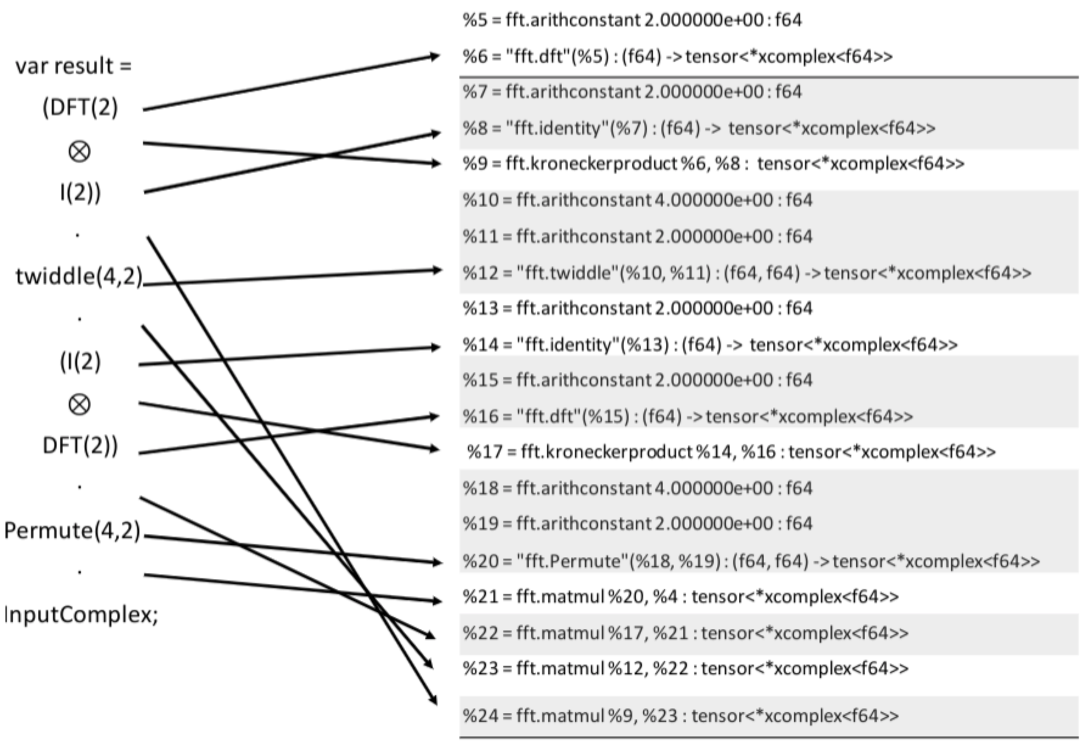
Identity

Permutation

```
var InputReal <4, 1> = [[1], [2], [3], [4]];
var InputImg  <4, 1> = [[1], [2], [3], [4]];
var InputComplex = createComplex(InputReal, InputImg);
var result =  (DFT(2) ⊗ I(2)) · twiddle(4,2) ·
              (I(2) ⊗ DFT(2)) · Permute(4,2) · InputComplex;
```

FFT Dialect: Operations, attributes, and types to represent the FFT formula

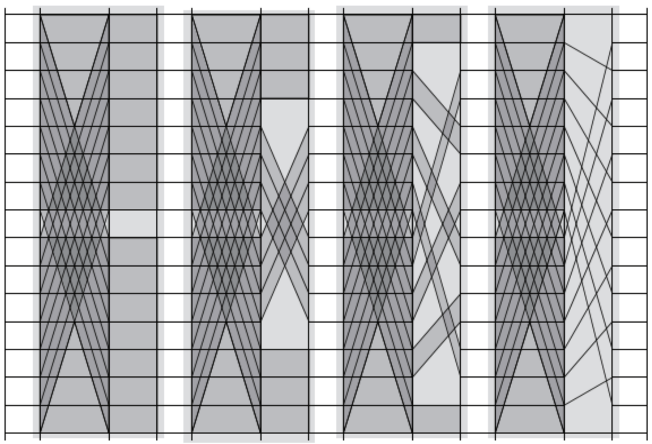
FFTc DSL	FFT Dialect
createComplex(A, B)	fft.createCT(a,b)
$A \cdot B$	fft.matmul a, b :
$A \otimes B$	fft.kroneckerproduct a, b
twiddle (a,b)	fft.twiddle (a , b)
I(size)	fft.identity (a)
DFT(size)	fft.dft(a)
Permute (a ,b)	fft.Permute(a, b)



Utilize sparsity in FFT Computation: Sparse Fusion

FFTc DSL Pattern	Sparse Fusion	Bufferization
$Y = (A_m \otimes I_n) \cdot X$	FusedMKIV(A, n, X)	$\text{for}(i = 0;\ i < n;\ i++)$ $Y[i : n : i + m * n - n] =$ $A*(X[i : n : i + m * n - n])$
$Y = (I_m \otimes A_n) \cdot X$	FusedIKMV(A, n, X)	$\text{for}(i = 0;\ i < m;\ i++)$ $Y[i * n : 1 : i * n + n - 1] =$ $A*(X[i * n : 1 : i * n + n - 1])$
$(\Pi_m^{mn} \otimes I_k) \cdot X$	FusedPKIV(m, mn, k, X)	$\text{for}(i = 0;\ i < m;\ i++)$ $\text{for}(j = 0;\ j < n;\ j++)$ $Y[k * (i + m * j) : 1 : k * (i + m * j)] =$ $X[k * (n * i + j) : 1 : k * (n * i + j)]$
$D_m^n \cdot X$	Mul(TwiddleCoe, X)	$\text{for}(i = 0;\ i < m;\ i++)$ $Y[i] = D_m^n[i] * X[i])$
$\Pi_m^{mn} \cdot X$	Permute(m, mn, X)	$\text{for}(i = 0;\ i < m;\ i++)$ $\text{for}(j = 0;\ j < n;\ j++)$ $Y[i + m * j : 1 : i + m * j] =$ $A*(X[n * i + j : 1 : n * i + j])$

From Stockham FFT to Vector Parallel Loops



$$DFT_m \otimes I_n$$

Fusion → Fused MKIV → Bufferization →

```
"affine.for" %arg2 = 0 to 16 {

  "affine.for" %arg3 = 0 to 16 {

    "affine.for" %arg4 = 0 to 16 {
      %8 = "affine.load" %7[%arg2, %arg3, 0] : memref
<16x16x2xf64>
      %10 = "affine.load" %4[%arg3 * 16 + %arg4, 0, 0]
: memref<256x2x1xf64>
      %12 = "arith.mulf" %8, %10 : f64
      ...
    }}}
```

MLIR Affine Loop Nests

[cite] Franchetti, Franz, and Markus Püschel. "Fast Fourier Transform." *Encyclopedia of Parallel Computing.*

Complex Data Handling: Convert Complex Data to an Array of Floating-point

- Problem: Complex data not a first-class type in LLVM, nor supported by most hardware ISA:
  - Vectorizers cannot work on complex data (aggregate data type)

- Solution:
  - fft-convert-complex-to-floating
  - fft-complex-mem-rep
  - affine-scalrep

- Benefits :
  - Vectorization enabled
  - Support multiple data layouts
    - Interleaved & Splited

```
From:
  %10 = "affine.load" %5[%arg0, 0] : memref<4x1xf64>
  %11 = "affine.load" %6[%arg0, 0] : memref<4x1xf64>
  %12 = "complex.create" %10, %11 : complex<f64>

To:
  %13 = "memref.alloc"() : memref<2xf64>
  %14 = "affine.load" %8[%arg0, 0] : memref<4x1xf64>
  %15 = "affine.load" %9[%arg0, 0] : memref<4x1xf64>
 "affine.store" %14, %13[0] : memref<2xf64>
 "affine.store" %15, %13[1] : memref<2xf64>
  %16 = builtin.unrealized_conversion_cast %13 :
 memref<2xf64> to complex<f64>
```

Automatic CPU Vectorization

- MLIR-SuperVectorize: all dimension
  - virtual vector operations -> machine-specific vector operations
- SLP vectorizer: innermost loop vectorized
- VPLAN vectorizer: outermost loop vectorized

```
for(i=0;i<M;i++)
  for(j=0;j<N;j++)
    c[i][j]=
    a[i][j]+
    b[i][j];
```

(a) Scalar Loop

```
for(i=0;i<M;i++)
  for(j=0;j<N;j+=4)
    c[i][j:j+3]=
    a[i][j:j+3]+
    b[i][j:j+3];
```

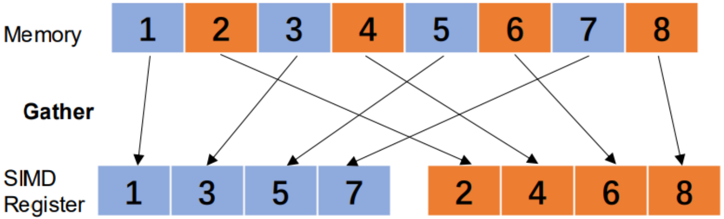
(b) Inner Loop Vectorized

```
for(i=0;i<M;i+=4)
  for(j=0;j<N;j++)
    c[i:i+3][j]=
    a[i:i+3][j]+
    b[i:i+3][j];
```

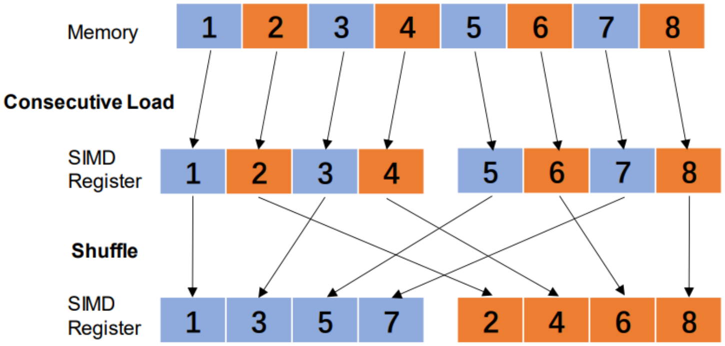
(c) Outer Loop Vectorized

Vectorization: Memory Access Optimization on the Fly

Interleaved memory access optimization for complex array



(a) Directly Load Complex Data Using Gather Instructions



(b) Optimized Interleaved Memory Access

CPU Vectorization

LLVM VPLAN Vectorizer

```
#Pack data from memory into SIMD register using masked
   gather intrinsic
%wide.masked.gather130 = tail call <8 x double> @llvm.
   masked.gather.v8f64.  p0(<8 x ptr> %1052, i32 8, <8
   x i1> <i1 true, i1 t  e, i1 true, i1 true, i1 true,
   i1 true, i1 true, i1 true>, <8 x double> undef), !
%wi                                              le> @llvm.
                                                 i32 8, <8
   x i1> <i1 true, i1 true, i1 true, i1 true, i1 true,
   i1 true, i1 true, i1 true>, <8 x double> undef), !
   dbg !2047
...
#Arithmetic operation on vectors
%1057 = fmul <8 x double> %wide.masked.gather130, %wide
   .masked.gather132, !dbg !2049
```

mask.gather

LLVM SLP Vectorizer (Interleaved)

```
#Pack data continuously from memory and then perform in
   -register shuffles
%wide.vec131 = load <16 x double>, ptr %1098, align 8,
   !dbg !2041
%strided.vec132 = shufflev   tor <16 x double> %wide.
   vec131, <16 x double> p   on, <8 x i32> <i32 0, i32
   2,
%str                                                  e.
   ve                                                 i32
   3, i32 5, i32 7, i32 9, i32 11, i32 13, i32 15>, !
%1100 = fmul <8 x double> %strided.vec130, %strided.
   vec133, !dbg !2043
%1101 = fsub <8 x double> %1099, %1100, !dbg !2044
%1102 = fmul <8 x double> %strided.vec130, %strided.
   vec132, !dbg !2045
%1103 = fmul <8 x double> %strided.vec129, %strided.
   vec133, !dbg !2046
%1104 = fadd <8 x double> %1102, %1103, !dbg !2047
```

Load<16 x double>

shufflevector<16 x double>

2023-10-08

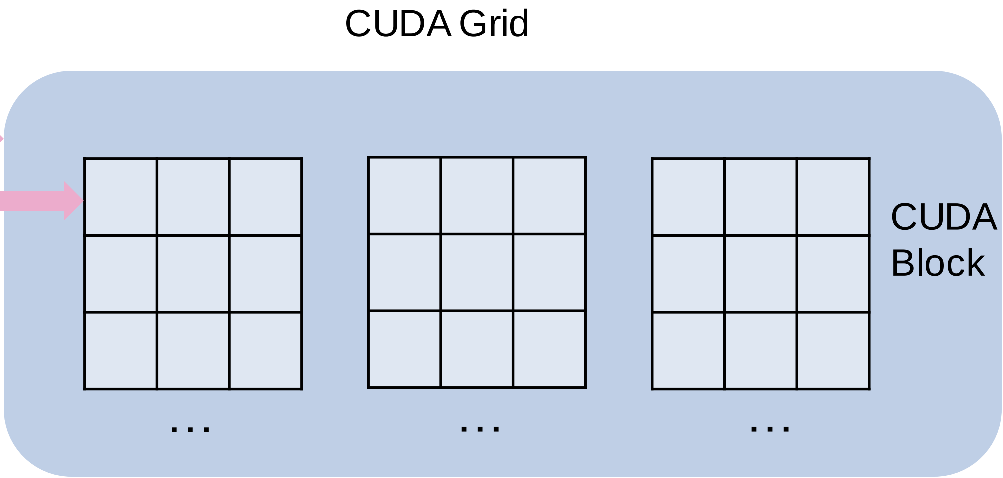
Automatic GPU Kernel Generation

Same Source code as CPU

```
"affine.for" %arg2 = 0 to 16 {

  "affine.for" %arg3 = 0 to 16 {

    "affine.for" %arg4 = 0 to 16 {
      %8 = "affine.load" %7[%arg2, %arg3, 0] : memref
   <16x16x2xf64>
      %10 = "affine.load" %4[%arg3 * 16 + %arg4, 0, 0]
   : memref<256x2x1xf64>
      %12 = "arith.mulf" %8, %10 : f64
      ...
   }}}
```



Map affine parallel loops to GPU parallel hierarchy

GPU Kernel → Convert to NVVM dialect → Convert GPU kernel to CUDA binary

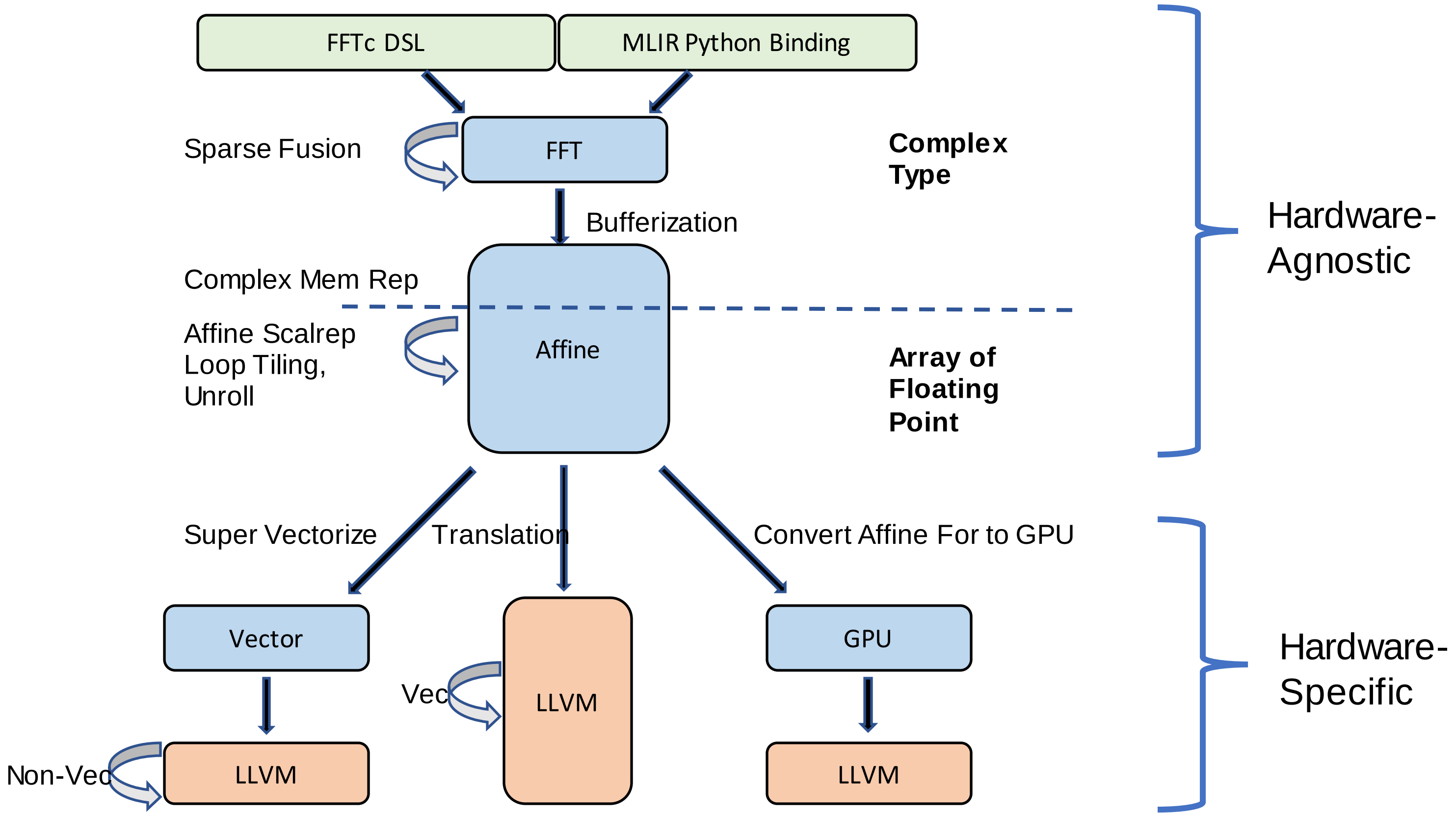
NVVM Dialect

CUDA Binary

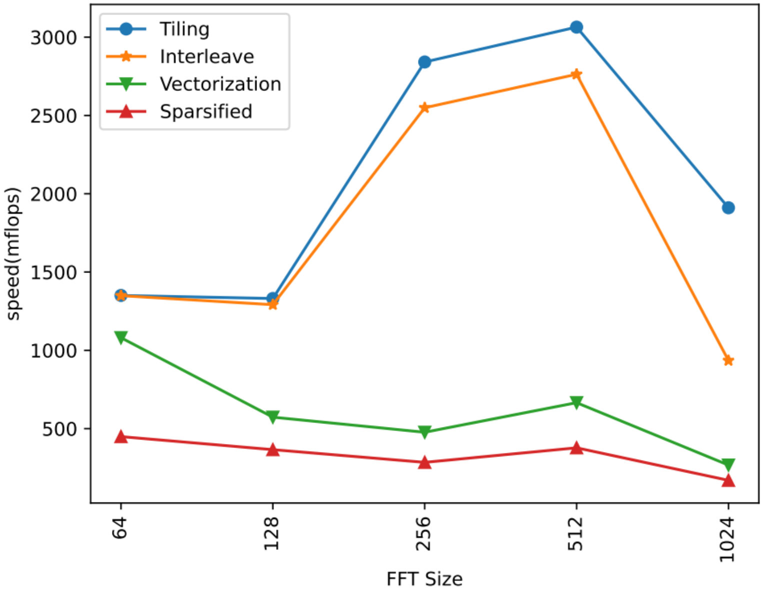
Host Code → Convert to LLVM dialect

LLVM Dialect

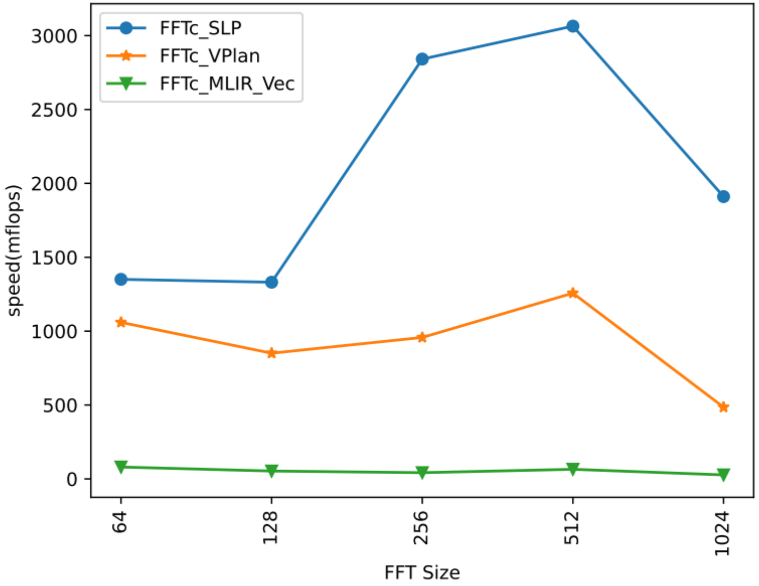
FFTc: Current Compilation Pipeline



Performance Evaluation: higher the better

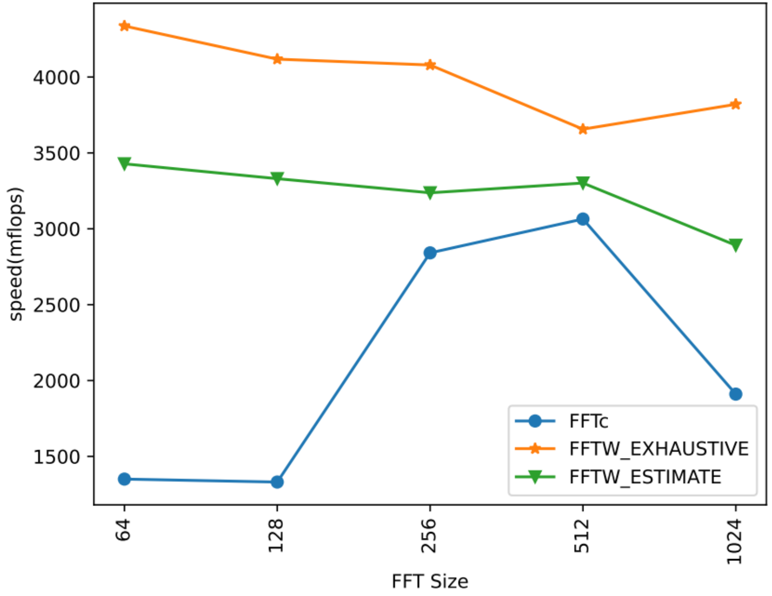


A): Difference optimizations

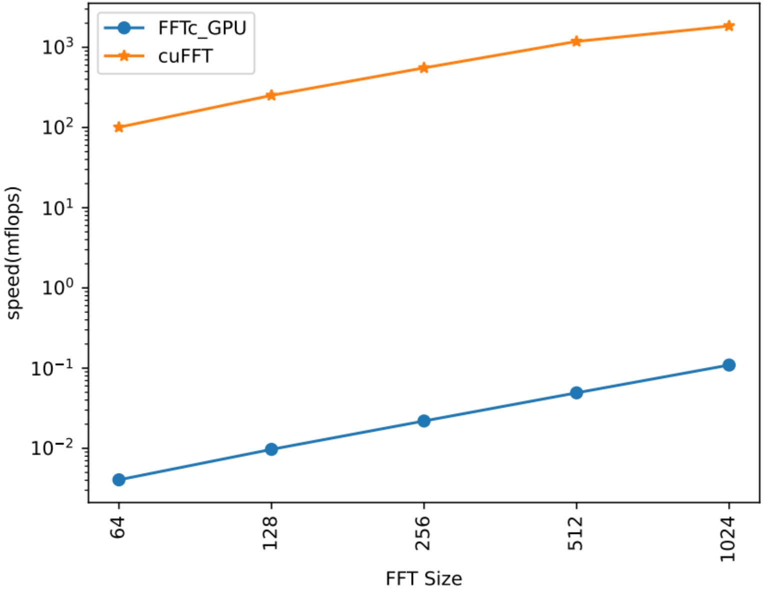


B): Difference vectorizers

Performance Evaluation



C): Compared with FFTW



D): GPU performance

Future Work

- **Fully Optimized Compilation:**
  - Auto-tuning for Loop tiling, vectorization, etc
  - Optimize MLIR vectorization
  - Data layout transformation for complex numbers
- **Support various hardware backends:**
  - CPU, NVIDIA/AMD GPU, Tensor core, FPGA, etc
- **Runtime**
  - Automatically generate decomposition plans(cost model)

Thanks!

Q&A