Optimizing Matrix Computations with PolyMage

Kumudha Narasimhan

Advisor : Dr. Uday Kumar Reddy B

Computer Science and Automation
Indian Institute of Science
Bengaluru, India

July 10, 2018

Overview

Introduction

Matrix computations are found in many domains:

- **Scientific computing**
  - Multi-resolution analysis kernel (MADNESS)(doitgen)
- **Neural networks**
  - Convolution operation is represented and matrix-matrix multiplication
  - Recurrent Neural networks consist of many matrix-vector multiplications
- **Digital signal processing**
  - convolution operations are used in low pass filters

These computations usually form the **bottleneck** in the applications and hence optimizing them will improve the performance of the application

Motivation -Current state-of-art

Opimized Libraries

FFTW

Optimizing Compilers

PLUTO

PPCG

DSLs

LGen:

Motivation -Current state-of-art

Opimized Libraries

FFTW

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*

Optimizing Compilers

PLUTO

PPCG

DSLs

FLAME

LGen:

Motivation -Current state-of-art

Opimized Libraries

FFTW

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*

Optimizing Compilers

PLUTO

PPCG

- *Compiler performs architecture independent optimization*
- *Better productivity than libraries*

DSLs

LGen:

Opimized Libraries

FFTW

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*

Optimizing Compilers

PLUTO

PPCG

- *Compiler performs architecture independent optimization*
- *Better productivity than libraries*

DSLs

FLAME

LGen:

- *Improves productivity*
- *Performs domain-specific optimizations*

Motivation -Current state-of-art

Opimized Libraries

FFTW

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*
- *Optimized only large matrix sizes*
- *Trade-off: Productivity for Generality*
- *No reuse across library calls*

Optimizing Compilers

PLUTO

PPCG

- *Compiler performs architecture independent optimization*
- *Better productivity than libraries*

DSLs

FLAME

LGen:

- *Improves productivity*
- *Performs domain-specific optimizations*

Opimized Libraries

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*
- *Optimized only large matrix sizes*
- *Trade-off: Productivity for Generality*
- *No reuse across library calls*

Optimizing Compilers

PLUTO

PPCG

- *Compiler performs architecture independent optimization*
- *Better productivity than libraries*
- *Manual tuning of tile sizes*
- *Does not map to library calls*

DSLs

LGen:

- *Improves productivity*
- *Performs domain-specific optimizations*

Opimized Libraries

FFTW

- *Hand Optimized or Highly tuned.*
- *Customized for various architectures*
- *Optimized only large matrix sizes*
- *Trade-off: Productivity for Generality*
- *No reuse across library calls*

Optimizing Compilers

PLUTO

PPCG

- *Compiler performs architecture independent optimization*
- *Better productivity than libraries*
- *Manual tuning of tile sizes*
- *Does not map to library calls*

DSLs

FLAME

LGen:

- *Improves productivity*
- *Performs domain-specific optimizations*
- *Naively map to library calls*
- *Target only small matrices*
- *Auto-tuning for locality optimizations*

Objective

- Perform Data Locality Optimizations
- Map to library calls
- Remove Manual or Auto-tuning
- Storage Optimization
- High level language constructs

Objective

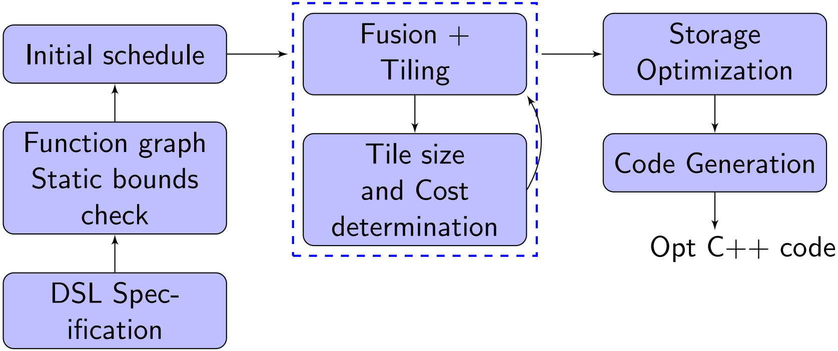
- Perform Data Locality Optimizations
- Map to library calls
- Remove Manual or Auto-tuning
- Storage Optimization : *Available in Polymage*
- High level language constructs: *Available in Polymage*

PolyMage

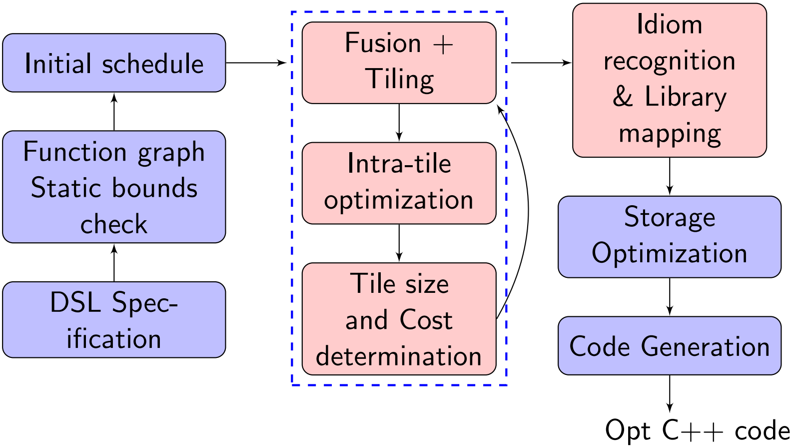
PolyMage is Domain Specific Language which supports optmizations for:

- stencil operations
- point-wise operations
- down-sample and up-sample operations

PolyMage


Initial schedule
Function graph Static bounds check
DSL Spec-ification
Fusion + Tiling
Tile size and Cost determination
Storage Optimization
Code Generation
Opt C++ code

PolyMage - Compiler Flow



Language Specification - Matmul Example

Existing PolyMage specification

```
# Parameters
N = Parameter(Int,"N")
# variables
i = Variable(Int,"i")
j = Variable(Int,"j")
k = Variable(Int,"k")
# Input
A = Image(Double,"A",[N, N])
B = Image(Double,"B",[N, N])
# Domain/ Intervals
n_dom = Interval(Int,0,N-1)

# Matrix multiplication operation
C = Reduction(([i,j], [n_dom,n_dom]),
              ([i,j,k],
               [n_dom,n_dom,n_dom]),
              Double,"C")
C.defn = [Reduce(C(i, j),
                 A(i, k) * B(k, j),
                 Op.Sum)]
```

Language Specification - Matmul Example

Existing PolyMage specification

```
# Parameters
N = Parameter(Int,"N")
# variables
i = Variable(Int,"i")
j = Variable(Int,"j")
k = Variable(Int,"k")
# Input
A = Image(Double,"A",[N, N])
B = Image(Double,"B",[N, N])
# Domain/ Intervals
n_dom = Interval(Int,0,N-1)

# Matrix multiplication operation
C = Reduction(([i,j], [n_dom,n_dom]),
              ([i,j,k],
               [n_dom,n_dom,n_dom]),
              Double,"C")
C.defn = [Reduce(C(i, j),
                 A(i, k) * B(k, j),
                 Op.Sum)]
```

New PolyMage specification

```
# Parameters
N = Parameter(Int,"N")

# Input matrices
A = Matrix(Double,"A",[N, N])
B = Matrix(Double,"B",[N, N])

# Matrix multiplication
C = A * B
```

Language Specification

Overloaded Operators introduced

Operator	Usage	Description
$+$	$\mathbf{A} + \mathbf{B}$	Point-wise addition
$-$	$\mathbf{A} - \mathbf{B}$	Point-wise subtraction
$*$	$\mathbf{A} * \mathbf{B}$	Multiplication

Language Specification

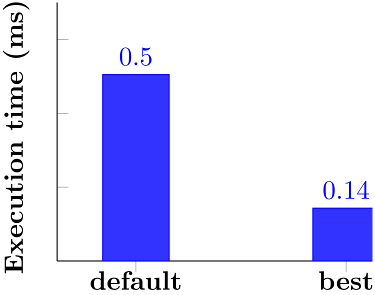
Overloaded Operators introduced

Operator	Usage	Description
$+$	$\mathbf{A} + \mathbf{B}$	Point-wise addition
$-$	$\mathbf{A} - \mathbf{B}$	Point-wise subtraction
$*$	$\mathbf{A} * \mathbf{B}$	Multiplication

Functions introduced

Function name with usage	Description
$\mathit{elementwise\_mul}(\mathbf{A}, \mathbf{B})$	Element-wise multiplication
$\mathit{scalar\_mul}(\mathbf{A}, \alpha)$	Matrix/Vector Scalar multiplication
$\mathit{transpose}(\mathbf{A})$	Transpose
$\mathit{symm}(\mathbf{matA}, \mathbf{matB}, \mathbf{matC}, M, N, \alpha, \beta)$	Symmetric Matrix multiply
$\mathit{syr2k}(\mathbf{matA}, \mathbf{matB}, \mathbf{matC}, \alpha, \beta)$	Symmetric rank-2k operations
$\mathit{syrk}(\mathbf{matA}, \mathbf{matC}, \alpha, \beta)$	Symmetric rank-k operations
$\mathit{trmm}(\mathbf{matA}, \mathbf{matB}, \alpha, \beta)$	Triangular Matrix multiply

Tile Size Selection Model



- Tile size has an effect on performance
- 3.57× improvement between default and best tile size for matmul

Tile Size Selection Model - Matmul

- Based on dimensional reuse along a dimension
- Let t_i, t_j and t_k be tile sizes for loops i, j and k respectively
- Tile Volume is given by:

$$t_i * t_j + t_j * t_k + t_k * t_i = T. \tag{1}$$

- Let tile size for dim i be $t_i = \gamma_i * t$ where γ_i is the dimensional reuse

Tile Size Selection Model - Matmul

- Based on dimensional reuse along a dimension
- Let t_i, t_j and t_k be tile sizes for loops i, j and k respectively
- Tile Volume is given by:

$$t_i * t_j + t_j * t_k + t_k * t_i = T. \quad (1)$$

- Let tile size for dim i be $t_i = \gamma_i * t$ where γ_i is the dimensional reuse

$$(\gamma_i * \gamma_j + \gamma_j * \gamma_k + \gamma_k * \gamma_i) * t^2 = \mathbf{C}. \quad (2)$$

$$\gamma_i = 0.5, \gamma_j = 0.5, \gamma_k = 1$$

$$(0.5 * t) * 256 + 256 * (0.5 * t) + (1.0 * 0.5) * t^2 = (32768/8) \quad (3)$$

$$0.5 * t^2 + 256 * t - 4096 = 0 \quad (4)$$

$t_i = 7, t_j = 256 and t_k = 15.$

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i + j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i - j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i + j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i - j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

DSP Code Snippet

```
for (int ii=0;(ii<=t1);ii++)
  for (int jj=0;(jj<=t2);jj++)
    ybs[ii]+=yds[(M + ii) - jj]
                  * window[jj];
```

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i+j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i-j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

DSP Code Snippet

```
for (int ii=0;(ii<=t1);ii++)
  for (int jj=0;(jj<=t2);jj++)
    ybs[ii]+=yds[(M + ii) - jj]
                    * window[jj];
```

- Let tile size of i be $t1$ and j be $t2$

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i+j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i-j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

DSP Code Snippet

```
for (int ii=0;(ii<=t1);ii++)
  for (int jj=0;(jj<=t2);jj++)
    ybs[ii]+=yds[(M + ii) - jj]
                    * window[jj];
```

- Let tile size of i be $t1$ and j be $t2$
- Memory required by *ybs* is $t1$ and *window* is $t2$

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i+j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i-j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

DSP Code Snippet

```
for (int ii=0;(ii<=t1); ii++)
  for (int jj=0;(jj<=t2); jj++)
    ybs[ii]+=yds[(M + ii) - jj]
                        * window[jj];
```

- Let tile size of i be $t1$ and j be $t2$
- Memory required by *ybs* is $t1$ and *window* is $t2$
- Memory required by *yds* is calculated as $(t1 - 0) - (0 - t2) = t1 + t2$

Tile Size Selection Model - Reuse Equation

Reuse Equations

access	distinct accesses	reuse equation
$a[i]$	t_i	$\gamma_i * t$
$a[\alpha * i]$	t_i	$\gamma_i * t$
$a[i + j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i - j]$	$t_i + t_j$	$(\gamma_i + \gamma_j) * t$
$a[i][j]$	$t_i * t_j$	$(\gamma_i * \gamma_j) * t$

DSP Code Snippet

```
for (int ii=0;(ii<=t1);ii++)
  for (int jj=0;(jj<=t2);jj++)
    ybs[ii]+=yds[(M + ii) - jj]
                      * window[jj];
```

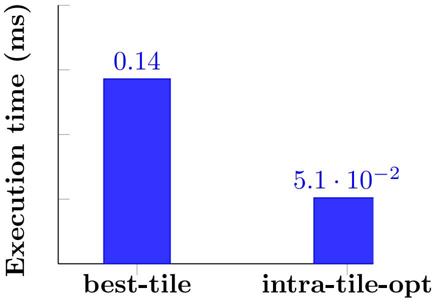
- Let tile size of i be $t1$ and j be $t2$
- Memory required by *ybs* is $t1$ and *window* is $t2$
- Memory required by *yds* is calculated as $(t1 - 0) - (0 - t2) = t1 + t2$
- $(t1) + (t1 + t2) + (t2) = T$

Tile Size Selection Model - Algorithm

Input: group G, cache_size, inner_tile_size, inner_dim, nDims
Output: Tile sizes of each dimension of G

```
Function ComputeTileSize(G, cache_size, inner_tile_size, inner_dim, nDims):
    dim_reuse [1...nDims] ← getDimReuse(G)
    inner_dim_size ← getInnerDimSize(G)
    tile_sizes [inner_dim] ← min (inner_dim_size, inner_tile_size )
    mem_access ← distinct memory references in G
    reuse_eqn ← getReuseEquation (mem_access, dim_reuse, inner_dim,
                    tile_size )
    root ← floor(positive_root(reuse_eqn))
    for each i ∈ nDims do
        tile_sizes [i] ← dim_reuse[i] * root
    endfor
    return tile_sizes
```

Intra-tile optimization



- Vectorization benefits performance
- Performance benefits by making the inner-loop vectorizable
- 2.74× improvement over best performing tiled code.

Intra-tile optimization - Matmul

```
for(int i = 0; i <= NI; i=i+1)
  for(int j = 0; j <= NJ; j=j+1)
    for(int k = 0; k <= NK; k=k+1)
      C[i][j] = C[i][j] + (A[i][k]
                          * B[k][j]);
```

- loop k carries a dependence => Not parallel => Not vectorizable
- loop i has non contiguous accesses for arrays C and A => Not vectorizable

Intra-tile optimization - Matmul

```
for(int i = 0; i <= NI; i=i+1)
  for(int j = 0; j <= NJ; j=j+1)
    for(int k = 0; k <= NK; k=k+1)
      C[i][j] = C[i][j] + (A[i][k]
                        * B[k][j]);
```

dim	s	t	v	a	score
i	0	1	false	4	-44
j	3	1	true	4	18
k	1	2	false	4	-6

- loop k carries a dependence => Not parallel => Not vectorizable
- loop i has non contiguous accesses for arrays C and A => Not vectorizable
- $score = score + (2 * s) + (4 * t) + (8 * v) - (16 * (a - s - t))$
- loop j has the highest score and is selected as the inner-most dimension

Intra-tile optimization

Algorithm 1: INTRA-TILE OPTIMIZATION

Input: group (G)
Output: Innermost dimension for each function in G

```
Function Intra-tile Optimization(G):
    for each function (f) ∈ group (G) do
        for each dimension (d) ∈ function (f) do
            s ← getNumSpatialReuse(d, f)
            t ← getNumTemporalReuse(d, f)
            a ← getTotalAccess(d, f)
            v ← isVectorizable(d, f)
            score[d] ← score[d] +
                    (2 ∗ s) + (4 ∗ t) + (8 ∗ v) − (16 ∗ (a − s − t))
        endfor
    endfor
    inner_dim ← getDimWithMaxScore(score)
    return inner_dim
```

Mapping to function calls

- Idiom Recognition stage - Traverse the AST
  - Allows to maintain backward compatibility
- Map to libraries only when profitable
- BLAS Routines
  - Mapped only when $M * N * K \geq (256)^3$
  - Obtained after experimental evaluation
- FFTW: Always. Reduces complexity from
  - Always mapped to library call.
  - Reduces complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$

Fusion for Reductions

```
for (i1 = 0; i < NI; i++)
  for (j1 = 0; j < NJ; j++)
    for (k1 = 0; k < NK; ++k)
      tmp[i1][j1] += A[i1][k1] * B[k1][j1]; \\S1
for (i2 = 0; i < NI; i++)
  for (j2 = 0; j < NL; j++)
      out[i2][j2] = tmp[i2][j2] + b[j2]; \\S2
```

Fusion for Reductions

```
1 for (i1 = 0; i < NI; i++)
2   for (j1 = 0; j < NJ; j++)
3     for (k1 = 0; k < NK; ++k)
4       tmp[i1][j1] += A[i1][k1] * B[k1][j1]; \\S1
5 for (i2 = 0; i < NI; i++)
6   for (j2 = 0; j < NL; j++)
7       out[i2][j2] = tmp[i2][j2] + b[j2]; \\S2
```

- RAW dependence on Line 4 by $tmp[i1][j1]$
- RAW dependence between write of $tmp[i1][j1]$ on Line 4 and read of $tmp[i2][j2]$ on Line 7
- Dimension matching or Alignment is applied for S1 and S2
- Fusion happens only if the alignment is successful
- **Alignment Vectors** S1: [i1, j1, k1], S2: [i2, j2, -]
- loops $i1, i2$ and $j1, j2$ are fused.

Experimental Setup

Processors	2-socket Intel Xeon E5-2630 v3
Clock	2.40 GHz
Cores	16 (8 per socket)
Hyperthreading	disabled
Private caches	64 KB L1 cache, 512 KB L2 cache
Shared cache	20,480 KB L3 cache
Memory	64 GB DDR4
Matlab version	9.3.0.713579 (R2017b)
Scipy version	1.0.0
Compiler	Intel C/C++ (icc/icpc) 18.0.1
Compiler flags	-O3 -xhost -qopenmp -fma -ipo
OS	Linux kernel 3.10.0 (64-bit) (Cent OS 7.3)

Benchmarks

PolyBench

- blas computations from linear algebra benchmark
- kernel computations from linear algebra benchmark

Digital Signal Processing

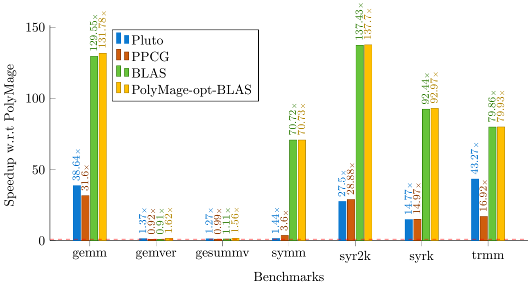
- unwanted spectral filter: Removes noise in input signal
- vuvuzela filter: Filters out vuvuzela noise from input signal

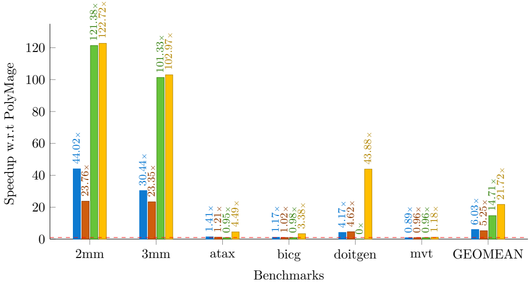
Image Processing

- To compare our tile size model with state-of-art

Performance Analysis - PolyBench

Speedup for EXTRALARGE Dataset



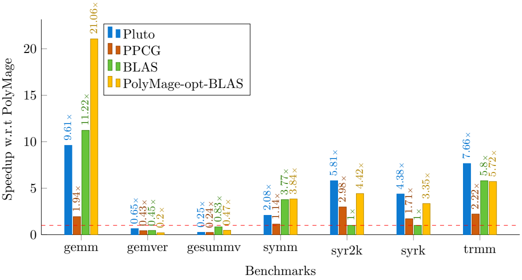


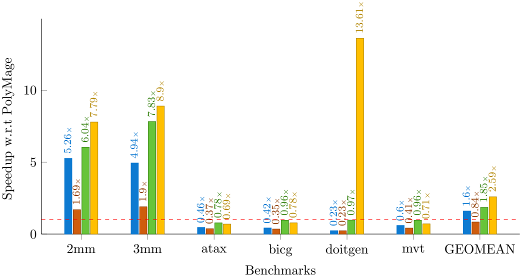
Mean Speedup

- 3.6× over Pluto
- 4.1× over PPCG
- 7.5× Polymage-unop
- 39% over Intel MKL

Performance Analysis - PolyBench

Speedup for EXTRALARGE Dataset

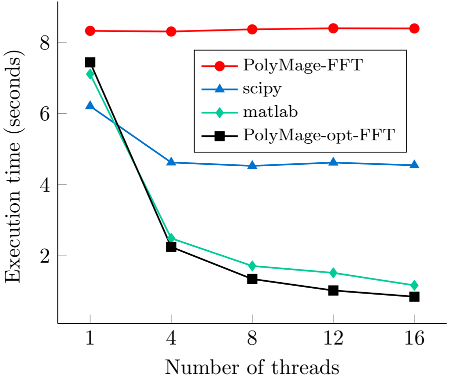


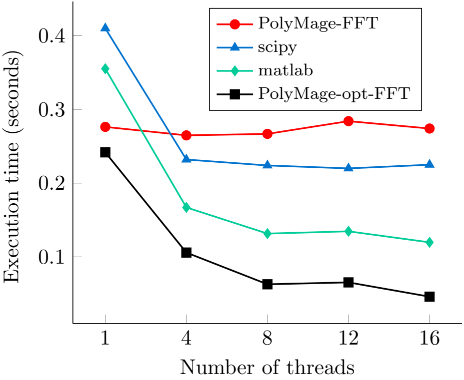


Mean Speedup

- 3.6× over Pluto
- 4.1× over PPCG
- 7.5× Polymage-unop
- 39% over Intel MKL

Execution time for DSP benchmarks and scaling across cores





Mean speed up of

- 7.7× over existing PolyMage optimizer,
- 5.1× over Intel's Scipy and
- 1.9× over MATLAB

Conclusion

- A DSL approach to optimize Matrix Computations
- Tile size selection model which is applicable for arbitrary affine access
- Implemented a heuristic to map to function calls when profitable
- Implemented an intra-tile optmization algorithm to enhance auto-vectorization
- PolyBench Benchmarks: Speedup of 3.6× over Pluto, 4.1× over PPCG
- DSP Benchmarks: speedup of 5.1× over Intel's Scipy and 1.9× over MATLAB