FFTc: An MLIR Dialect for Developing HPC Fast Fourier Transform Libraries

Yifei He, Artur Podobas,

Måns I. Andersson, Stefano Markidis

KTH Royal Institute of Technology

Outline

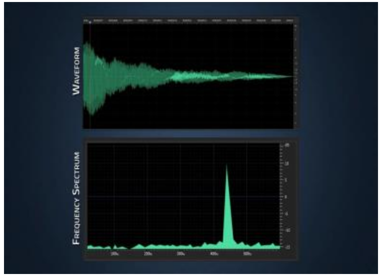
- Motivation
- Methodology
- Evaluation
- Conclusion & Future work

Motivation

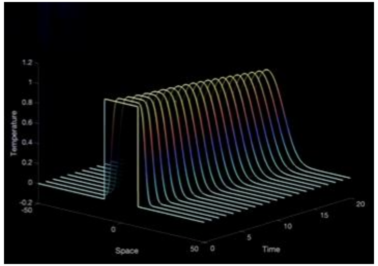
Motivation: Importance of FFT

- **Applications**

Signal processing



Partial Differential Equations(PDE)



- **Libraries for FFT:**

FFTE FFTPACK FFTW Intel oneAPI Math Kernel Library CUFFT

The Problems with FFT libraries like FFTW

- **Lack of support for modern hardware**
  - Newly introduced SIMD/tensor instructions in CPU, GPU, etc
- **Lack of portability over heterogeneous hardware**
  - Different code generation routines for different backends, cost is high
- **Cannot utilize the evolving compiler community**
  - MLIR/LLVM is more adaptive to search/learn based methods
- **Emit C code, lack of control on low level compilation**

FFT Algorithm in matrix-formalism

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

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

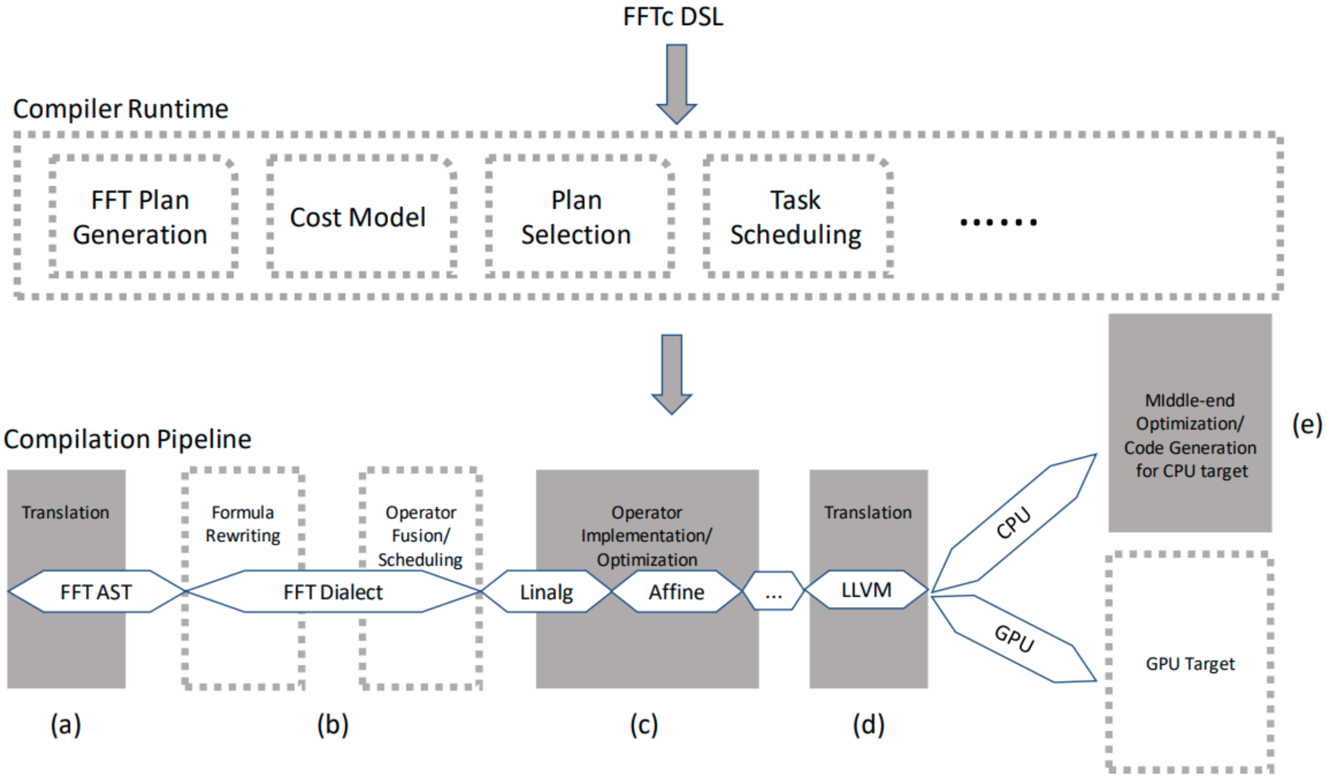
$$\mathrm{DFT_N} = (\mathrm{DFT_K} \otimes \mathrm{I_M})\, \mathrm{D^N_M} (\mathrm{I_K} \otimes \mathrm{DFT_M})\, \Pi^{\mathrm{N}}_{\mathrm{K}} \quad \text{with} \quad N = MK$$

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

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

Methodology

FFTc: A Domain Specific Compilation for Automatic Generation of FFT Algorithms



FFTc language: Declarative representation of FFT tensor Algorithm

Fourier transform — Diagonal matrix (twiddles)

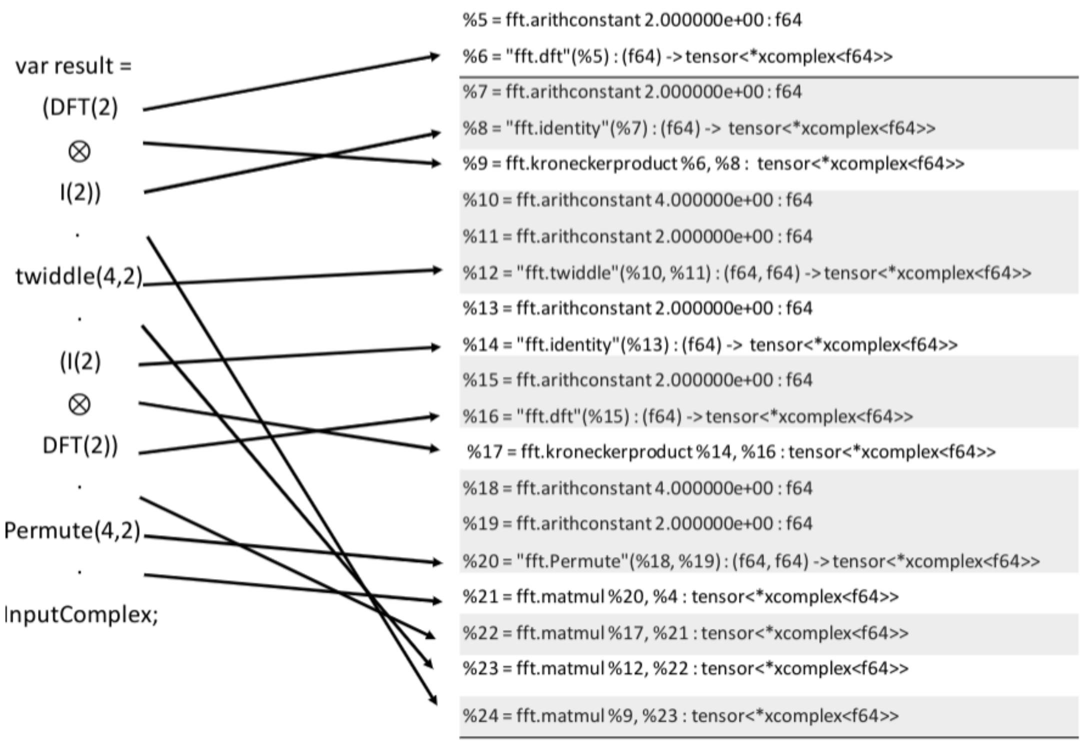
$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{D}_4^2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \Pi_4^2$$

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 (IR): Operations in FFT Dialect

FFTc DSL	FFT Dialect
createComplex(A, B)	fft.createCT(a,b)
A · B	fft.matmul a, b :
A ⊗ 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)



Progressive Lowering To Affine Dialect

```
From:
    %10 = fft.matmul %9, %3 : (tensor<4x4xcomplex<f64>>,
    tensor<4x1xcomplex<f64>>) ->
    tensor<4x1xcomplex<f64>>
```

```
To:
    affine.for %arg0 = 0 to 4 {
      affine.for %arg1 = 0 to 1 {
        affine.for %arg2 = 0 to 4 {
          %18 = affine.load %9[%arg0, %arg2] :
          memref<4x4xcomplex<f64>>
          %19 = affine.load %3[%arg2, %arg1] :
          memref<4x1xcomplex<f64>>
          %20 = complex.mul %18, %19 : complex<f64>
```

Different Code Generation Modes

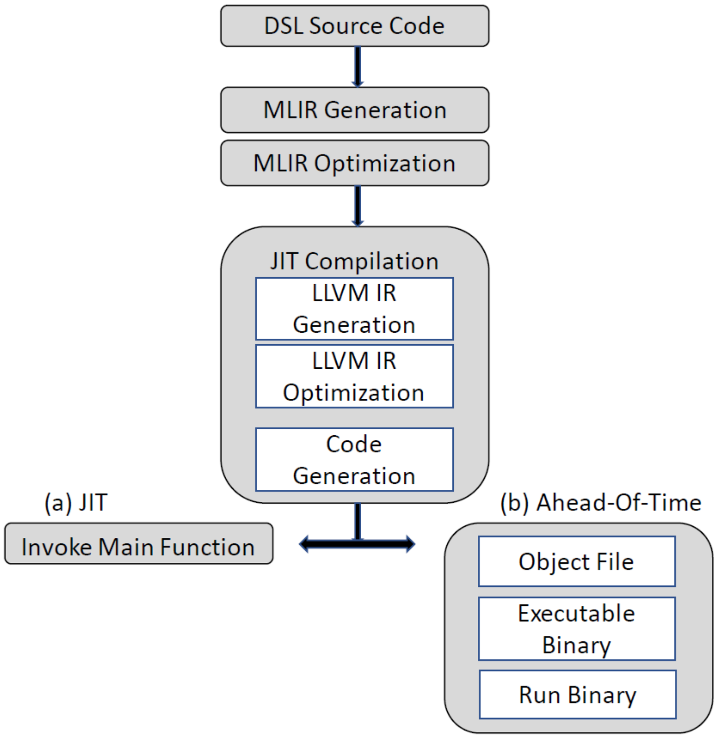
Ahead-Of-Time Compilation

Pros: Get rid of compilation time
Cons: Fixed FFT size for now

VS

Just-In-Time Compilation

Pros: Dynamic FFT size
Cons: Long compilation time



Evaluation

Performance Evaluation

Benchmark:
FFT from input size 32 to 128
Double complex input data
Single thread
Ahead-of-Time compilation mode

Evaluation:
Run for 1000 times, calculate standard deviation for 30 rounds

Hardware:
Dual-socket Intel Xeon Gold 6132 CPU, 192 GB of RAM

Performance Evaluation

FFT Size 32 Compile & Run: 6.8903s

- Frontend: 0.0%
- MLIR Compilation: 90.4%
- LLVM Middle-end optimization & Code Generation & Run: 8.9%

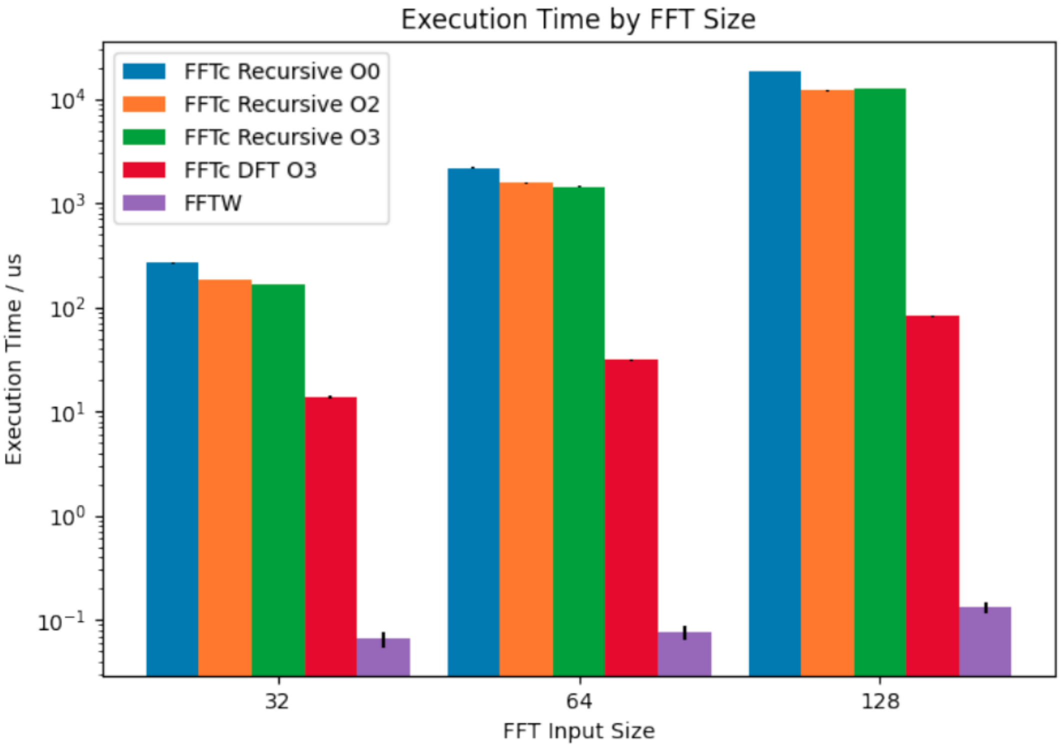
Execution Time Report in JIT Mode

Wall Time / Seconds	Name
0.0034 (0.0%)	Parser & MLIRGen
0.0003 (0.0%)	Inliner
0.0000 (0.0%)	(A) CallGraph
0.0000 (0.0%)	'builtin.func' Pipeline
0.0002 (0.0%)	Canonicalizer
6.2268 (90.4%)	builtin.func' Pipeline
0.0001 (0.0%)	{anonymous}::ShapeInference
0.0001 (0.0%)	Canonicalizer
0.0000 (0.0%)	CSE
0.0000 (0.0%)	(A) DominanceInfo
0.0116 (0.2%)	{anonymous}::AffineToLLVMLoweringPass
0.0226 (0.4%)	Canonicalizer
0.0014 (0.0%)	CSE
0.0000 (0.0%)	(A) DominanceInfo
0.6238 (9.1%)	AffineLoopFusion
5.5622 (80.7%)	AffineScalarReplacement
0.0000 (0.0%)	(A) PostDominanceInfo
0.0000 (0.0%)	(A) DominanceInfo
0.0009 (0.0%)	AffineLoopInvariantCodeMotion
0.0384 (0.6%)	{anonymous}::FFTToLLVMLoweringPass
0.0000 (0.0%)	output
0.6154 (8.9%)	Jit
0.0057 (0.1%)	Rest
6.8903 (100%)	Total

Performance Evaluation

- **O2**
  - Inliner, Canonicalizer, CSE
  - Affine: LoopFusion, LoopInvariantCodeMotion
  - LLVM O3 passes

- **O3**
  - MLIR O2 passes
  - Affine: ScalarReplacement
  - LLVM O3 passes



Performance Evaluation

- **Reasons contribute to the performance gap with FFTW**
  - The FFTs are computed through dense matrix-matrix multi-plication
  - Not fully optimized MLIR/LLVM compilation flow
  - No automatic FFT decomposition planner yet

Conclusion & Future Work

Conclusion

- **Tensor-based FFT DSL and FFT Dialect in MLIR**
  - DSL: Declarative representation of FFT tensor algorithm
  - FFT Dialect: Operations in FFT dialect to represent FFT algorithm
- **Code generation pipeline through MLIR and LLVM infrastructure**
  - Progressive lowering in MLIR for optimization & transformation at multiple abstraction level
  - Invoke LLVM JIT compilation for lower optimization on LLVM IR & code-generation

Future Work

- **Fully Optimized Compilation:**
  - FFT formula rewriting(decomposition): Pattern matching & Re-writing in MLIR
  - Loop tiling, vectorization
- **Support various hardware backends:**
  - CPU tensor unit, GPU, FPGA, etc
- **Reduce Compilation Time**
  - Multi-threading compilation & remove unnecessary MLIR passes
- **Dynamic FFT Size at Compilation Time**
  - Take advantage of MLIR bufferization process

Acknowledgement

This work is supported by IO-SEA under the European High-Performance Computing Joint Undertaking (JU)

Reference

- https://www.inf.ed.ac.uk/teaching/courses/ct/18-19/slides/llvm-1-intro.pdf
- https://llvm-hpc-2020-workshop.github.io/presentations/llvmhpc2020-amini.pdf