The Pluto Compiler and its Use for Computational Sciences

Uday Bondhugula

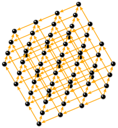
uday@csa.iisc.ernet.in

Dept of CSA

Indian Institute of Science

Bangalore 560012 India

with **Aravind Acharya, Vinay Vasista, Kumudha KN, Irshad Pananilath, Ravi Teja Mullapudi**



What is Pluto?

- **A source-to-source optimizer and parallelizer**

What is Pluto?

- **A source-to-source optimizer and parallelizer**
- **Uses many other polyhedral libraries and tools like ISL, Polylib, Cloog, Pet, Clan, Candl**

How can Pluto be used?

- **Push button:** fully automatically for optimization (tiling and other transformations), parallelization
- **Almost automatic:** With an understanding of what Pluto does, use it to obtain desired result
- **DSLs**: In domain-specific compilers/optimizers or library generators

How can Pluto be used?

- **Push button:** fully automatically for optimization (tiling and other transformations), parallelization
- **Almost automatic: Having understood what Pluto does, use it to obtain desired result**
- **DSLs**: In domain-specific compilers/optimizers or library generators

Big Picture: Role of Compilers

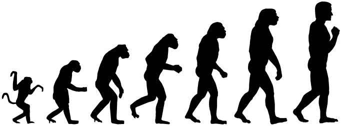
- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- LLVM/Polly, GCC/Graphite, PPCG, Pluto, other compilers/tools

- Build new **domain-specific languages and compilers**
- Scientists say **WHAT** they execute and not **HOW** they execute

Big Picture: Role of Compilers

The Evolutionary Approach

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- LLVM/Polly, GCC/Graphite, PPCG, Pluto, other compilers/tools



The Revolutionary Approach

- Build new **domain-specific languages and compilers**
- Scientists say **WHAT** they execute and not **HOW** they execute

Important to pursue both

OUTLINE

1. Pluto/Pluto+
   - Affine Transformations
   - Tiling
2. Case Studies
   - Solving Partial Differential Equations
   - Lattice Boltzmann Method
   - Image Processing Pipelines
3. Conclusions

Outline

1. Pluto/Pluto+
   - Affine Transformations
   - Tiling
2. Case Studies
   - Solving Partial Differential Equations
   - Lattice Boltzmann Method
   - Image Processing Pipelines
3. Conclusions

Affine Transformations

- Examples of affine functions of i, j: $i + j$, $i - j$, $i + 1$, $2i + 5$
- Not affine: ij, i^2, $i^2 + j^2$, $a[j]$

Affine Transformations

- Examples of affine functions of i, j: $i+j$, $i-j$, $i+1$, $2i+5$
- Not affine: ij, i^2, i^2+j^2, $a[j]$

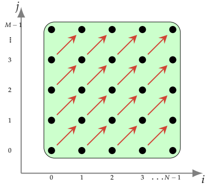


Figure: Iteration space

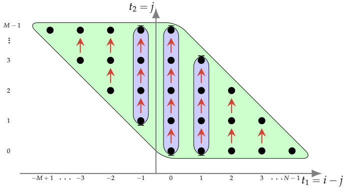


Figure: Transformed space

```
for (i = 0; i < N; i++){
    for (j = 0; j < M; j++){
          A[i+1][j+1] = f(A[i][j])
    }
}
```

```
 #pragma omp parallel for private(t2)
for (t1=-M+1; t1<=N-1; t1++) {
    for (t2=max(0,-t1); t2<=min(M-1,N-1-t1); t2++){
        A[t1+t2+1][t2+1] = f(A[t1+t2][t2]);
    }
}
```

- Transformation: $(i, j) \rightarrow (\mathbf{i-j}, \mathbf{j})$

AFFINE TRANSFORMATIONS

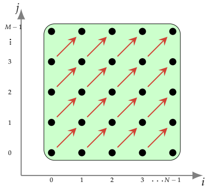


Figure: Iteration space

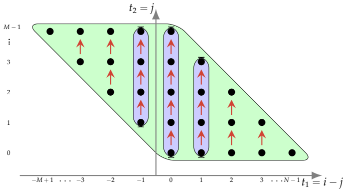


Figure: Transformed space

- Affine transformations are attractive because:
  - Preserve **collinearity** of points and **ratio of distances** between points
  - Code generation with affine transformations has thus been studied well (CLooG, ISL, OMEGA+)

AFFINE TRANSFORMATIONS

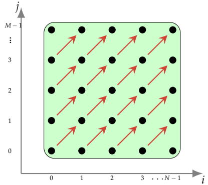


Figure: Iteration space

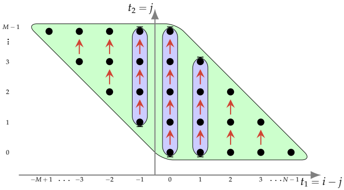


Figure: Transformed space

- Affine transformations are attractive because:
  - Preserve **collinearity** of points and **ratio of distances** between points
  - Code generation with affine transformations has thus been studied well (CLooG, ISL, OMEGA+)
  - Model a very rich class of loop re-orderings
  - Useful for several domains like dense linear algebra, stencils, image processing pipelines, Lattice Boltzmann Method

AFFINE TRANSFORMATIONS

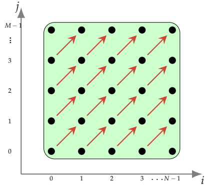


Figure: Iteration space

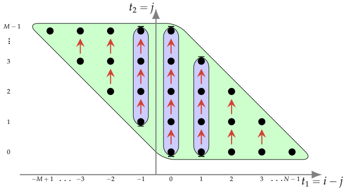


Figure: Transformed space

- Affine transformations can improve **parallelism** and **locality** (Feautrier 1992, Lengauer, Lim and Lam 1997, Griebl 2004, Pluto 2008)

The Pluto Algorithm

- Designed around 2008 [Bondhugula et al. CC 2008, PLDI 2008]
- **Finds good transformations to improves locality and parallelism**
- Extended in 2014-2015 (transformation coefficients need not be non-negative)

Finding Valid and Good Affine Transformations

(i, j)
(j, i)
(i+j, j)
(i-j, j)
(i, i+j)
(i+j, i-j)
. . .

Finding Valid and Good Affine Transformations

(i, j)
(j, i)
(i+j, j)
(i-j, j)
(i, i+j)
(i+j, i-j)
. . .

- One-to-one functions
- Validity: dependences should not be violated
- Coefficients: for $i - j$, the coefficients are 1,-1

Pluto Algorithm

- Optimization Problem: **Minimize dependence distance**

Pluto Algorithm

- Optimization Problem: **Minimize dependence distance**
- Constraints:
  - Tiling validity constraints
  - Dependence distance bounding constraints
  - Linear Independence constraints

Pluto Algorithm

- Optimization Problem: **Minimize dependence distance**
- Constraints:
  - Tiling validity constraints
  - Dependence distance bounding constraints
  - Linear Independence constraints

Tiling (blocking)

- Partition and execute iteration space in blocks

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

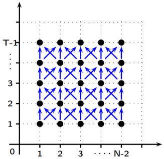


Figure: Iteration space

Tiling (blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

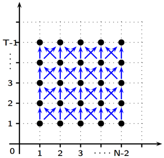


Figure: Iteration space

Tiling (blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

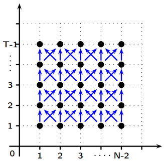


Figure: Iteration space

Tiling (blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

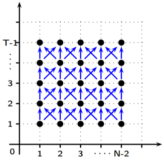


Figure: Iteration space

TILING (BLOCKING)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles
  - Sufficient condition: All dependence components should be non-negative

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

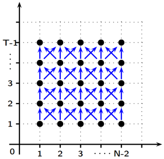


Figure: Iteration space

Tiling (Blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles
  - Sufficient condition: All dependence components should be non-negative
- Time tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

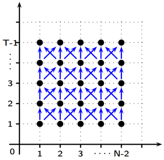


Figure: Iteration space

Tiling (blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles
  - Sufficient condition: All dependence components should be non-negative
- Time tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

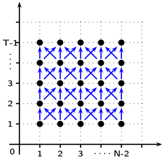


Figure: Iteration space

Tiling (blocking)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles
  - Sufficient condition: All dependence components should be non-negative
- Time tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

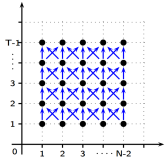


Figure: Iteration space

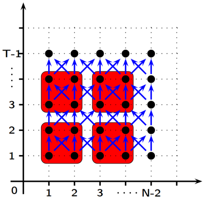


Figure: Invalid tiling

TILING (BLOCKING)

- Partition and execute iteration space in blocks
- Benefits - *cache locality* & *parallelism*
- Validity of tiling
  - No cycle between tiles
  - Sufficient condition: All dependence components should be non-negative
- Time tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    S(i,j)
```

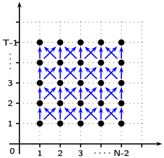


Figure: Iteration space

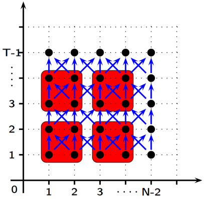


Figure: Invalid tiling

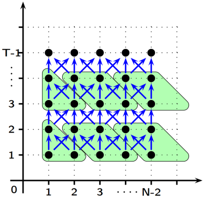


Figure: Valid tiling

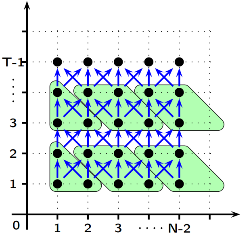


Figure: Parallelogram tiling

Concurrent Start and Diamond Tiling[1]

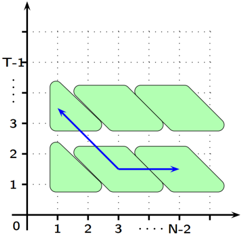


Figure: Pipelined start

CONCURRENT START AND DIAMOND TILING[1]

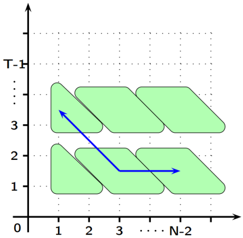


Figure: Pipelined start

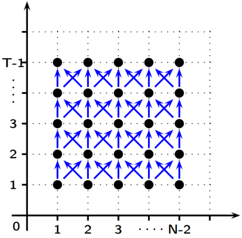


Figure: Original iteration space

Concurrent Start and Diamond Tiling[1]

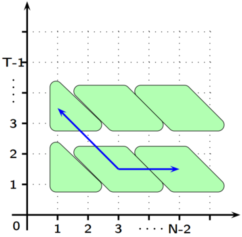


Figure: Pipelined start

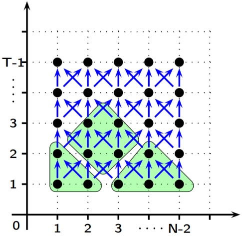


Figure: Group as diamonds

CONCURRENT START AND DIAMOND TILING[1]

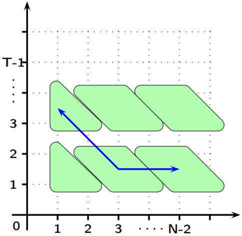


Figure: Pipelined start

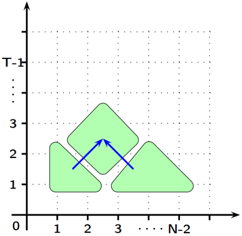


Figure: Concurrent start possible

Concurrent Start and Diamond Tiling[1]

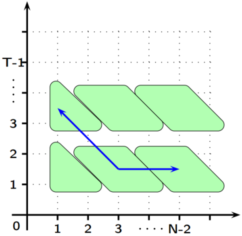


Figure: Pipelined start

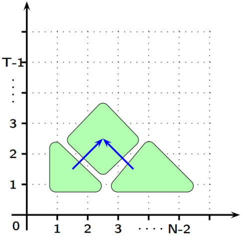


Figure: Concurrent start possible

- Diamond tiling
  - Face allowing concurrent should be strictly within the cone of the tiling hyperplanes
  - Eg: (1,0) is in the cone of (1,1) and (1,-1)

Classical Time Skewing vs Diamond Tiling

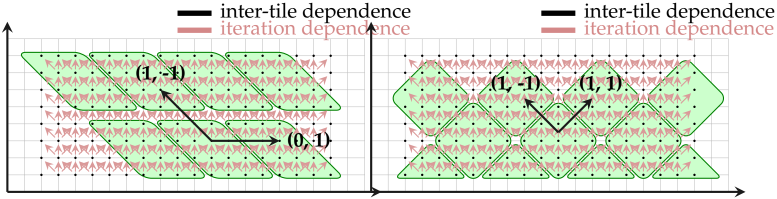


Figure: Two ways of tiling heat-1d: parallelogram & diamond

- Classical time skewing: $(t, i) \to (t, t + i)$
- Diamond tiling: $(t, i) \to (t + i, t - i)$

A sequence of transformations for 2-d Jacobi relaxations

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                  A[t%2][i][j+1], A[t%2][i][j-1], A[t%2][i][j]);
```

1. Enabling transformation for diamond tiling

$$T((t,i,j)) = (t+i, t-i, t+j).$$

A sequence of transformations for 2-d Jacobi relaxations

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                A[t%2][i][j+1], A[t%2][i][j-1], A[t%2][i][j]);
```

1. Enabling transformation for diamond tiling

$$T((t,i,j)) = (t+i, t-i, t+j).$$

2. Perform the actual tiling (in the transformed space)

$$T'((t,i,j)) = \left(\frac{t+i}{64}, \frac{t-i}{64}, \frac{t+j}{64}, t+i, t-i, t+j\right)$$

A sequence of transformations for 2-d Jacobi relaxations

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
             A[t%2][i][j+1], A[t%2][i][j-1], A[t%2][i][j]);
```

1. Enabling transformation for diamond tiling

$$T((t,i,j)) = (t+i, t-i, t+j).$$

2. Perform the actual tiling (in the transformed space)

$$T'((t,i,j)) = \left(\frac{t+i}{64}, \frac{t-i}{64}, \frac{t+j}{64}, t+i, t-i, t+j\right)$$

3. Create a wavefront of tiles

$$T''((t,i,j)) = \left(\frac{t+i}{64} + \frac{t-i}{64}, \frac{t-i}{64}, \frac{t+j}{64}, t, t+i, t+j\right)$$

Some examples of how Pluto can be used

- Optimize Jacobi and other relaxations via time tiling

Some examples of how Pluto can be used

- Optimize Jacobi and other relaxations via time tiling
- Optimize pre-smoothing steps at various levels of Geometric Multigrid method

Some examples of how Pluto can be used

- Optimize Jacobi and other relaxations via time tiling
- Optimize pre-smoothing steps at various levels of Geometric Multigrid method
- Optimize Lattice Boltzmann Method computations

Using Pluto: Recommendations

Web: http://pluto-compiler.sf.net

- Use git version
- Use 'pet' branch of git version
- Preferable: use Intel's C/C++ compiler (14.0 or higher) to compile generated code

Outline

Poisson's Equation

Poisson's equation:

$$\nabla^2 u = f.$$

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} u_h = f_h$$

- We are solving $y = Ax$
- What about A^{-1}?

Geometric Multigrid Method

- Use a hierarchical structure – a multi-scale representation of the grid
- Perform pre-smoothing at a finer level
- Restrict the error to a coarser grid
- Solve for the error at a coarser level (recursion)
- Interpolate the error to the finer level

- Run multiple iterations of the above

Pluto can be used to optimize the pre-smoothing or post-smoothing steps readily

Hierarchical Mesh Structure

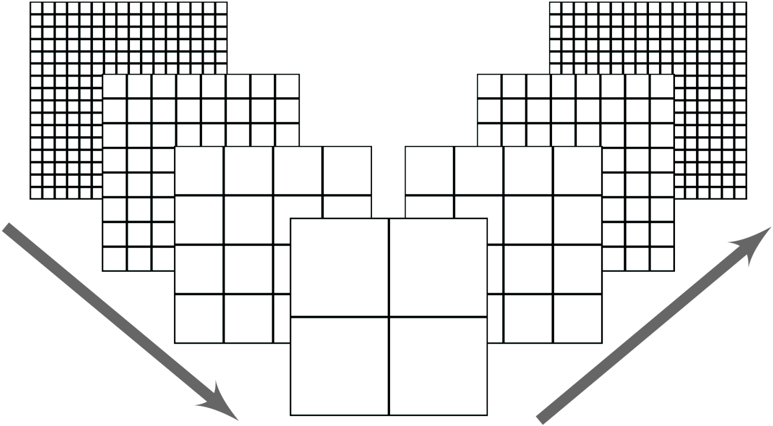


Figure: Hierarchical mesh structure for Multigrid levels

Mulitigrid V-cycle: Algorithm

Input : v^h, f^h

Relax v^h for n_1 iterations: $\nu^h \leftarrow (1 - \omega D^{-1} A^h) v^h + \omega D^{-1} f^h$
// pre-smoothing

if coarsest level **then**

  Relax v^h for n_2 iterations *// coarse smoothing*

$r^h \leftarrow f^h - A^h v^h$ *// residual*

$r^{2h} \leftarrow I_h^{2h} r^h$ *// restriction*

$e^{2h} \leftarrow 0$

$e^{2h} \leftarrow V\text{-}cycle^{2h}(e^{2h}, r^{2h})$

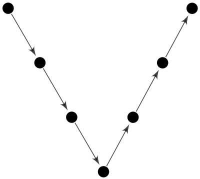
$e^h \leftarrow I_{2h}^h e^{2h}$ *// interploation*

$v^h \leftarrow v^h + e^h$ *// correction*

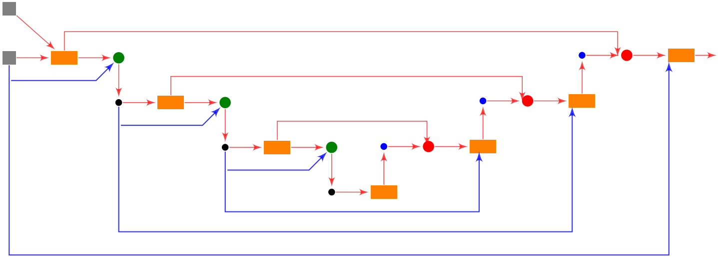
Relax v^h for n_3 iterations *// post smoothing*

return v^h

MULITIGRID V-CYCLE

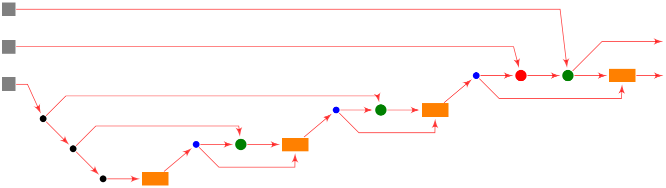


(a) V-cycle



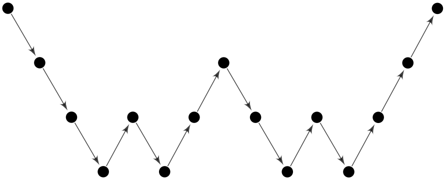
(b) V-cycle: complete DAG

NAS MG V-CYCLE

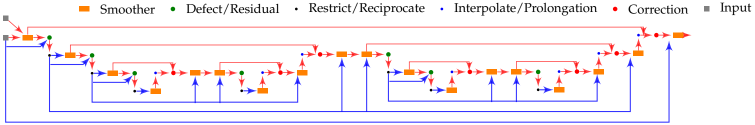


(c) NAS-PB MG V-cycle

MULTIGRID W-CYCLE



(d) W-cycle



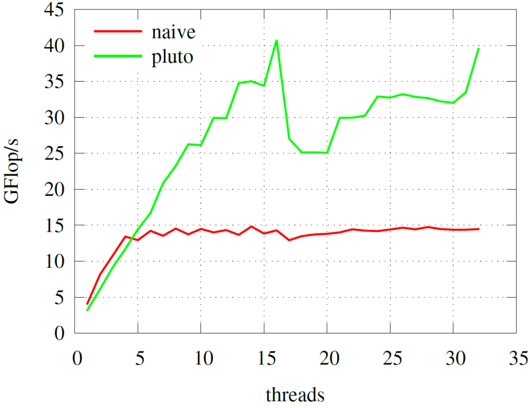
(e) W-cycle: complete DAG

Figure: DAG representation of (a) V-cycle and (b) W-cycle

Strongly recommend reading:

- P. Ghysels and W. Vanroose, Modeling the performance of geometric multigrid on many-core computer architectures, SIAM J. Scientific Computing (2015).
- W. Vanroose, P. Ghysels, D. Roose, and K.Meerbergen, Hiding global communication latency and increasing the arithmetic intensity in extreme-scale Krylov solvers, Position Paper at DOE/ASCR workshop on Applied Mathematics Research for Exascale Computing. Aug 2013.

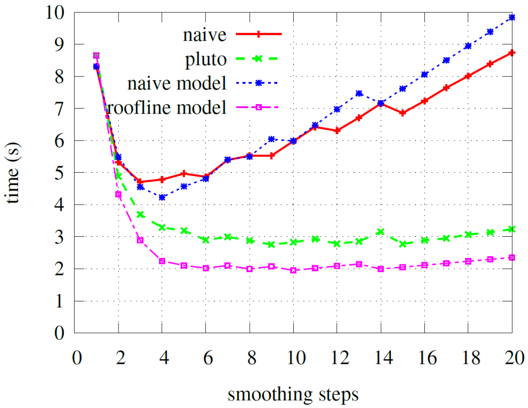
GMG: Smoother Scaling



Scalability of 10 iterations of the Jacobi smoother on an 8000^2 domain on a 16-core Intel Sandy Bridge

Source: Ghysels (LBNL) and Vanroose (University of Antwerp) SIAM J. Scientific Computing 2015

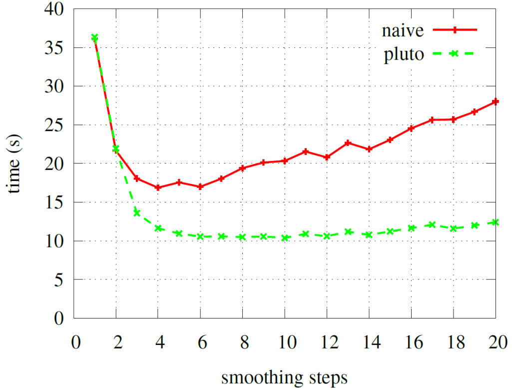
GMG: EXECUTION TIME (2-D)



Timings for a full solve on a 8191^2 domain using V -cycles with a relative stopping tolerance 10^{-12}

Source: Ghysels and Vanroose 2015

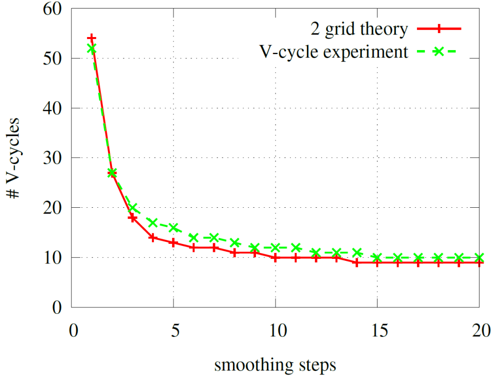
GMG: Execution time (3-d)



Timings for a full solve on a 511^3 domain using V -cycles with a relative stopping tolerance 10^{-12} on a dual socket Sandy Bridge machine for a 3D domain

Source: Ghysels and Vanroose 2015

GMG: CONVERGENCE FOR SMOOTHING STEPS



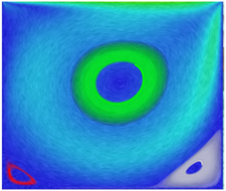
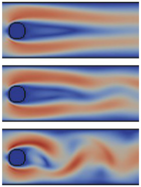
The corresponding number of V-cycles required to reach a 10^{-12} relative stopping criterion for both two-grid and multigrid.

Source: Ghysels and Vanroose 2015

Outline

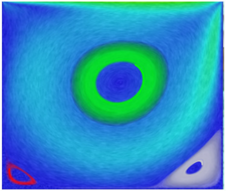
1. Pluto/Pluto+
   - Affine Transformations
   - Tiling
2. Case Studies
   - Solving Partial Differential Equations
   - Lattice Boltzmann Method
   - Image Processing Pipelines
3. Conclusions

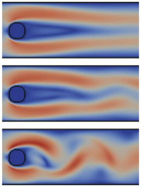
LBM: Introduction

- Lattice–Boltzmann method (LBM) is used for simulation of complex fluid flows in Computational Fluid Dynamics (CFD)

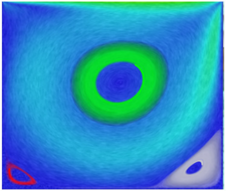
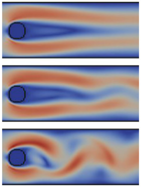
LBM: Introduction





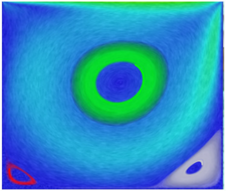
- Lattice–Boltzmann method (LBM) is used for simulation of complex fluid flows in Computational Fluid Dynamics (CFD)
- The simplicity of formulation and its versatility explain the rapid expansion of LBM to applications in complex and multiscale flows

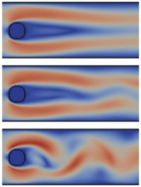
LBM: Introduction

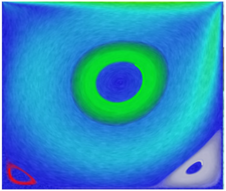
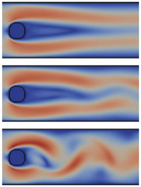
- Lattice–Boltzmann method (LBM) is used for simulation of complex fluid flows in Computational Fluid Dynamics (CFD)
- The simplicity of formulation and its versatility explain the rapid expansion of LBM to applications in complex and multiscale flows
- Particularly suited for parallel and high performance implementations

LBM: Introduction





- Lattice–Boltzmann method (LBM) is used for simulation of complex fluid flows in Computational Fluid Dynamics (CFD)
- The simplicity of formulation and its versatility explain the rapid expansion of LBM to applications in complex and multiscale flows
- Particularly suited for parallel and high performance implementations
- In spite of tremendous advances in its application, several fundamental opportunities for optimization remain

- Lattice–Boltzmann method (LBM) is used for simulation of complex fluid flows in Computational Fluid Dynamics (CFD)
- The simplicity of formulation and its versatility explain the rapid expansion of LBM to applications in complex and multiscale flows
- Particularly suited for parallel and high performance implementations
- In spite of tremendous advances in its application, several fundamental opportunities for optimization remain
- We explore one such opportunity through this work

Lattice-Boltzmann Method

- Fluid flows are modelled as hypothetical particles
  - moving in a lattice domain (discretized space)
  - with different lattice velocities (discretized momentum)
  - over different time steps (discretized time)

Lattice-Boltzmann Method

- Fluid flows are modelled as hypothetical particles
  - moving in a lattice domain (discretized space)
  - with different lattice velocities (discretized momentum)
  - over different time steps (discretized time)
- Solves the discrete Boltzmann equation for the particle distribution function (a probability density function)

LBM - Lattice Arrangements

- Lattice arrangements are represented as $DmQn$
  - m is the space dimensionality of the lattice
  - n is the number of PDFs (or speeds) involved

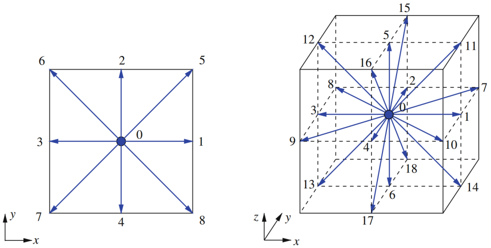


Figure: $D2Q9$ (left) & $D3Q19$ (right) lattice arrangements

LBM - Lattice Arrangements

- Lattice arrangements are represented as $DmQn$
  - m is the space dimensionality of the lattice
  - n is the number of PDFs (or speeds) involved
- Choice of lattice affects precision and duration of simulation

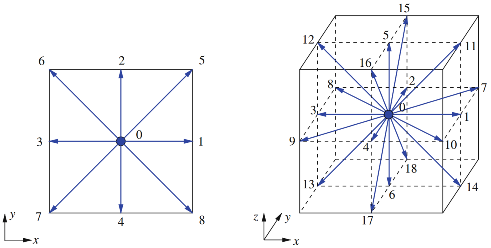


Figure: $D2Q9$ (left) & $D3Q19$ (right) lattice arrangements

- The discretized form of Lattice–Boltzmann Equation forms the basis of all LBM models

$$f_i(\mathbf{x} + \mathbf{c_i}\Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)),\ i = 1, \ldots n. \quad (1)$$

- The discretized form of Lattice–Boltzmann Equation forms the basis of all LBM models

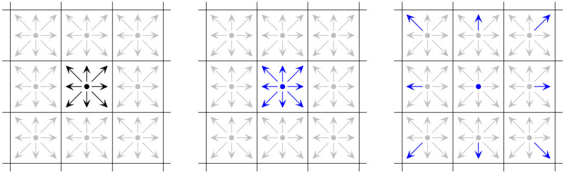
$$f_i(\mathbf{x} + \mathbf{c_i}\Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)),\ i = 1, \ldots n. \quad (1)$$

- Eqn. 1 is solved in two steps, the **collision** step (Eqn. 2) & the **advection** step (Eqn. 3)

$$f_i^*(\mathbf{x}, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)) \quad (2)$$

$$f_i(\mathbf{x} + \mathbf{c_i}\Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t + \Delta t) \quad (3)$$

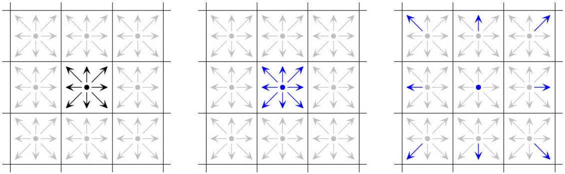
LBM - Implementation Strategies



(a) Pre-collision (b) Post-collision (c) Post-propagation

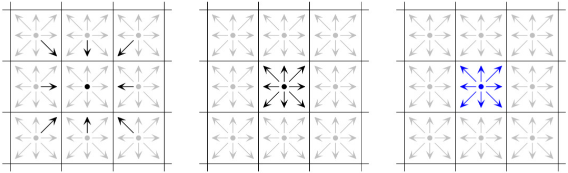
Figure: Push Scheme

LBM - Implementation Strategies



(a) Pre-collision (b) Post-collision (c) Post-propagation

Figure: Push Scheme



(a) Initial state (b) Pre-collision (c) Post-collision

Figure: Pull Scheme

Is it possible to optimize LBM using time tiling?

Time Tiling LBM Computations

- LBM can be written using storage for either one grid or two grids

Time Tiling LBM Computations

- LBM can be written using storage for either one grid or two grids
- One grid ⇒ Separate collision and advection
- Fused Collision + Advection ⇒ Two grids

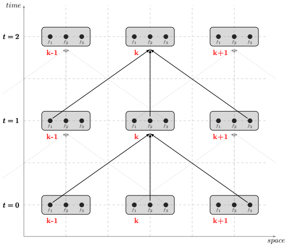


Figure: 1D LBM with single grid

1

Time Tiling LBM Computations

- LBM can be written using storage for either one grid or two grids
- One grid ⇒ Separate collision and advection
- Fused Collision + Advection ⇒ Two grids

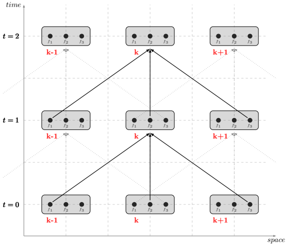


Figure: 1D LBM with single grid

- **Not possible to "*time tile*" LBM with single grid**

Time Tiling LBM Computations

- LBM can be written using storage for either one grid or two grids
- One grid ⇒ Separate collision and advection
- Fused Collision + Advection ⇒ Two grids

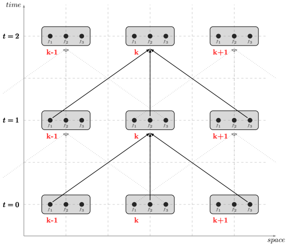


Figure: 1D LBM with single grid

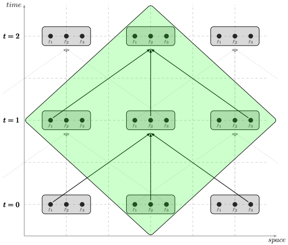


Figure: Pull scheme on a 1D LBM

- **Not possible to "*time tile*" LBM with single grid**

1

1

Time Tiling LBM Computations

- LBM can be written using storage for either one grid or two grids
- One grid ⇒ Separate collision and advection
- Fused Collision + Advection ⇒ Two grids

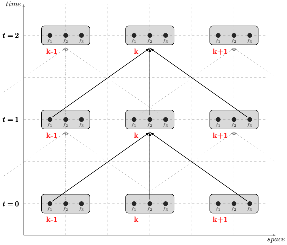


Figure: 1D LBM with single grid

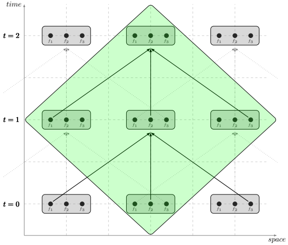


Figure: Pull scheme on a 1D LBM

- **Not possible to "*time tile*" LBM with single grid**
- **Time tiling is possible with two grids**

1

1

msLBM - LBM Optimization Framework

- We utilize a fused version of the LBM kernel
  - No explicit *advection* phase
  - 2 data grids with a pull scheme to read data
  - Array-of-Structures (*AoS*) layout for data

Input to Polyhedral Tiler

```
#pragma scop
for (t = 0; t < _nTimesteps; t++)
   for (y = 2; y < _nY; y++)
      for (x = 1; x < _nX; x++)
         lbm_kernel(grid[t % 2][y][x][C],
            grid[t % 2][y - 1][x + 0][N],
            grid[t % 2][y + 1][x + 0][S],
            grid[t % 2][y + 0][x - 1][E],
            grid[t % 2][y + 0][x + 1][W],
            grid[t % 2][y - 1][x - 1][NE],
            grid[t % 2][y - 1][x + 1][NW],
            grid[t % 2][y + 1][x - 1][SE],
            grid[t % 2][y + 1][x + 1][SW],

            &grid[(t + 1) % 2][y][x][C],
            &grid[(t + 1) % 2][y][x][N],
            &grid[(t + 1) % 2][y][x][S],
            &grid[(t + 1) % 2][y][x][E],
            &grid[(t + 1) % 2][y][x][W],
            &grid[(t + 1) % 2][y][x][NE],
            &grid[(t + 1) % 2][y][x][NW],
            &grid[(t + 1) % 2][y][x][SE],
            &grid[(t + 1) % 2][y][x][SW], t, y, x);
```

- Use the **PET polyhedral frontend [Verdoolaege and Grosser 2012]**: the LBM collision is treated as a blackbox (abstracted as a single function)
- Dependence structure is now similar to “toy time-iterated stencils”
- **All time tiling strategies can now be applied!**

Experimental Setup

Intel Xeon E5-2680 (SandyBridge)	
Clock	2.7 GHz
Cores / socket	8
Total cores	16
L1 cache / core	32 KB
L2 cache / core	512 KB
L3 cache / socket	20 MB
Peak GFLOPs	172.8
Compiler	Intel C compiler (icc) 14.0.1
Compiler flags	-O3 -xHost -ipo -fno-alias -fno-fnalias -restrict -fp-model precise -fast-transcendentals
Linux kernel	3.8.0-38

Table: Architecture details

- We compare the performance of our framework on 7 benchmarks against
  - Palabos - an open-source CFD solver based on LBM
  - Compiler auto-parallelization [*icc-auto-par*]
  - Naive manual parallelization using OpenMP [*icc-omp-par*]

BENCHMARKS

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`

BENCHMARKS

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`
- SPEC LBM [*470.lbm* from SPEC2006] - `d3q19`

Benchmarks

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`
- SPEC LBM [*470.lbm* from SPEC2006] - `d3q19`
- Poiseuille Flow - `d2q9`

BENCHMARKS

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`
- SPEC LBM [*470.lbm* from SPEC2006] - `d3q19`
- Poiseuille Flow - `d2q9`
- Flow Past Cylinder - `d2q9`

Benchmarks

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`
- SPEC LBM [*470.lbm* from SPEC2006] - `d3q19`
- Poiseuille Flow - `d2q9`
- Flow Past Cylinder - `d2q9`
- MRT - GLBM - `d2q9`

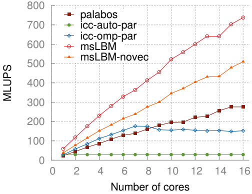
BENCHMARKS

- Lid Driven Cavity - `d2q9`, `d3q19` and `d3q27`
- SPEC LBM [*470.lbm* from SPEC2006] - `d3q19`
- Poiseuille Flow - `d2q9`
- Flow Past Cylinder - `d2q9`
- MRT - GLBM - `d2q9`

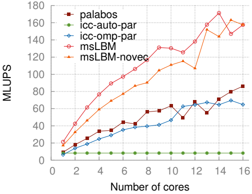
Performance Metrics

- MLUPS - Million Lattice site Updates Per Second
- MEUPS - Million Element Updates Per Second

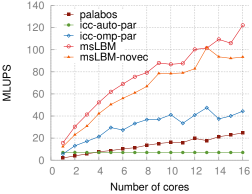
Performance - LDC



(a) `ldc-d2q9`

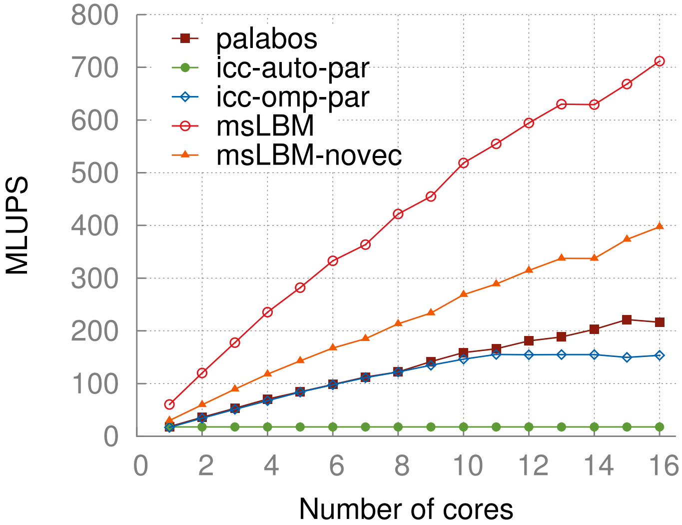


(b) `ldc-d3q19`



(c) `ldc-d3q27`

Performance - MRT (d2q9)



Roofline Performance Model - Contd.

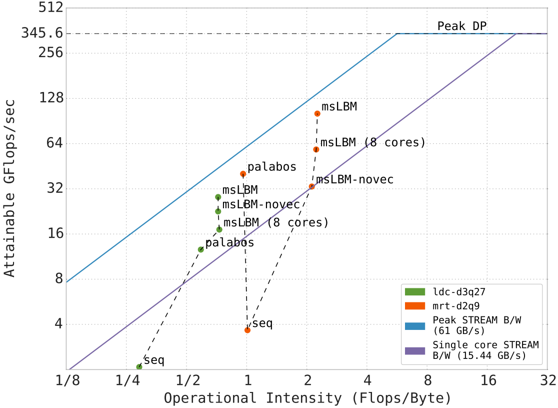


Figure: Roofline model for `mrt-d2q9` & `ldc-d3q27`

ROOFLINE PERFORMANCE MODEL - CONTD.

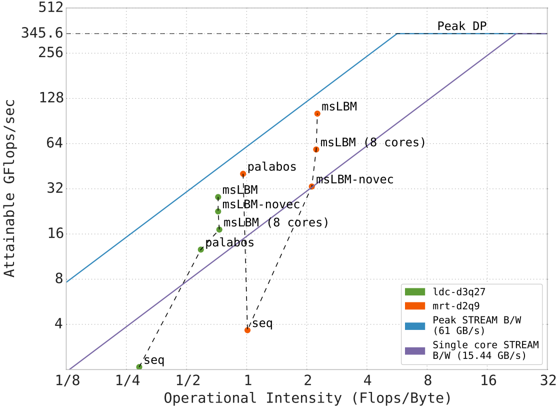


Figure: Roofline model for `mrt-d2q9` & `ldc-d3q27`

- `msLBM` obtains further improvement over Palabos in both operational intensity and peak achievable performance

Outline

1. Pluto/Pluto+
   - Affine Transformations
   - Tiling
2. Case Studies
   - Solving Partial Differential Equations
   - Lattice Boltzmann Method
   - Image Processing Pipelines
3. Conclusions

PolyMage

http://mcl.csa.iisc.ernet.in/polymage.html

A DSL and Compiler for Automatic Parallelization and Optimization of Image Processing Pipelines

Graphs of interconnected processing stages

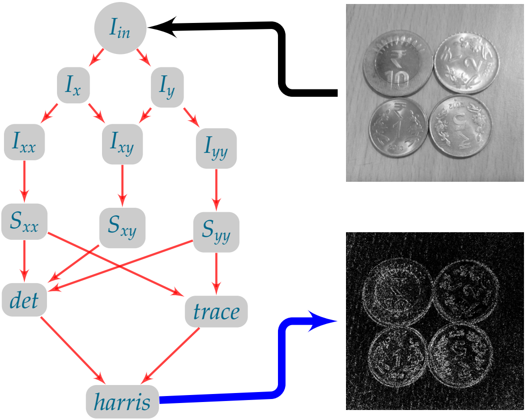
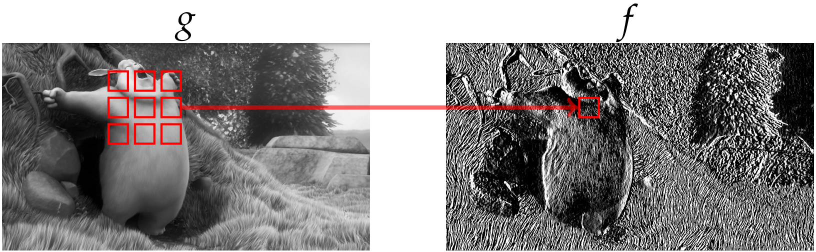


Figure: **Harris corner detection**

Point-wise

$$f(x,y) = w_r \cdot g(x,y,{\color{red}\bullet}) + w_g \cdot g(x,y,{\color{green}\bullet}) + w_b \cdot g(x,y,{\color{blue}\bullet})$$

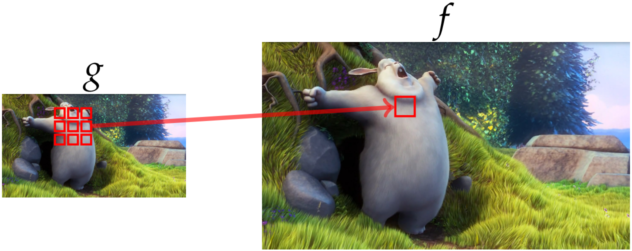


Stencil

$$f(x, y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x + \sigma_x, y + \sigma_y) \cdot w(\sigma_x, \sigma_y)$$

Downsample

$$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(2x+\sigma_x, 2y+\sigma_y) \cdot w(\sigma_x, \sigma_y)$$



Upsample

$$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g((x+\sigma_x)/2, (y+\sigma_y)/2) \cdot w(\sigma_x, \sigma_y, x, y)$$

Example: Pyramid Blending pipeline

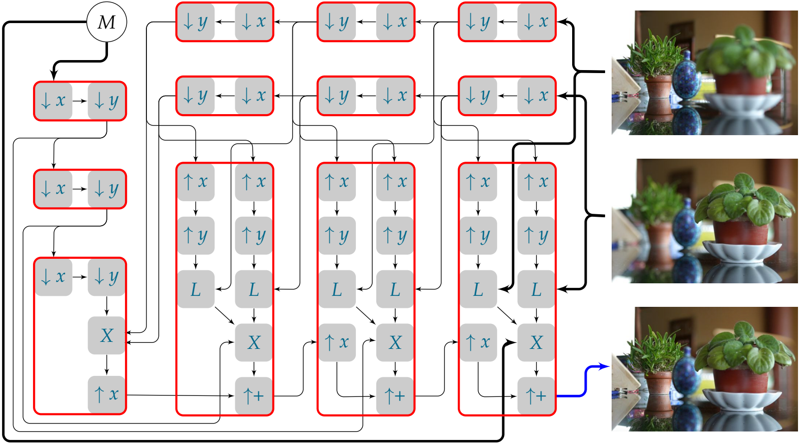


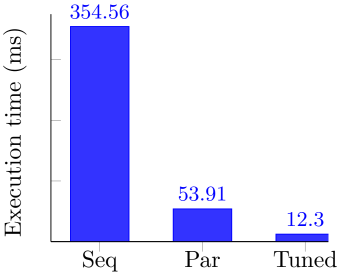
Image courtesy: Kyros Kutulakos

Where are Image Processing Pipelines used?

- On images uploaded to social networks like Facebook, Google+
- On all camera-enabled devices
- Everyday workloads from data center to mobile device scales
- **Computational photography, computer vision, medical imaging,** ...

Google+ Auto Enhance

Naive vs Optimized Implementation

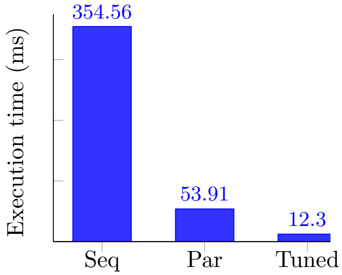


Harris corner detection (16 cores)

- Naive implementation in C
- Naive parallelization – **7**×
  OpenMP, Vector pragmas (icc)
- Manual optimization – **29**×
  Locality, Parallelism, Vector intrinsics

Manually optimizing pipelines is hard

NAIVE VS OPTIMIZED IMPLEMENTATION



Harris corner detection (16 cores)

- Naive implementation in C
- Naive parallelization – **7**×
  OpenMP, Vector pragmas (icc)
- Manual optimization – **29**×
  Locality, Parallelism, Vector intrinsics

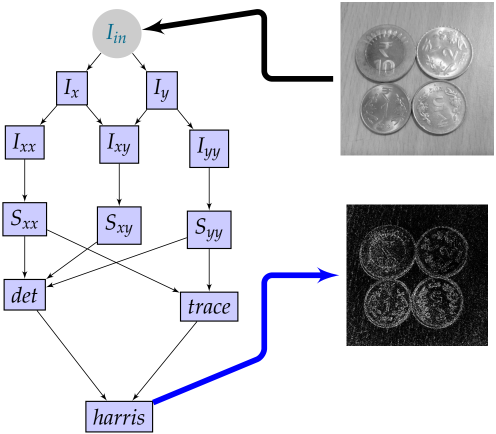
Goal: Performance levels of manual tuning
Without the pain

Our Approach: PolyMage

- **High-level language (DSL embedded in Python)**
  - Allow expressing common patterns intuitively
  - Enables compiler analysis and optimization
- **Automatic Optimizing Code Generator**
  - Uses domain-specific cost models to apply complex combinations of scaling, alignment, **tiling** and **fusion** to optimize for **parallelism** and **locality**

Harris Corner Detection

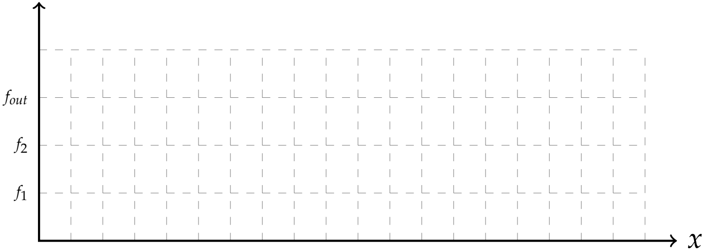
```
R, C = Parameter(Int), Parameter(Int) (∗\label{param}∗)
I = Image(Float, [R+2, C+2]) (∗\label{image}∗)

x, y = Variable(), Variable() (∗\label{vars}∗)
row, col = Interval(0,R+1,1), Interval(0,C+1,1)(∗\label{intervals}∗)

c = Condition(x,’>=’,1) & Condition(x,’<=’,R) &(∗\label{cond1}∗)
    Condition(y,’>=’,1) & Condition(y,’<=’,C)

cb = Condition(x,’>=’,2) & Condition(x,’<=’,R-1) &(∗\label{cond2}∗)
     Condition(y,’>=’,2) & Condition(y,’<=’,C-1)

Iy = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f1}∗)
Iy.defn = [ Case(c, Stencil(I(x,y), 1.0/12,(∗\label{d1}∗)
                    [[-1, -2, -1],
                     [ 0, 0, 0],
                     [ 1, 2, 1]]) ]

Ix = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f2}∗)
Ix.defn = [ Case(c, Stencil(I(x,y), 1.0/12,(∗\label{d2}∗)
                    [[-1, 0, 1],
                     [-2, 0, 2],
                     [-1, 0, 1]]) ]

Ixx = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f3}∗)
Ixx.defn = [ Case(c, Ix(x,y) ∗ Ix(x,y)) ](∗\label{d3}∗)

Iyy = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f4}∗)
Iyy.defn = [ Case(c, Iy(x,y) ∗ Iy(x,y)) ](∗\label{d4}∗)

Ixy = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f5}∗)
Ixy.defn = [ Case(c, Ix(x,y) ∗ Iy(x,y)) ](∗\label{d5}∗)

Sxx = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f6}∗)
Syy = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f7}∗)
Sxy = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f8}∗)
for pair in [(Sxx, Ixx), (Syy, Iyy), (Sxy, Ixy)]:(∗\label{meta}∗)
    pair[0].defn = [ Case(cb, Stencil(pair[1], 1,(∗\label{d6}∗)
                              [[1, 1, 1],
                               [1, 1, 1],
                               [1, 1, 1]]) ]

det = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f9}∗)
d = Sxx(x,y) ∗ Syy(x,y) - Sxy(x,y) ∗ Sxy(x,y)
det.defn = [ Case(cb, d) ] (∗\label{d7}∗)

trace = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f10}∗)
trace.defn = [ Case(cb, Sxx(x,y) + Syy(x,y)) ](∗\label{d8}∗)

harris = Function(varDom = ([x,y],[row,col]),Float)(∗\label{f11}∗)
coarsity = det(x,y) - .04 ∗ trace(x,y) ∗ trace(x,y)(∗\label{d9}∗)
harris.defn = [ Case(cb, coarsity) ]
```



Our Approach: PolyMage

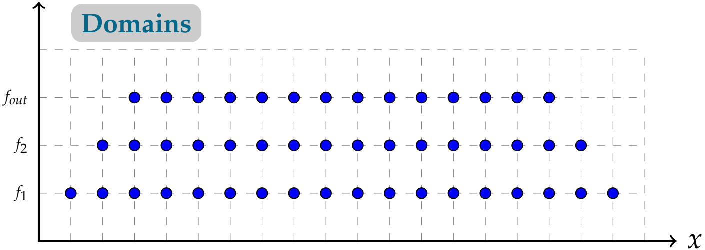
- **High-level language (DSL embedded in Python)**
  - Allow expressing common patterns intuitively
  - Enables compiler analysis and optimization
- **Automatic Optimizing Code Generator**
  - Uses domain-specific cost models to apply complex combinations of scaling, alignment, **tiling** and **fusion** to optimize for **parallelism** and **locality**

Polyhedral Representation



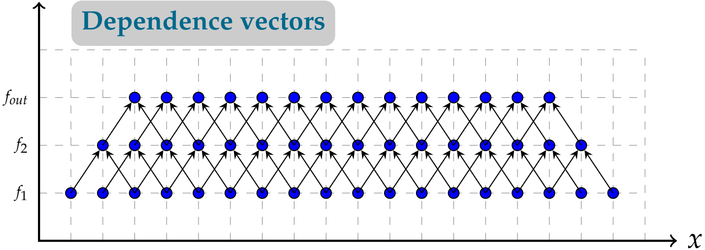

```
x = Variable()
f_in = Image(Float, [18])
f_1 = Function(varDom = ([x], [Interval(0, 17, 1)]), Float)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(varDom = ([x], [Interval(1, 16, 1)]), Float)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(varDom = ([x], [Interval(2, 15, 1)]), Float)
f_out.defn = [ f_2(x-1) + f_2(x+1) ]
```

Polyhedral Representation



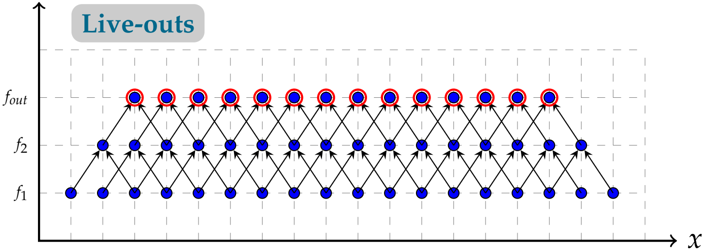

```
x = Variable()
f_in = Image(Float, [18])
f_1 = Function(varDom = ([x], [Interval(0, 17, 1)]), Float)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(varDom = ([x], [Interval(1, 16, 1)]), Float)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(varDom = ([x], [Interval(2, 15, 1)]), Float)
f_out.defn = [ f_2(x-1) + f_2(x+1) ]
```

Polyhedral Representation



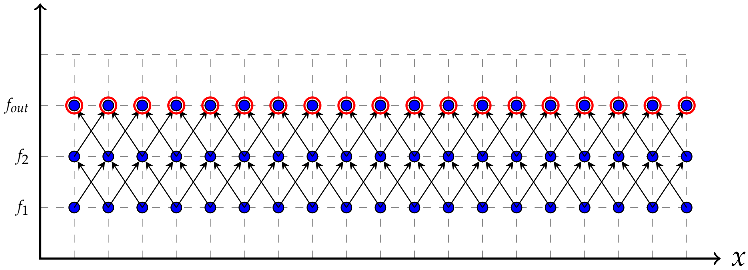
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1), (1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1), (1,-1)$
$f_1(x) = f_{in}(x)$	

Polyhedral Representation



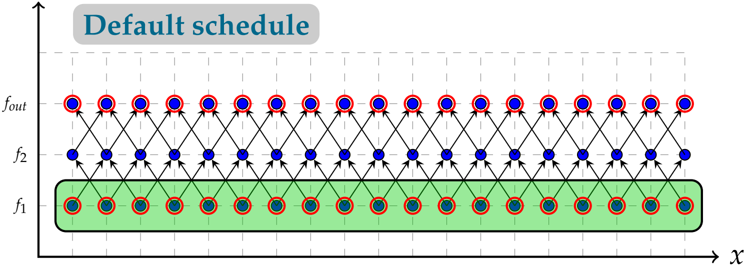
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1), (1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1), (1,-1)$
$f_1(x) = f_{in}(x)$	

Scheduling Criteria



Parallelism **Locality** **Storage**

Scheduling Criteria

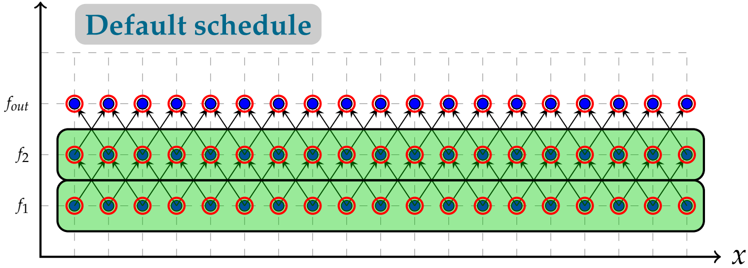


Parallelism

Locality

Storage

Scheduling Criteria

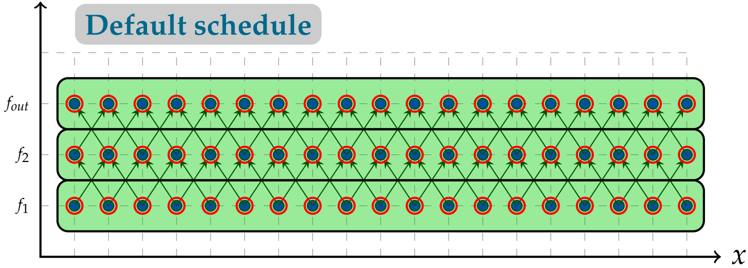


Parallelism

Locality

Storage

Scheduling Criteria

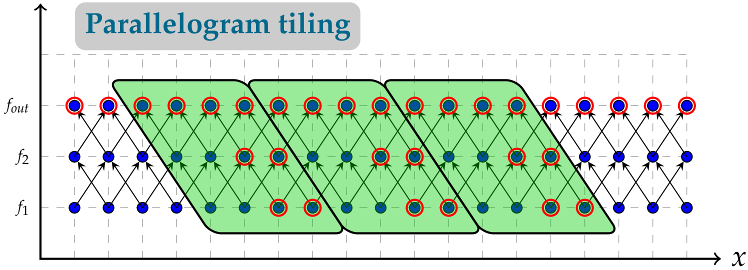


Parallelism

Locality

Storage

Scheduling Criteria

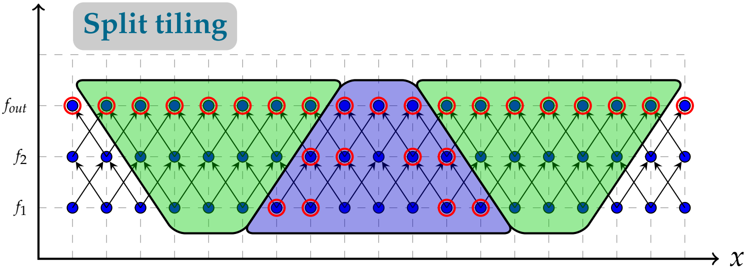


Parallelism

Locality

Storage

Scheduling Criteria

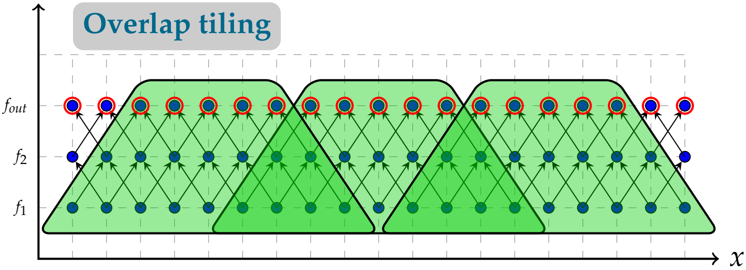


Parallelism

Locality

Storage

Scheduling Criteria



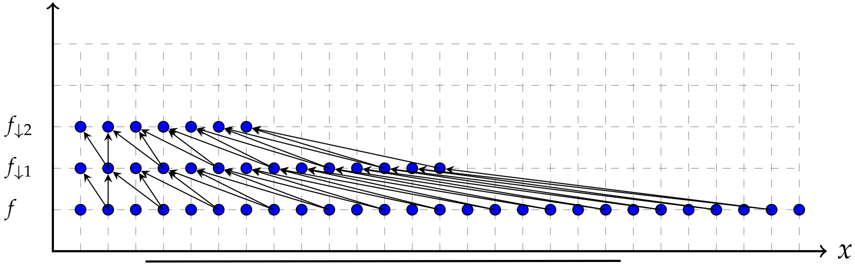
Parallelism

Locality

Storage

Re-computation

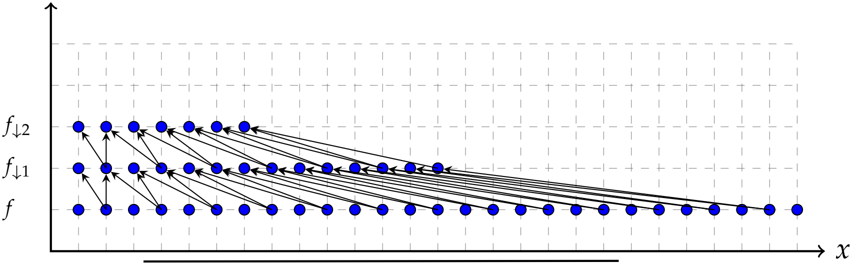
Overlapped Tiling for Heterogeneous Functions



Function	Schedule
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

- **Prior approaches for overlapped tiling only consider homogeneous time-iterated stencils**

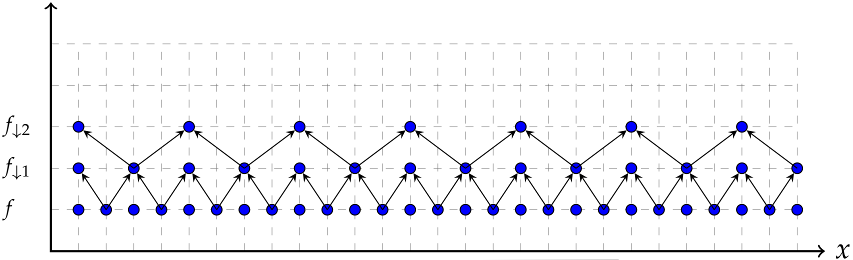
Overlapped Tiling for Heterogeneous Functions



Function	Schedule
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

- **Cannot have a fixed tile shape when dependence vectors are non-constant**

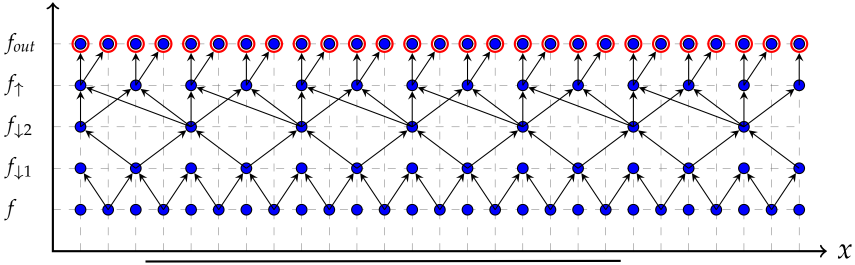
Overlapped Tiling for Heterogeneous Functions



Function	**Schedule**
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, 4x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, 2x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

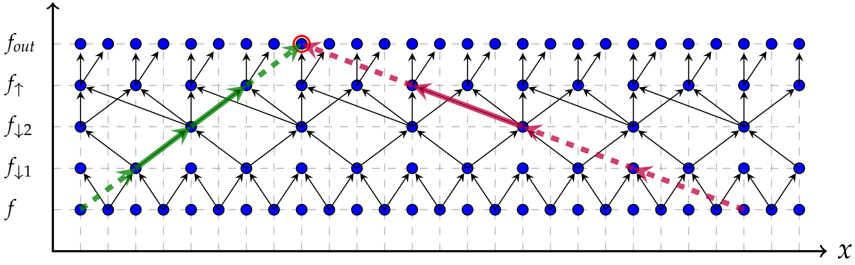
- **Scaling and aligning the schedules**

Overlapped Tiling for Heterogeneous Functions



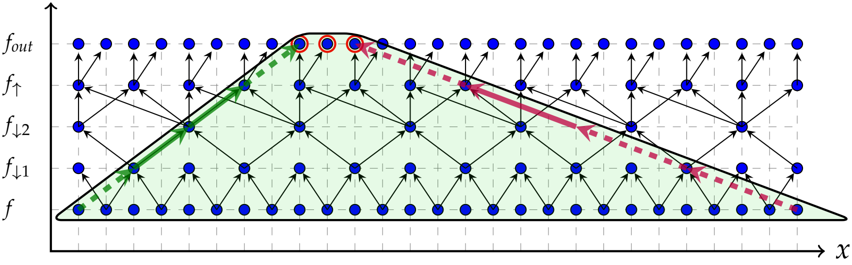
Function	Schedule
$f_{out}(x) = f_{\uparrow}(x/2)$	$(x) \rightarrow (4, x)$
$f_{\uparrow}(x) = f_{\downarrow 2}(x/2) \cdot f_{\downarrow 2}(x/2+1)$	$(x) \rightarrow (3, 2x)$
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, 4x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, 2x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

Overlapped Tiling for Heterogeneous Functions



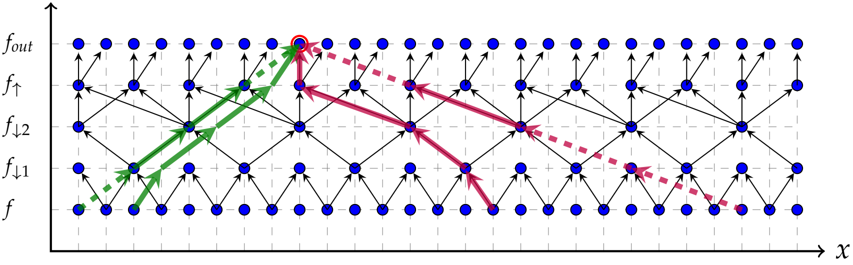
- **Determining tile shape**
  Conservative vs precise bounding faces

Overlapped Tiling for Heterogeneous Functions



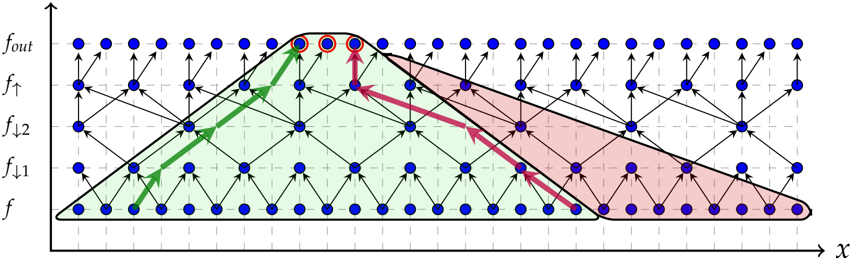
- **Determining tile shape**
  Conservative vs precise bounding faces

Overlapped Tiling for Heterogeneous Functions



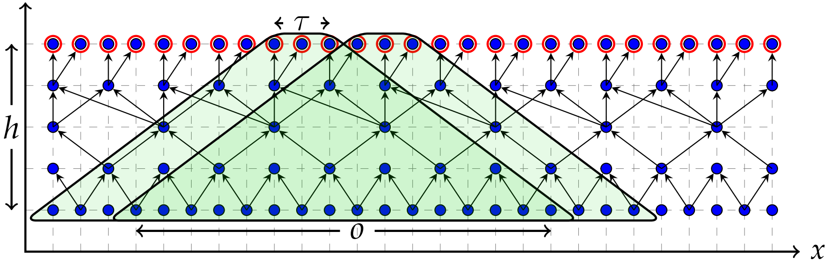
- **Determining tile shape**
  Conservative vs precise bounding faces

Overlapped Tiling for Heterogeneous Functions



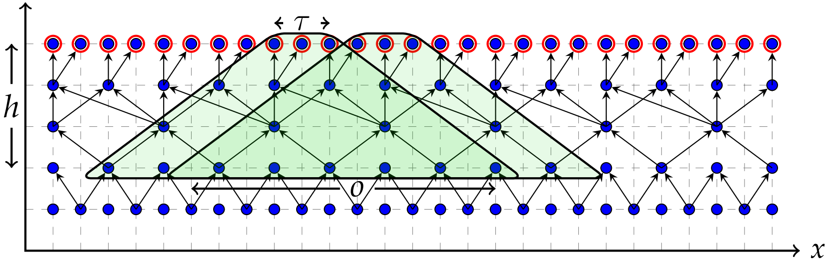
- **Significant reduction in redundant computation**

Overlapped Tiling for Heterogeneous Functions



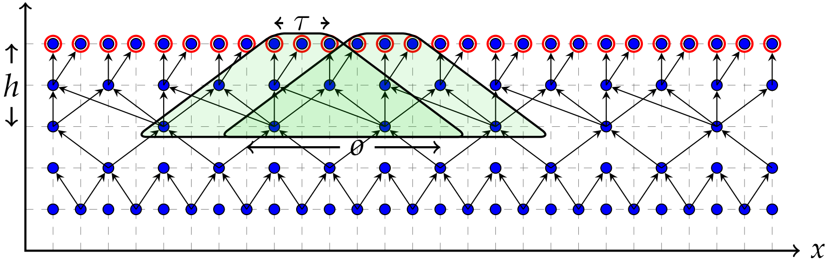
- **Tile size** τ**, Overlap** O**, Height** h
  Trade-off between fusion height and overlap

Overlapped Tiling for Heterogeneous Functions



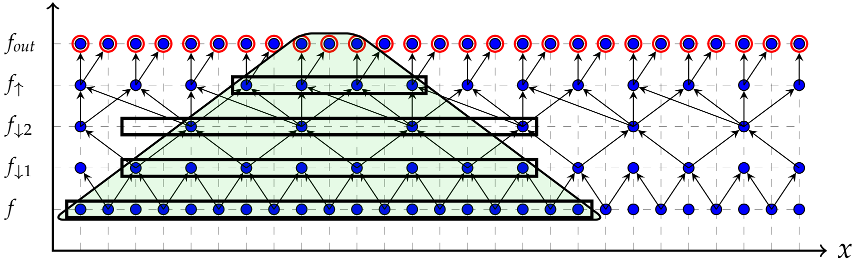
- **Tile size τ, Overlap O, Height h**
  Trade-off between fusion height and overlap

Overlapped Tiling for Heterogeneous Functions



- **Tile size τ, Overlap O, Height h**
  Trade-off between fusion height and overlap

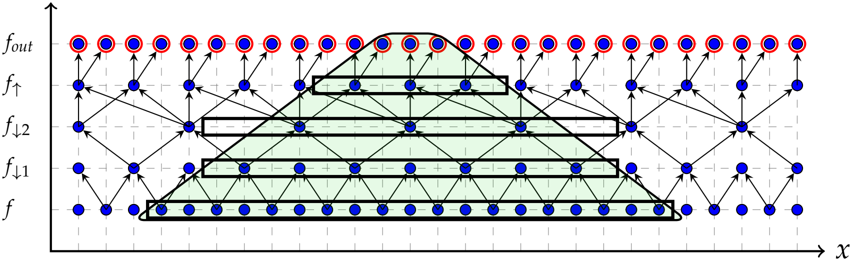
Overlapped Tiling for Heterogeneous Functions



Scratch pads

- Reduction in intermediate storage
- Better locality and reuse
- Privatized for each thread

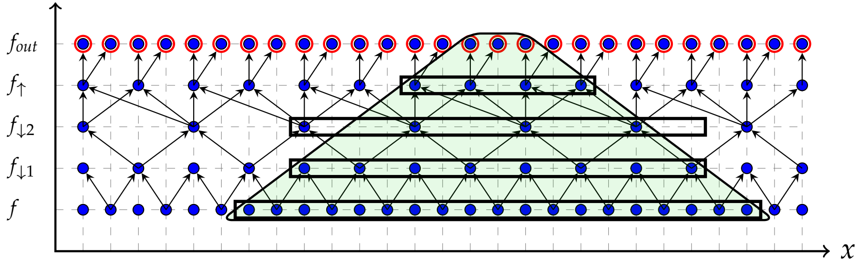
Overlapped Tiling for Heterogeneous Functions



Scratch pads

- Reduction in intermediate storage
- Better locality and reuse
- Privatized for each thread

Overlapped Tiling for Heterogeneous Functions



Scratch pads

- Reduction in intermediate storage
- Better locality and reuse
- Privatized for each thread

Benchmarks

Seven representative benchmarks of varying structure and complexity

Benchmark	Stages	Lines	Image size
Unsharp Mask	4	16	2048×2048×3
Bilateral Grid	7	43	2560×1536
Harris Corner	11	43	6400×6400
Camera Pipeline	32	86	2528×1920
Pyramid Blending	44	71	2048×2048×3
Multiscale Interpolate	49	41	2560×1536×3
Local Laplacian	99	107	2560×1536×3

Outline

1. Pluto/Pluto+
   - Affine Transformations
   - Tiling
2. Case Studies
   - Solving Partial Differential Equations
   - Lattice Boltzmann Method
   - Image Processing Pipelines
3. Conclusions

Conclusions

- Interesting to see how numerical techniques can be chosen and designed around modern parallel architectures and optimization infrastructure

Acknowledgements

- *Ravi Teja Mullapudi, Vinay Vasista, Irshad Pananilath, Aravind Acharya, Vinayaka Bandishti* (**students at IISc**)
- **Google Research**
- *Anand Deshpande, Aniruddha Shet, Bharat Kaul, and Sunil Sherlekar* from **Intel Labs, Bangalore** for discussions

Thank You!