E0358

Uday Kumar Reddy B

uday@csa.iisc.ernet.in

Dept of CSA, Indian Institute of Science, Bangalore, India

A course on **advanced compilation** at
Dept of CSA
IISc

Research in Programming and Compiler Technologies

- **Current:**
  - C, C++, Java, Python, MATLAB, R, ...

Research in Programming and Compiler Technologies

- **Current:**
  - C, C++, Java, Python, MATLAB, R, ...
- **What will the new and disruptive programming technologies of the 21st century be?**

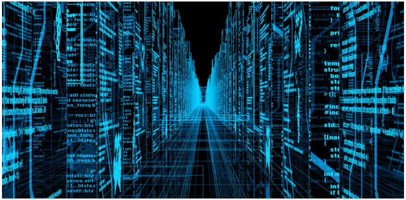
Research in Programming and Compiler Technologies

1. **What do programmers want?**
2. **How are architectures evolving?**
   - Multiple cores and many cores on a chip
   - GPUs, accelerators, and heterogeneous parallel architectures
   - Wider vector processing units
   - Deep memory hierarchies

High-Performance Compilation: What do you want to program?

- Scientific and engineering simulations
  - Eg: Solving partial differential equations numerically
- Embedded vision (Eg: Autonomous/self-driving cars)
- Smartphones — HPC in data centers and cloud drives a number of smartphone technologies
- **Scientific and Engineering simulations**
- **Data Analytics**
- **Deep Learning**
- **Artificial Intelligence**

- **What will the new programming technologies for the emerging domains be?**
  - **Current:** C, C++, Fortran with OpenMP, MPI, CUDA, OpenCL, ...
  - **Future: New languages, compilers, libraries, and DSLs**



Questions to Think About

- **What will the new programming technologies for Deep Learning be?**
  - Caffe, Theano, Torch, TensorFlow, ... are library-based approaches
  - **Just scratches the surface**

The Need for High Performance

- **More/Larger Data**
  - Instagram — 60 million photos / day
  - YouTube — 100 hours of video uploaded every minute
- **Need for a fast/real-time response in some domains**
- **More complex algorithms**
- **Science/Engineering simulations/modeling: Time to solution**

Programming Modern Hardware Effectively

- Compute speed: 4 multiply-adds per cycle
- Synchronization (2 cores 0.25 μs, 8 cores 1.25 μs, 2x8 cores 1.54 μs); memory bandwidth (20 GB/s)

Programming Modern Hardware Effectively

- Compute speed: 4 multiply-adds per cycle
- Synchronization (2 cores 0.25 μs, 8 cores 1.25 μs, 2x8 cores 1.54 μs); memory bandwidth (20 GB/s)
- High-Performance Programming and Compilation
  - Exploiting locality (caches, registers)
  - Reduce synchronization and communication as much as possible
  - Exploit single core hardware well (vectorization)
  - Multi-core parallelism
- Good scaling without good single thread performance is a great waste of resources (power, equipment cost)

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN

A Classification of Various Approaches

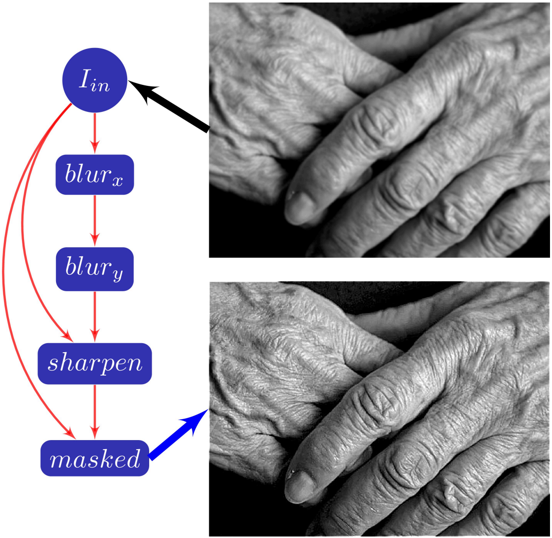
1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN
3. Ultra-high level languages/packages (R, MATLAB, ...)

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN
3. Ultra-high level languages/packages (R, MATLAB, ...)

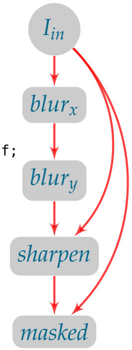
- **DSLs**: Obtain productivity of the last class and the performance of the first

Example 1: Unsharp Mask – an image processing pipeline



Unsharp Mask: Computation

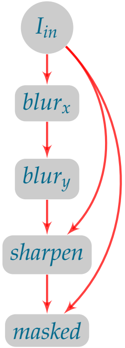
```
for (i = 0; i <= 2; i++)
 for (j = 2; j <= (R + 1); j++)
  for (k = 0; (k <= (C + 3)); k++)
   blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
    + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

for (i = 0; (i <= 2); i++)
 for (j = 2; (j <= (R + 1)); j++)
  for (k = 2; (k <= (C + 1)); k++)
   blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
       + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

for (i = 0; (i <= 2); i++)
 for (j = 2; (j <= (R + 1)); j++)
  for (k = 2; (k <= (C + 1)); k++)
   sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

for (i = 0; i <= 2; i++)
 for (j = 2; j <= R + 1; j++)
  for (k = 2; k <= C + 1; k++) {
   _ct0 = img[i][j][k];
   _ct1 = sharpen[i][j-2][k-2];
   _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
   mask[i][j-2][k-2] = _ct2;
  }
```



A sequential version in C: **18.6 ms** / frame
(using GCC with opts, quad-core Nehalem, 720p video)

Unsharp Mask - A Naive OpenMP version

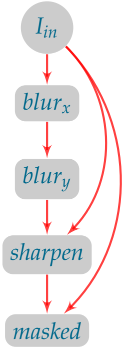
```
for (i = 0; i <= 2; i++)
#pragma omp parallel for
 for (j = 2; j <= (R + 1); j++)
#pragma ivdep
   for (k = 0; k <= C + 3; k++)
     blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
      + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

for (i = 0; i <= 2; i++)
#pragma omp parallel for
 for (j = 2; j <= R + 1; j++)
#pragma ivdep
   for (k = 2; k <= C + 1; k++)
     blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
          + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

for (i = 0; i <= 2; i++)
#pragma omp parallel for
 for (j = 2; j <= R + 1; j++)
#pragma ivdep
   for (k = 2; k <= C + 1; k++)
     sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

for (i = 0; i <= 2; i++)
#pragma omp parallel for private(_ct0,_ct1,_ct2)
 for (j = 2; j <= R + 1; j++)
#pragma ivdep
   for (k = 2; k <= C + 1; k++) {
     _ct0 = img[i][j][k];
     _ct1 = sharpen[i][j-2][k-2];
     _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
     mask[i][j-2][k-2] = _ct2;
   }
```



20.2 ms / frame on 1 thread, **18.02 ms** / frame on 4 threads

UNSHARP MASK - A BETTER OPENMP VERSION

```
#pragma omp parallel for
for (j = 2; j <= (R + 1); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 0; (k <= (C + 3)); k++)
      blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
       + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

#pragma omp parallel for
for (j = 2; (j <= (R + 1)); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; (k <= (C + 1)); k++)
      blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
           + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

#pragma omp parallel for
for (j = 2; (j <= (R + 1)); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; (k <= (C + 1)); k++)
      sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

#pragma omp parallel for private(_ct0,_ct1,_ct2)
for (j = 2; j <= R + 1; j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; k <= C + 1; k++) {
      _ct0 = img[i][j][k];
      _ct1 = sharpen[i][j-2][k-2];
      _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
      mask[i][j-2][k-2] = _ct2;
    }
```



18.6 ms / frame on 1 thread, **15.03 ms** / frame on 4 threads

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
frame_f = np.float32(frame) / 255.0
res = frame_f
kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
blury = sepFilter2D(frame_f, -1, kernelx, kernely)
sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
choose = choose.astype(bool)
np.copyto(res, sharpen, 'same_kind', choose)
return res
```

Performance: **35.9 ms** / frame

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
frame_f = np.float32(frame) / 255.0
res = frame_f
kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
blury = sepFilter2D(frame_f, -1, kernelx, kernely)
sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
choose = choose.astype(bool)
np.copyto(res, sharpen, 'same_kind', choose)
return res
```

   Performance: **35.9 ms** / frame

2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
frame_f = np.float32(frame) / 255.0
res = frame_f
kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
blury = sepFilter2D(frame_f, -1, kernelx, kernely)
sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
choose = choose.astype(bool)
np.copyto(res, sharpen, 'same_kind', choose)
return res
```

   Performance: **35.9 ms** / frame

2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
frame_f = np.float32(frame) / 255.0
res = frame_f
kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
blury = sepFilter2D(frame_f, -1, kernelx, kernely)
sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
choose = choose.astype(bool)
np.copyto(res, sharpen, 'same_kind', choose)
return res
```

   Performance: **35.9 ms** / frame

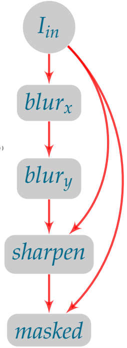
2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get to this, and build compilers/code generators that can achieve this automatically)

- **Video demo**

UNSHARP MASK - A HIGHLY OPTIMIZED VERSION

Note: *Code below is indicative and not meant for reading! Zoom into soft copy or browse source code repo listed in references.*

```
#pragma omp parallel for schedule(static)
 for (int _T_i1 = 0; (_T_i1 <= ((R + 1) / 32)); _T_i1 = (_T_i1 + 1))
 {
  int _ct0 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
  int _ct1 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
  int _ct4 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
  int _ct5 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
  int _ct8 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
  int _ct9 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
  int _ct12 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
  int _ct13 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
  for (int _T_i2 = -1; (_T_i2 <= ((C + 3) / 256)); _T_i2 = (_T_i2 + 1))
  {
   int _ct2 = (((C + 3) < ((256 * _T_i2) + 261))? (C + 3): ((256 * _T_i2) + 261));
   int _ct3 = ((0 > (256 * _T_i2))? 0: (256 * _T_i2));
   int _ct6 = (((C + 1) < ((256 * _T_i2) + 260))? (C + 1): ((256 * _T_i2) + 260));
   int _ct7 = ((2 > ((256 * _T_i2) + 1))? 2: ((256 * _T_i2) + 1));
   int _ct10 = (((C + 1) < ((256 * _T_i2) + 259))? (C + 1): ((256 * _T_i2) + 259));
   int _ct11 = ((2 > ((256 * _T_i2) + 2))? 2: ((256 * _T_i2) + 2));
   int _ct14 = (((C + 1) < ((256 * _T_i2) + 258))? (C + 1): ((256 * _T_i2) + 258));
   int _ct15 = ((2 > ((256 * _T_i2) + 3))? 2: ((256 * _T_i2) + 3));
   for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
   {
    for (int _i1 = _ct1; (_i1 <= _ct0); _i1 = (_i1 + 1))
    {
     #pragma ivdep
     for (int _i2 = _ct3; (_i2 <= _ct2); _i2 = (_i2 + 1))
     {
      blurx[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] = (((((img[(((_i0 * ((R + 4) * (C + 4))) + ((-2 + _i1) * (C + 4))) + _i2)] * 0.0625f) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((-1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 * ((R + 4) * (C + 4)))
+ (_i1 * (C + 4))) + _i2)] * 0.375f)) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((2 + _i1) * (C + 4))) + _i2)] * 0.0625f));
     }
    }
   }
   for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
   {
    for (int _i1 = _ct5; (_i1 <= _ct4); _i1 = (_i1 + 1))
    {
     #pragma ivdep
     for (int _i2 = _ct7; (_i2 <= _ct6); _i2 = (_i2 + 1))
     {
      blury[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] = (((((blurx[_i0][((-32 * _T_i1) + _i1)][(-2 + ((-256 * _T_i2) + _i2))] * 0.0625f) + (blurx[_i0][((-32 * _T_i1) + _i1)][(-1 + ((-256 * _T_i2) + _i2))] * 0.25f))
+ (blurx[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] * 0.375f)) + (blurx[_i0][((-32 * _T_i1) + _i1)][(1 + ((-256 * _T_i2) + _i2))] * 0.25f)) + (blurx[_i0][((-32 * _T_i1) + _i1)][(2 + ((-256 * _T_i2) + _i2))] * 0.0625f));
     }
    }
   }
   for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
   {
    for (int _i1 = _ct9; (_i1 <= _ct8); _i1 = (_i1 + 1))
    {
     #pragma ivdep
     for (int _i2 = _ct11; (_i2 <= _ct10); _i2 = (_i2 + 1))
     {
      sharpen[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] = ((img[(((_i0 * ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)] * (1 + weight)) + (blury[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] * -(weight)));
     }
    }
   }
   for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
   {
    for (int _i1 = _ct13; (_i1 <= _ct12); _i1 = (_i1 + 1))
    {
     #pragma ivdep
     for (int _i2 = _ct15; (_i2 <= _ct14); _i2 = (_i2 + 1))
     {
      float _ct16 = img[(((_i0 * ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)];
      float _ct17 = sharpen[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)];
      float _ct18 = ((std::abs((img[(((_i0 * ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)] - blury[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)])) < threshold)? _ct16: _ct17);
      mask_flip[((((_i1-2) * (3 * C)) + ((_i2 - 2) * 3)) + (_i0))] = _ct18;
     }
    }
   }
  }
 }
```



15.5 ms / frame on 1 threads, **8.97 ms** / frame on 4 threads

Example 2: GEMVER

$$
\begin{aligned}
B &= A + u_1 * v_1^T + u_2 * v_2^T \\
x &= x + B^T y \\
x &= x + z \\
w &= w + B * x
\end{aligned}
$$

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  x[i] = x[i] + beta* B[j][i]*y[j];

for (i=0; i<N; i++)
 x[i] = x[i] + z[i];

 for (i=0; i<N; i++)
  for (j=0; j<N; j++)
   w[i] = w[i] + alpha* B[i][j]*x[j];
```

The second loop nest operates in parallel along columns of B
The fourth loop nest operates in parallel along rows of B

Example 2. GEMVER – Block distribution

- The first loop nest requires distributing B column-wise:

P0	P1	P2	P3
P0	P1	P2	P3
P0	P1	P2	P3
P0	P1	P2	P3

- And the second loop nest requires it row-wise:

P0	P0	P0	P0
P1	P1	P1	P1
P2	P2	P2	P2
P3	P3	P3	P3

- One needs a transpose in between (an all-to-all communication) to extract parallelism from both steps (ignore reduction parallelism)
- $O(N^2)$ communication for matrix B

EXAMPLE 2. GEMVER WITH A BLAS LIBRARY

- With a library, one would just use a block cyclic distribution:

  *dcopy(m * n, A, 1, B, 1);*
  dger(m, n, 1.0, u1, 1, v1 , 1, B, m);
  dger(m, n, 1.0, u2, 1, v2 , 1, B, m);
  dcopy(n,z,1,x,1);
  dgemv('T', m, n, beta, B, m, y, 1, 1.0, x, 1);
  dgemv('N', m, n, alpha, B, m, x, 1, 0.0, w, 1);

- Can we do better?

EXAMPLE 2. GEMVER: SUDOKU MAPPING

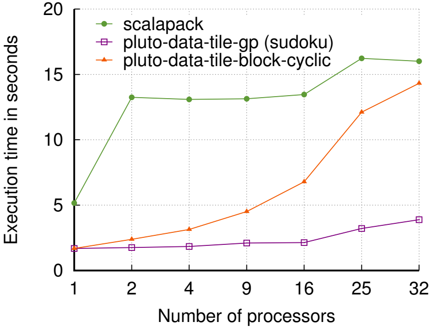
- Use a Sudoku-style mapping [NAS MG, BT, dHPF]
- Both load balance and $O(N)$ communication on x and w (no communication for B) (optimal)

P0	P1	P2	P3
P1	P2	P3	P0
P2	P3	P0	P1
P3	P0	P1	P2

- A compiler can derive such a mapping based on a model and generate much better code – mapping that is **globally** good

EXAMPLE 2. GEMVER: PERFORMANCE

- A compiler optimizer or code generator can select a globally good transformation



- On a 32-node InfiniBand cluster (32x8 cores) (weak scaling: same problem size per node)

Domain-Specific Languages (DSL)

- **Both examples above motivate a domain-specific language + compiler approach**

Domain-Specific Languages (DSL)

- **Both examples above motivate a domain-specific language + compiler approach**
- **High-performance domain-specific language + compiler**: productivity similar to ultra high-level or high-level but performance similar to manual or even better!

DSLs

- Exploit domain information to improve programmability, performance, and portability

Domain-Specific Languages (DSL)

DSLs

- Exploit domain information to improve programmability, performance, and portability
- Expose greater information to the compiler and programmer specifies less
- abstract away many things from programmers (parallelism, memory)

DSL compilers

- can "see" **across** routines – allow whole program optimization
- generate optimized code for multiple targets
- Programmers say **what** to execute and not **how** to execute

Big Picture: Role of Compilers

General-Purpose

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite

Domain-Specific

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide

Big Picture: Role of Compilers

General-Purpose

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite
- Limited improvements, not everything is possible
- Broad impact

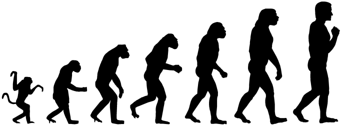
Domain-Specific

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide
- Dramatic speedups, Automatic parallelization
- Narrower impact and adoption

Big Picture: Role of Compilers

EVOLUTIONARY approach

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite



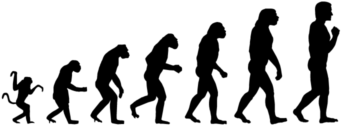
REVOLUTIONARY approach

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide

Big Picture: Role of Compilers

EVOLUTIONARY approach

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite



REVOLUTIONARY approach

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide

- **Both approaches share infrastructure**
- **Important to pursue both**

Outline

HANDS-ON TRIAL

- Tools/Infrastructure to install and try
  - Barvinok tool: `http://barvinok.gforge.inria.fr/`
  - Pluto `http://pluto-compiler.sourceforge.net` (**pet** branch of git version)
- For assignment at the end of second lecture
  - PolyMage: https://bitbucket.org/udayb/polymage.git *e0358* git branch

Compilers: What comes to mind?

- GCC, LLVM

- Scanning, Parsing, Semantic analysis
- Scalar optimizations: SSA, constant propagation, dead code elimination
- **High-level optimizations**
- Backend: Register allocation, Instruction scheduling

What should a compiler designer think about?

1. Productivity: how easy it is to program?
2. **Performance: how well does the code perform?**
3. Portability: how portable is your code? Will it run on a different architecture?

1. Productivity
   - Expressiveness: ease of writing, lines of code
   - Productivity in writing a correct program, and in writing a performing parallel program
   - Library support, Debugging support, Interoperability

High-Performance language/compiler design

1. Productivity
   - Expressiveness: ease of writing, lines of code
   - Productivity in writing a correct program, and in writing a performing parallel program
   - Library support, Debugging support, Interoperability
2. Performance
   - Locality (spatial, temporal, ...)
   - Multi-core parallelism, coarse-grained parallelization
   - SIMD parallelism, vectorization
   - Parallelism granularity, Synchronization, Communication
   - Dynamic scheduling, Load balancing
   - Data allocation, Memory mapping and optimization

HIGH-PERFORMANCE LANGUAGE/COMPILER DESIGN

1. Productivity
   - Expressiveness: ease of writing, lines of code
   - Productivity in writing a correct program, and in writing a performing parallel program
   - Library support, Debugging support, Interoperability
2. Performance
   - Locality (spatial, temporal, ...)
   - Multi-core parallelism, coarse-grained parallelization
   - SIMD parallelism, vectorization
   - Parallelism granularity, Synchronization, Communication
   - Dynamic scheduling, Load balancing
   - Data allocation, Memory mapping and optimization
3. Portability
   - Given a new machine, how much time does it take to port?
   - How well will it perform? How much more time to tune and optimize?

Automatic Parallelization

Automatic parallelization: programmer provides a sequential specification, and the compiler or compiler+runtime parallelizes it

Automatic Parallelization

Automatic parallelization: programmer provides a sequential specification, and the compiler or compiler+runtime parallelizes it

- **Myths**
  - Automatic parallelization is about just detecting and marking loops parallel
  - Has been a failure
  - Scope restricted to general-purpose compilers

Automatic Parallelization

Automatic parallelization: programmer provides a sequential specification, and the compiler or compiler+runtime parallelizes it

- **Myths**
  - Automatic parallelization is about just detecting and marking loops parallel
  - Has been a failure
  - Scope restricted to general-purpose compilers
- **What it really is**
  - Execution and data restructuring to execute in parallel efficiently
  - Important in DSL compilers
  - Can be used for library creation/generation

Outline

Polyhedral Framework

```
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]);
```

1. **Domains**
   - Every statement has a domain or an index **set –** instances that have to be executed
   - Each instance is a vector (of loop index values from outermost to innermost)
     $D_S = \{[t, i, j] \mid 0 \leq t \leq T-1,\ 1 \leq i, j \leq N\}$
2. **Dependences**
   - A dependence is a **relation** between domain / index set instances that are in conflict (more on next slide)
3. **Schedules**
   - are **functions** specifying the *order* in which the domain instances should be executed
   - Specified statement-wise and **typically** one-to-one
   - $T((i, j)) = (i + j, j)$ or $\{[i, j] \rightarrow [i + j, j] \mid \ldots\}$

Domains, Dependences, and Schedules

```
for (i=1; i<=N-1; i++)
   for (j=1; j<=N-1; j++)
      A[i][j] = f(A[i-1][j], A[i][j-1]);
```

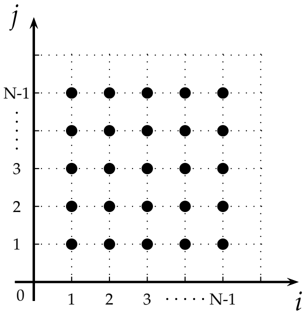


Figure: Original space (i, j)

- **Domain**: $\{[i, j] \mid 1 \leq i, j \leq N - 1\}$

```
for (i=1; i<=N-1; i++)
   for (j=1; j<=N-1; j++)
      A[i][j] = f(A[i-1][j], A[i][j-1]);
```

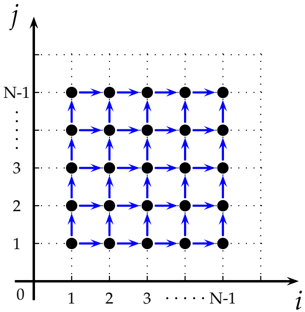


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \rightarrow [i+1,j] \mid 1 \leq i \leq N-2, 0 \leq j \leq N-1\}$ — **(1,0)**
  2. $\{[i,j] \rightarrow [i,j+1] \mid 1 \leq i \leq N-1, 0 \leq j \leq N-2\}$ — **(0,1)**

Domains, Dependences, and Schedules

```
for (i=1; i<=N-1; i++)
   for (j=1; j<=N-1; j++)
      A[i][j] = f(A[i-1][j], A[i][j-1]);
```

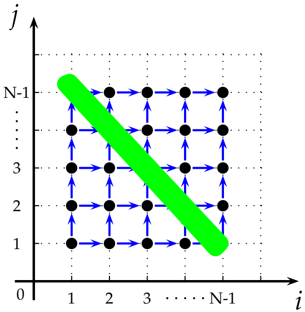


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \to [i+1,j] \mid 1 \le i \le N-2, 0 \le j \le N-1\}$ — **(1,0)**
  2. $\{[i,j] \to [i,j+1] \mid 1 \le i \le N-1, 0 \le j \le N-2\}$ — **(0,1)**

Domains, Dependences, and Schedules

```
for (i=1; i<=N-1; i++)
   for (j=1; j<=N-1; j++)
      A[i][j] = f(A[i-1][j], A[i][j-1]);
```

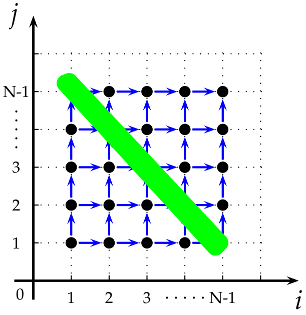


Figure: Original space (i, j)

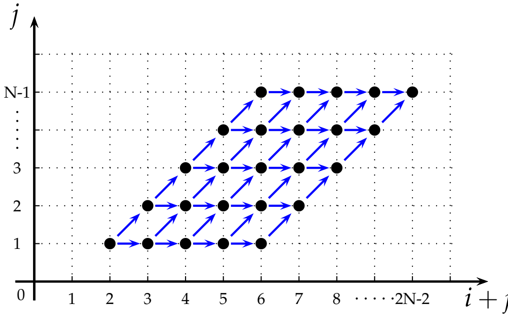


Figure: Transformed space $(i + j, j)$

- **Schedule**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.

Domains, Dependences, and Schedules

```
for (i=1; i<=N-1; i++)
   for (j=1; j<=N-1; j++)
      A[i][j] = f(A[i-1][j], A[i][j-1]);
```

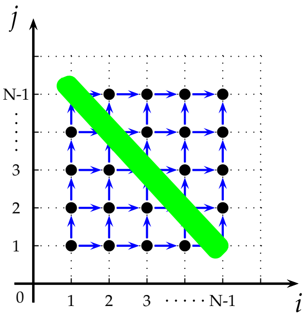


Figure: Original space (i, j)

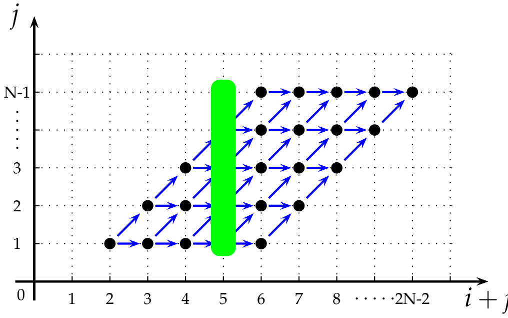


Figure: Transformed space $(i + j, j)$

- **Schedule**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.
  - Inner loop is now parallel

Lexicographic Ordering

- **Lexicographic ordering**: $\succ, \succ \vec{0}$
- **Schedules/Affine Transformations/Polyhedral Transformations** as a way to provide multi-dimensional timestamps
- Code generation: **Scanning points in the transformed space in lexicographically increasing order**

Polyhedral Framework: Schedules

```
for (i=1 i<N; i++)
  P(i); /* Produces B[i] using another array A */

for (i=1; i<N; i++)
  C(i); /* Consumes B[i] and B[i-1] to create D[i] */
```

- Original schedule: $T_P(i) = (0, i)$, $T_C(i) = (1, i)$

Polyhedral Framework: Schedules

```
for (i=1 i<N; i++)
  P(i); /* Produces B[i] using another array A */

for (i=1; i<N; i++)
  C(i); /* Consumes B[i] and B[i-1] to create D[i] */
```

- Original schedule: $T_P(i) = (0, i), T_C(i) = (1, i)$
- Fused
  - Schedule: $T_P(i) = (i, 0), \quad T_C(i) = (i, 1).$

```
for (t1=1; t1<N; t1++) {
   P(t1);
   C(t1);
}
```

- A code generator needs **domains** and **schedules**

Polyhedral Framework: Schedules

```
for (i=1 i<N; i++)
  P(i); /* Produces A[i] */

for (i=1; i<N; i++)
  C(i); /* Consumes A[i] and A[i-1] */
```

- Original schedule: $T_P(i) = (0, i)$, $T_C(i) = (1, i)$
- Fused + Tiled
  - Schedule: $T_P(i) = (i/32, i, 0)\,, \quad T_C(i) = (i/32, i, 1).$

```
for (t1=0;t1<=floord(N-1,32);t1++) {
   for (t3=max(1,32*t1);t3<=min(N-1,32*t1+31);t3++) {
      P(t3);
      C(t3);
   }
}
```

- A code generator needs **domains** and **schedules**

Polyhedral Framework: Schedules

```
for (i=1 i<N; i++)
  P(i); /* Produces A[i] */

for (i=1; i<N; i++)
  C(i); /* Consumes A[i] and A[i-1] */
```

- Original schedule: $T_P(i) = (0, i)$, $T_C(i) = (1, i)$
- Fused + Tiled + Innermost distribute
  - Produce a chunk of A and consume it before a new chunk is produced
  - Schedule: $T_P(i) = (i/32, 0, i)$, $\quad T_C(i) = (i/32, 1, i)$.

```
for (t1=0;t1<=floord(N-1,32);t1++) {
 for (t3=max(1,32*t1;t3<=min(N-1,32*t1+31);t3++)
   P(t3);
 for (t3=max(1,32*t1);t3<=min(N-1,32*t1+31);t3++)
   C(t3);
}
```

- A code generator needs **domains** and **schedules**

Outline

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, $i^2 + j^2$, $a[j]$

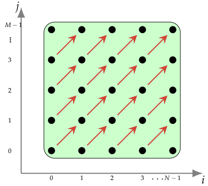


Figure: Iteration space

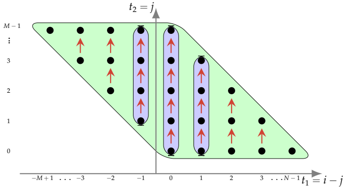


Figure: Transformed space

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

/* O(N) synchronization if j is parallelized */
```

```
#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]);

/* Synchronization-free */
```

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

Affine Transformations

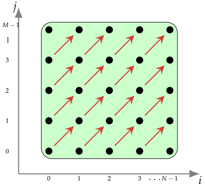


Figure: Iteration space

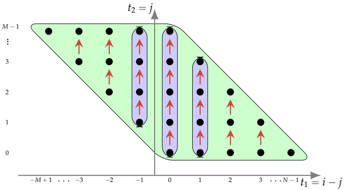


Figure: Transformed space

- 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+)
  - Model a very rich class of loop re-orderings
  - Useful for several domains like dense linear algebra, stencil computations, image processing pipelines, deep learning

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile (rectangular)
$\ldots$	

- One-to-one functions

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile (rectangular)
$\ldots$	

- One-to-one functions
- Can be expressed using matrices:

$$T(i, j) = (i + j, j) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}.$$

- Validity: dependences should not be violated

Dependences

- Dependences are determined pairwise between conflicting accesses

```
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]);
```

- Dependence notations
  - Distance vectors: (1,-1,0), (1,0,0), (1,1,0), (1,0,-1), (1,0,1)
  - Direction vectors
  - Dependence relations as integer sets with affine constraints and existential quantifiers or Presburger formulae — powerful
- Consider the dependence from the write to the third read: $A[(t+1)\%2][i][j] \rightarrow A[t'\%2][i'-1][j']$
  Dependence relation: $\{[t,i,j] \rightarrow [t',i',j'] \mid t' = t+1, i' = i+1, j' = j, 0 \le t \le T-1,\ 0 \le i \le N-1, 0 \le j \le N\}$

Preserving Dependences

```
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]);
```

- For affine loop nests, these dependences can be analyzed and represented precisely
- **Side note**: A DSL simplifies dependence analysis

Preserving Dependences

```
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]);
```

- For affine loop nests, these dependences can be analyzed and represented precisely
- **Side note**: A DSL simplifies dependence analysis
- **Next step:** Transform while preserving dependences
  - Find execution reorderings that **preserve** dependences and improve performance
  - Execution reordering as a function: $T(\vec{i})$
  - For all dependence relation instances $(\vec{s} \rightarrow \vec{t})$,
    $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$,
    i.e., the source should precede the target even in the transformed space
- What is the structure of **T**?

Valid Transformations

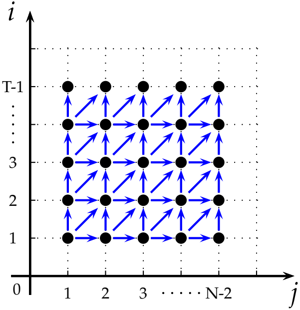
```
for (t = 0; t < T; t++)
 for (i = 1; i < N+1; i++)
  for (j = 1; j < N+1; j++)
   A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                     A[t%2][i][j+1], A[t%2][i][j-1]);
```

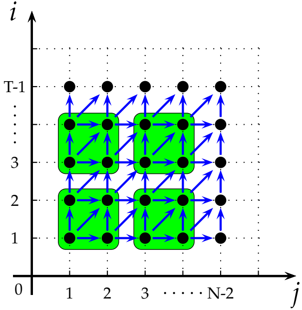
- Dependences: $(1, 0, 0)$, $(1, 0, 1)$, $(1, 0, -1)$, $(1, 1, 0)$, (1,-1,0)
- Validity: $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$, i.e., $T(\vec{t} - \vec{s}) \succ \vec{0}$
- Examples of invalid transformations
  - $T(t, i, j) = (i, j, t)$
  - Similarly, (i, t, j), (j, i, t), $(t + i, i, j)$, $(t + i + j, i, j)$ are all invalid transformations
- Valid transformations
  - (t, j, i), $(t, t + i, t + j)$, $(t, t + i, t + i + j)$
  - However, only some of the infinitely many valid ones are interesting

Outline

TILING (BLOCKING)

- Partition and execute iteration space in blocks
- A tile is executed atomically
- Benefits: exploits *cache locality* & improves *parallelization* in the presence of synchronization
- **Allows reuse in multiple directions**
- **Reduces frequency of synchronization** for parallelization: synchronization after you execute *tiles* (as opposed to *points*) in parallel





$(\mathbf{i},\mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50},\mathbf{j}/\mathbf{50},\mathbf{i},\mathbf{j});$ $\quad (\mathbf{i},\mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50}+\mathbf{j}/\mathbf{50},\mathbf{j}/\mathbf{50},\mathbf{i},\mathbf{j})$

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

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

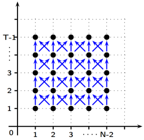


Figure: Iteration space

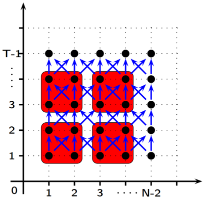


Figure: Invalid tiling

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

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

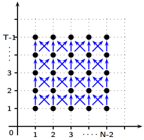


Figure: Iteration space

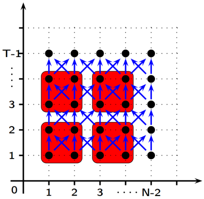


Figure: Invalid tiling

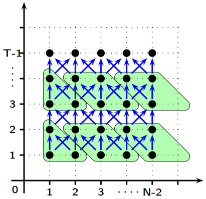


Figure: Valid tiling

Tiling (blocking)

- Affine transformations can enable tiling
  - First skew: $T(i,j) = (i, i+j)$
  - Then, create a wavefront of tiles:
    $T(i,j) = (i/64 + (i+j)/64, (i+j)/64, i, i+j)$

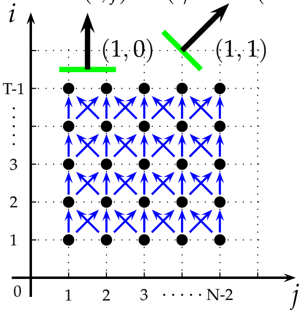


Figure: Original space (i,j)

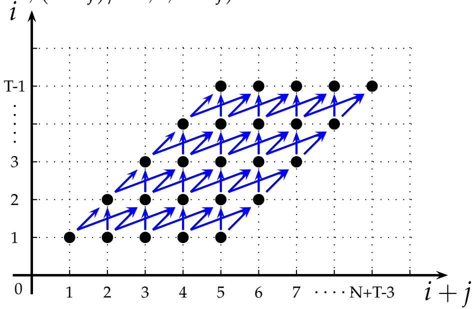


Figure: Transformed space $(i, i+j)$

TILING (BLOCKING)

- Affine transformations can enable tiling
  - First skew: $T(i,j) = (i, i+j)$
  - Then, apply (rectangular) tiling:
    $T(i,j) = (i/64, (i+j)/64, i, i+j)$
    - i and $i+j$ are also called *tiling hyperplanes*
  - Then, create a wavefront of tiles:
    $T(i,j) = (i/64 + (i+j)/64, (i+j)/64, i, i+j)$

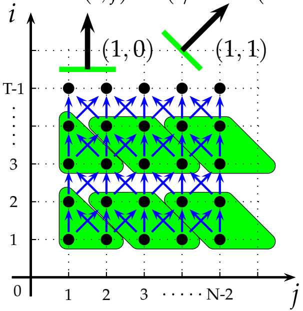


Figure: Original space (i, j)

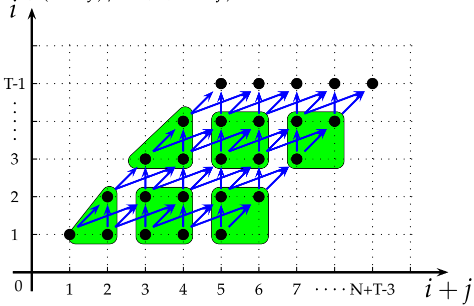


Figure: Transformed space $(i, i+j)$

Algorithms to Find Transformations

- **The Past**
  - A data locality optimizing algorithm, Wolf and Lam, PLDI 1991
    Improve locality through unimodular transformations
    - Characterize self-spatial, self-temporal, and group reuse
    - Find unimodular transformations (permutation, reversal, skewing) to transform to permutable loop nests with reuse, and subsequently tile them
- Several advances on polyhedral transformation algorithms through 1990s and 2000s – Feautrier [1991–1992], Lim and Lam – Affine Partitioning [1997–2001], Pluto [2008 – present]
- **The Present**
  - Polyhedral framework provides a powerful mathematical abstraction (away from the syntax)
  - A number of new techniques, open-source libraries and tools have been developed and are **actively maintained**

Back to 3-D example

```
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]);
```

- What is a good transformation here to improve parallelism and locality?
- Steps

BACK TO 3-D EXAMPLE

```
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]);
```

- What is a good transformation here to improve parallelism and locality?
- Steps
  - Skewing: $(t, t+i, t+j)$

Back to 3-D example

```
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]);
```

- What is a good transformation here to improve parallelism and locality?
- Steps
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-D example

```
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]);
```

- What is a good transformation here to improve parallelism and locality?
- Steps
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Parallelize by creating tile wavefront: $(t/64 + (t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Polyhedral Transformation Algorithms

- Feautrier [1991–1992] scheduling
- Lim and Lam, Affine Partitioning [1997–2001]
- Pluto algorithm [Bondhugula et al. 2008]
  - **Finds a sequence of affine transformations to improve locality and parallelism**
  - Transforms to bands of tilable dimensions
  - Bounds dependence distances and minimizes them
  - Objective: **minimize dependence distances** while maximizing tilability
- PPCG [Verdoolaege et al. 2013] (mainly for GPUs) – can generate CUDA or OpenCL code

A Cost Function to Select Affine Transformations

- $T_1(t,i) = (t/64 + (t+i)/64, t/64, t, t+i)$
- $T_2(t,i) = (t/64 + (t+i)/64, (t+i)/64, t, t+i)$
- $T_3(t,i) = (t/64 + (2t+i)/64, (2t+i)/64, t, 2t+i)$

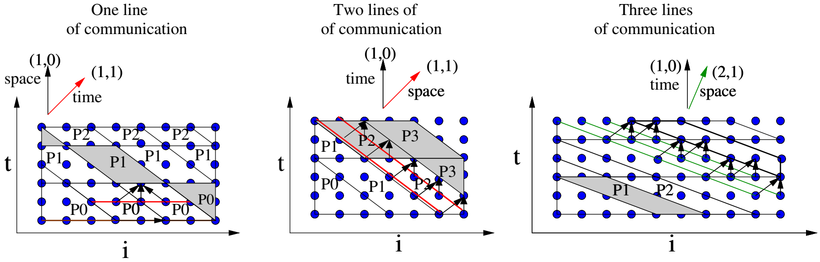
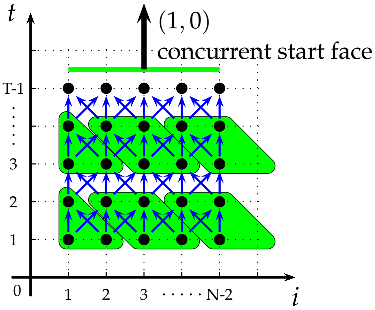


Figure: Communication volume with different valid hyperplanes for 1-d Jacobi: shaded tiles are to be executed in parallel

- Select the $\vec{h}$ that minimizes $\vec{h}.(\vec{t} - \vec{s})$, i.e., minimizes $\vec{h}.\vec{d}$
- Examples: $\vec{h} = (2,1), \vec{h}.(1,1) = 3; \vec{h} = (1,0), \vec{h}.(1,1) = 1.$

Outline

Pipelined Start and Load Imbalance




```
for (t = 0; t <= T-1; t++)
  for (i = 1; i <= N-2; i++)
    A[(t+1)%2][i] = 0.125 * (A[t%2][i+1]
        - 2.0 * A[t%2][i] + A[t%2][i-1]);
```

PIPELINED START AND LOAD IMBALANCE

Classical time skewing suffers from pipelined startup

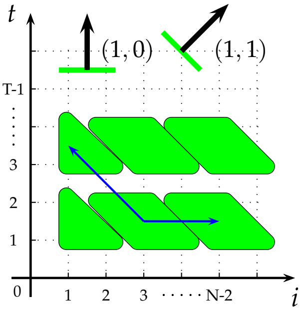


Figure: Pipelined start

Pipelined Start and Load Imbalance

Classical time skewing suffers from pipelined startup

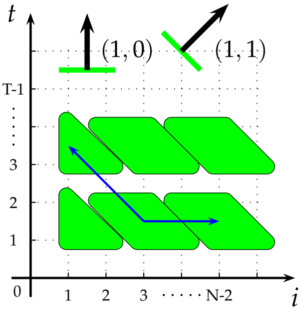


Figure: Pipelined start

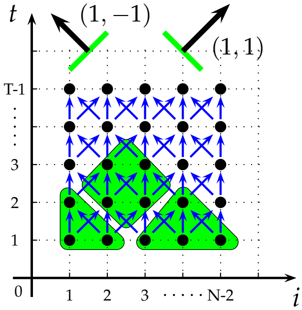


Figure: Group as diamonds

Classical time skewing suffers from pipelined startup

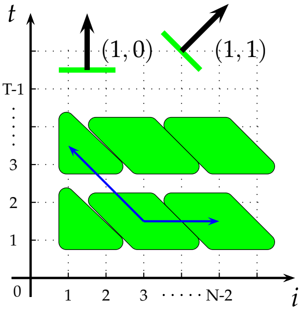


Figure: Pipelined start

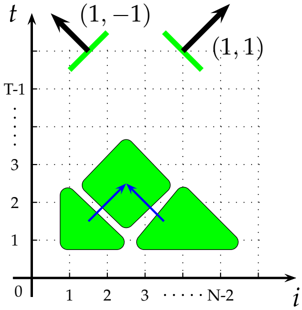


Figure: Concurrent start possible

- Diamond tiling
  - Face allowing concurrent start 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

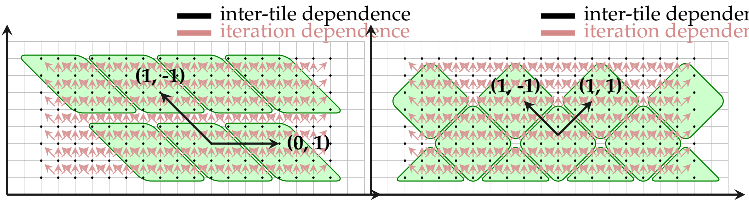


Figure: Two ways of tiling heat-1d: parallelogram & diamond

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

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

4. Choose tile sizes in Step 2 such that vectorization and prefetching works well (for the innermost dimension)

Transformed code

```
/* Start of CLooG code */
for (t1=-1; t1<=31; t1++) {
  int lbp=ceild(t1,2), ubp=floord(t1+125,2);
  #pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
  for (t2=lbp; t2<=ubp; t2++)
    for (t3=max(0,ceild(t1-1,2)); t3<=floord(t1+126,2); t3++)
      for (t4=max(max(max(0,32*t1),64*t3-4000),64*t1-64*t2+1);
              t4<=min(min(min(999,32*t1+63),64*t2+62),64*t3+62); t4++)
        for (t5=max(max(64*t2,t4+1),-64*t1+64*t2+2*t4-63);
                t5<=min(min(64*t2+63,t4+4000),-64*t1+64*t2+2*t4); t5++)
#pragma ivdep
#pragma vector always
          for (t6=max(64*t3,t4+1); t6<=min(64*t3+63,t4+4000); t6++)
            A[( t4 + 1) % 2][ (-t4+t5)][ (-t4+t6)] = (((0.125 * ((A[ t4 % 2][ (-t4+t5) + 1][ (-t4+t6)]
                - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)])) + A[ t4 % 2][ (-t4+t5) - 1][ (-t4+t6)]))
            + (0.125 * ((A[ t4 % 2][ (-t4+t5)][ (-t4+t6) + 1] - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]))
                    + A[ t4 % 2][ (-t4+t5)][ (-t4+t6) - 1]))) + A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]);
}
/* End of CLooG code */
```

Performance on an 8-core Intel Xeon Haswell (all code compiled with ICC 16.0), N=4000, T=1000

- Original: 6.2 GFLOPS

TRANSFORMED CODE

```
/* Start of CLooG code */
for (t1=-1; t1<=31; t1++) {
  int lbp=ceild(t1,2), ubp=floord(t1+125,2);
  #pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
  for (t2=lbp; t2<=ubp; t2++)
    for (t3=max(0,ceild(t1-1,2)); t3<=floord(t1+126,2); t3++)
      for (t4=max(max(max(0,32*t1),64*t3-4000),64*t1-64*t2+1);
              t4<=min(min(min(999,32*t1+63),64*t2+62),64*t3+62); t4++)
        for (t5=max(max(64*t2,t4+1),-64*t1+64*t2+2*t4-63);
                t5<=min(min(64*t2+63,t4+4000),-64*t1+64*t2+2*t4); t5++)
#pragma ivdep
#pragma vector always
          for (t6=max(64*t3,t4+1); t6<=min(64*t3+63,t4+4000); t6++)
            A[( t4 + 1) % 2][ (-t4+t5)][ (-t4+t6)] = (((0.125 * ((A[ t4 % 2][ (-t4+t5) + 1][ (-t4+t6)]
                - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)])) + A[ t4 % 2][ (-t4+t5) - 1][ (-t4+t6)]))
             + (0.125 * ((A[ t4 % 2][ (-t4+t5)][ (-t4+t6) + 1] - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]))
                     + A[ t4 % 2][ (-t4+t5)][ (-t4+t6) - 1]))) + A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]);
}
/* End of CLooG code */
```

Performance on an 8-core Intel Xeon Haswell (all code compiled with ICC 16.0), N=4000, T=1000

- Original: 6.2 GFLOPS
- Straightforward OMP: 21.8 GFLOPS

TRANSFORMED CODE

```
/* Start of CLooG code */
for (t1=-1; t1<=31; t1++) {
  int lbp=ceild(t1,2), ubp=floord(t1+125,2);
  #pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
  for (t2=lbp; t2<=ubp; t2++)
    for (t3=max(0,ceild(t1-1,2)); t3<=floord(t1+126,2); t3++)
      for (t4=max(max(max(0,32*t1),64*t3-4000),64*t1-64*t2+1);
              t4<=min(min(min(999,32*t1+63),64*t2+62),64*t3+62); t4++)
        for (t5=max(max(64*t2,t4+1),-64*t1+64*t2+2*t4-63);
                t5<=min(min(64*t2+63,t4+4000),-64*t1+64*t2+2*t4); t5++)
#pragma ivdep
#pragma vector always
          for (t6=max(64*t3,t4+1); t6<=min(64*t3+63,t4+4000); t6++)
            A[( t4 + 1) % 2][ (-t4+t5)][ (-t4+t6)] = (((0.125 * ((A[ t4 % 2][ (-t4+t5) + 1][ (-t4+t6)]
                - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)])) + A[ t4 % 2][ (-t4+t5) - 1][ (-t4+t6)]))
             + (0.125 * ((A[ t4 % 2][ (-t4+t5)][ (-t4+t6) + 1] - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]))
                     + A[ t4 % 2][ (-t4+t5)][ (-t4+t6) - 1]))) + A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]);
}
/* End of CLooG code */
```

Performance on an 8-core Intel Xeon Haswell (all code compiled with ICC 16.0), N=4000, T=1000

- Original: 6.2 GFLOPS
- Straightforward OMP: 21.8 GFLOPS
- Classical time skewing: 52 GFLOPS (2.39x over simple OMP)

TRANSFORMED CODE

```
/* Start of CLooG code */
for (t1=-1; t1<=31; t1++) {
  int lbp=ceild(t1,2), ubp=floord(t1+125,2);
  #pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
  for (t2=lbp; t2<=ubp; t2++)
    for (t3=max(0,ceild(t1-1,2)); t3<=floord(t1+126,2); t3++)
      for (t4=max(max(max(0,32*t1),64*t3-4000),64*t1-64*t2+1);
              t4<=min(min(min(999,32*t1+63),64*t2+62),64*t3+62); t4++)
        for (t5=max(max(64*t2,t4+1),-64*t1+64*t2+2*t4-63);
                t5<=min(min(64*t2+63,t4+4000),-64*t1+64*t2+2*t4); t5++)
#pragma ivdep
#pragma vector always
          for (t6=max(64*t3,t4+1); t6<=min(64*t3+63,t4+4000); t6++)
            A[( t4 + 1) % 2][ (-t4+t5)][ (-t4+t6)] = (((0.125 * ((A[ t4 % 2][ (-t4+t5) + 1][ (-t4+t6)]
                 - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)])) + A[ t4 % 2][ (-t4+t5) - 1][ (-t4+t6)]))
             + (0.125 * ((A[ t4 % 2][ (-t4+t5)][ (-t4+t6) + 1] - (2.0 * A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]))
                         + A[ t4 % 2][ (-t4+t5)][ (-t4+t6) - 1]))) + A[ t4 % 2][ (-t4+t5)][ (-t4+t6)]);
}
/* End of CLooG code */
```

Performance on an 8-core Intel Xeon Haswell (all code compiled with ICC 16.0), N=4000, T=1000

- Original: 6.2 GFLOPS
- Straightforward OMP: 21.8 GFLOPS
- Classical time skewing: 52 GFLOPS (2.39x over simple OMP)
- **Diamond tiling**: 91 GFLOPS (4.17x over simple OMP)

Where are Affine Transformations useful?

- Application domains
  - Optimize Jacobi and other relaxations via time tiling
  - Optimize pre-smoothing steps at various levels of Geometric Multigrid method
  - Optimize Lattice Boltzmann Method computations
  - Image Processing Pipelines
  - Convolutional Neural Network computations
  - **Wherever you have loops and want to transform loops**
- Architectures
  - General-purpose multicores
  - GPUs, accelerators
  - FPGAs: transformations for HLS

Putting Transformations into Practice

- **Where are these transformations useful?**
  - In general-purpose compilers: LLVM, GCC, ...
  - In DSL compilers
- **Tools: How to use these?**
  - ISL `http://isl.gforge.inria.fr` – an Integer Set Library
  - CLooG – polyhedral code generator/library `http://cloog.org`
  - Pluto `http://pluto-compiler.sourceforge.net` – a source-to-source automatic transformation framework that uses a number of libraries including Pet, Clan, Candl, ISL, Cloog, Piplib
  - PPCG – Polyhedral parallel code generation for CUDA http://repo.or.cz/ppcg.git
  - Polly `http://polly.llvm.org` – Polyhedral infrastructure in LLVM
- **An exercise now**

References

- Reading material, tutorials, and slides
  - *Presburger Formulas and Polyhedral Compilation* by Sven Verdoolaege `http://isl.gforge.inria.fr/`
  - Barvinok tutorial at http://barvinok.gforge.inria.fr/
  - Background and Theory on Automatic Polyhedral Transformations `http://www.csa.iisc.ernet.in/~uday/poly-transformations-intro.pdf`
  - Polyhedral.info `http://polyhedral.info`
- Tools/Infrastructure to try
  - Barvinok tool: `http://barvinok.gforge.inria.fr/`
  - Pluto `http://pluto-compiler.sourceforge.net` – use **pet** branch of **git** version
  - PPCG – Polyhedral parallel code generation for CUDA `http://repo.or.cz/ppcg.git`
  - Polly `http://polly.llvm.org`

Assignment 1

- Download PolyMage's **e0358** branch
  $ git clone https://bitbucket.org/udayb/polymage.git -b e0358
- Modify *sandbox/video_demo/harris_corner/harris_opt.cpp* to improve performance over harris_naive.cpp
- Test performance through the video demo (see README.md in *sandbox/video_demo/*
- Use any 1080p video for testing
- Either transform manually or consider using Barvinok (iscc): `http://barvinok.gforge.inria.fr/`
- Optimize for performance targeting 4 cores of a CL workstation
- **What to submit**: harris_opt.cpp and report.pdf, a report describing optimizations you performed, and the performance you observed (in ms) when running on **4 cores** of the CL workstation; also report execution times and scaling from 1 to 4 cores. Use the printout when you exit the video demo to report timing. Submit by email in a single compressed tar file named <your name>.tar.gz
- **Deadline: Fri Oct 7, 4:59pm**

Outline

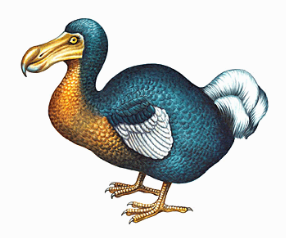
1. Introduction, Motivation, and Foundations
2. Optimizations for Parallelism, Locality and More
   - Polyhedral Framework
   - Affine Transformations
   - Tiling
   - Concurrent Start in Tiled Spaces
3. High-Performance DSL Compilation
   - Image Processing Pipelines
   - Solving PDEs Numerically
   - Deep Neural Networks
4. Conclusions

Domain-Specific Languages

- Standalone DSLs: own syntax
- **Embedded DSLs: embedded in/hosted by an existing language**

Domain-Specific Languages

- Standalone DSLs: own syntax
- **Embedded DSLs: embedded in/hosted by an existing language**
- **Arguments against DSLs**
  - Too specialized
  - Need to learn a new language!



A Dodo (highly specialized, but extinct)

- Standalone DSLs: own syntax
- **Embedded DSLs: embedded in/hosted by an existing language**
- **Arguments against DSLs**
  - Too specialized
  - Need to learn a new language!

  But
  - DSLs can be embedded in existing languages
  - Can grow and become more general-purpose

A Dodo (generalized)

DSL Compilation

- Frameworks studied for general-purpose languages/compilation can be reused
- **Customized optimization strategies necessary**
- Examples of high-performance DSLs: SPIRAL, Green-Marl, Halide, PolyMage, SystemML

Programming/Compiler Technologies for Emerging Domains

- **Catch 22**
  - Progress requires the right programming, compiler, and hardware technologies
  - Architects of programming, compiler, and hardware technologies cannot build these unless they know what the domain experts want
- **Tough problem: solutions?**

Programming/Compiler Technologies for Emerging Domains

- **Catch 22**
  - Progress requires the right programming, compiler, and hardware technologies
  - Architects of programming, compiler, and hardware technologies cannot build these unless they know what the domain experts want
- **Tough problem: solutions?**
  - Get lucky with the right hardware / primitives (Deep learning? — relies on BLAS, FFT)
  - Work closely with domain scientists
  - Domain scientist does both

Outline

Where are Image Processing Pipelines used?

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

Google+ Auto Enhance

Graphs of interconnected processing stages

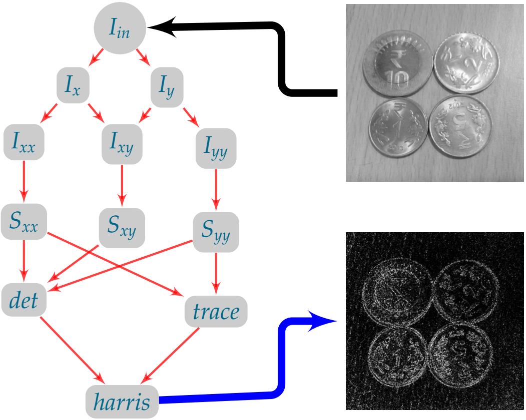
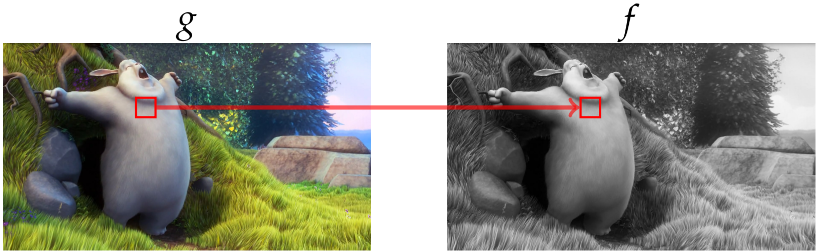


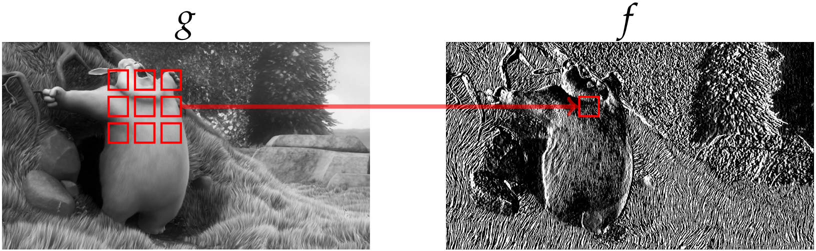
Figure: **Harris corner detection**



Point-wise

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

Computation Patterns



Stencil

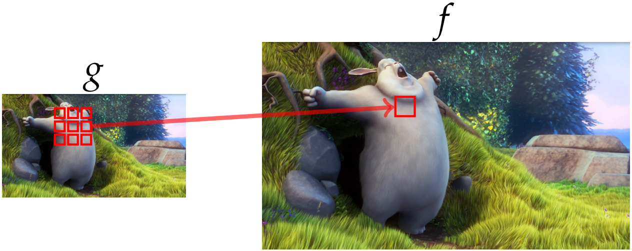
$$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x+\sigma_x, y+\sigma_y) \cdot w(\sigma_x, \sigma_y)$$

Computation Patterns

Downsample

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

Computation Patterns



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

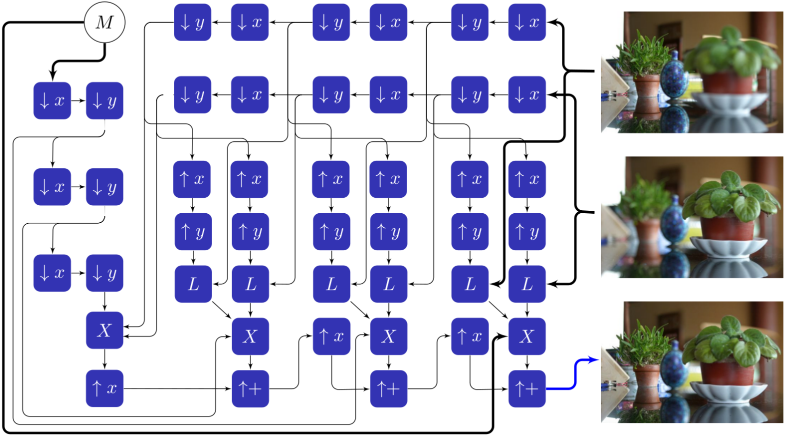
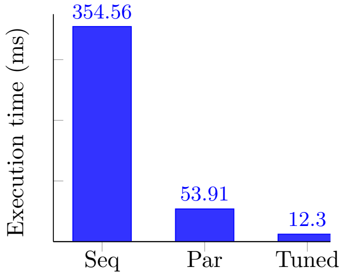


Image courtesy: Kyros Kutulakos

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

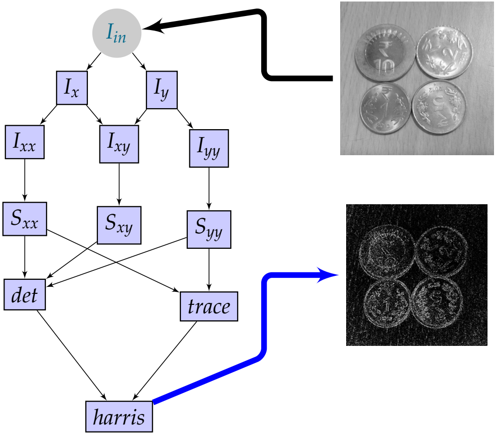
- **Manually optimizing pipelines is hard**
- **Goal: Performance levels of manual tuning without the pain**

A DSL Approach

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

Embedded DSL — an example

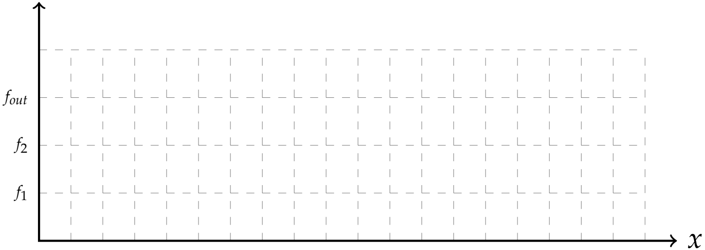
```
R, C = Parameter(Int), Parameter(Int)
I = Image(Float, [R+2, C+2])

x, y = Variable(), Variable()
row, col = Interval(0,R+1,1), Interval(0,C+1,1)

c = Condition(x,'>=',1) & Condition(x,'<=',R) &
    Condition(y,'>=',1) & Condition(y,'<=',C)

cb = Condition(x,'>=',2) & Condition(x,'<=',R-1) &
     Condition(y,'>=',2) & Condition(y,'<=',C-1)

Iy = Function(varDom = ([x,y],[row,col]),Float)
Iy.defn = [ Case(c, Stencil(I(x,y), 1.0/12,
                 [[-1, -2, -1],
                  [ 0,  0,  0],
                  [ 1,  2,  1]]) ]

Ix = Function(varDom = ([x,y],[row,col]),Float)
Ix.defn = [ Case(c, Stencil(I(x,y), 1.0/12,
                 [[-1, 0, 1],
                  [-2, 0, 2],
                  [-1, 0, 1]]) ]

Ixx = Function(varDom = ([x,y],[row,col]),Float)
Ixx.defn = [ Case(c, Ix(x,y) * Ix(x,y)) ]

Iyy = Function(varDom = ([x,y],[row,col]),Float)
Iyy.defn = [ Case(c, Iy(x,y) * Iy(x,y)) ]

Ixy = Function(varDom = ([x,y],[row,col]),Float)
Ixy.defn = [ Case(c, Ix(x,y) * Iy(x,y)) ]

Sxx = Function(varDom = ([x,y],[row,col]),Float)
Syy = Function(varDom = ([x,y],[row,col]),Float)
Sxy = Function(varDom = ([x,y],[row,col]),Float)
for pair in [(Sxx, Ixx), (Syy, Iyy), (Sxy, Ixy)]:
    pair[0].defn = [ Case(cb, Stencil(pair[1], 1,
                         [[1, 1, 1],
                          [1, 1, 1],
                          [1, 1, 1]]) ]

det = Function(varDom = ([x,y],[row,col]),Float)
d = Sxx(x,y) * Syy(x,y) - Sxy(x,y) * Sxy(x,y)
det.defn = [ Case(cb, d) ]

trace = Function(varDom = ([x,y],[row,col]),Float)
trace.defn = [ Case(cb, Sxx(x,y) + Syy(x,y)) ]

harris = Function(varDom = ([x,y],[row,col]),Float)
coarsity = det(x,y) - .04 * trace(x,y) * trace(x,y)
harris.defn = [ Case(cb, coarsity) ]
```



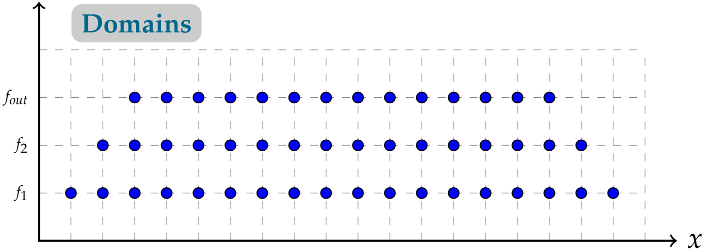
Embedded in Python

Functional, domain-level operations

Polyhedral Representation

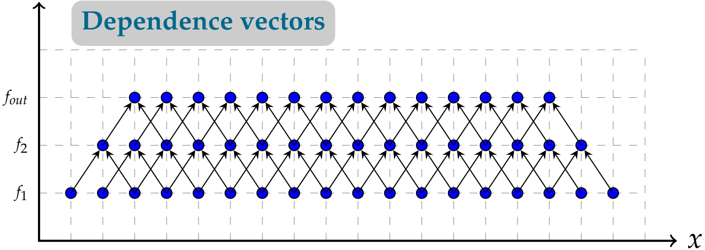



```
x = Variable()
f_in = Image(Float, [18])
f_1 = Function(varDom = ([x], [Interval(0, 17, 1)]), Float)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(varDom = ([x], [Interval(1, 16, 1)]), Float)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(varDom = ([x], [Interval(2, 15, 1)]), Float)
f_out.defn =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



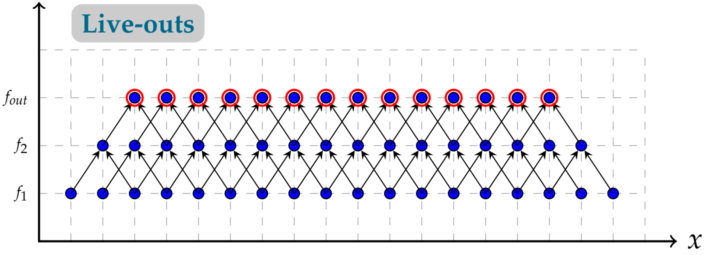

```
x = Variable()
f_in = Image(Float, [18])
f_1 = Function(varDom = ([x], [Interval(0, 17, 1)]), Float)
f_1.defn = [ f_in(x) + 1 ]
f_2 = Function(varDom = ([x], [Interval(1, 16, 1)]), Float)
f_2.defn = [ f_1(x-1) + f_1(x+1) ]
f_out = Function(varDom = ([x], [Interval(2, 15, 1)]), Float)
f_out.defn =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

Polyhedral Representation



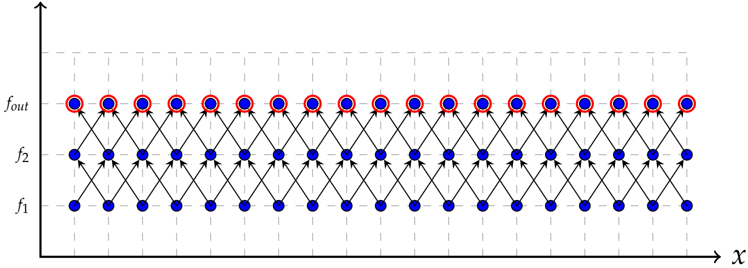
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1), (1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1), (1,-1)$
$f_1(x) = f_{in}(x)$	

Polyhedral Representation



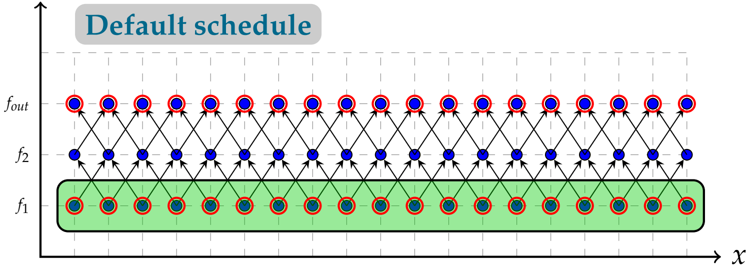
Function	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(1,1), (1,-1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(1,1), (1,-1)$
$f_1(x) = f_{in}(x)$	

Scheduling Techniques



Parallelism **Locality** **Storage**

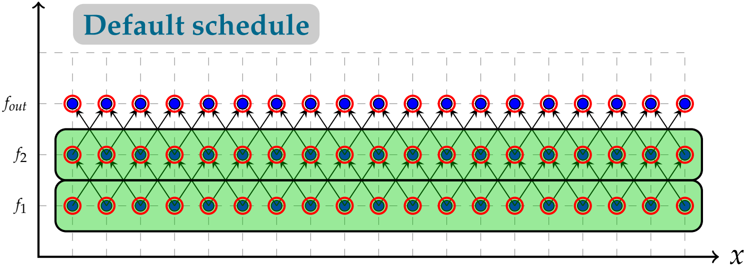
Scheduling Techniques



Parallelism

Locality

Storage

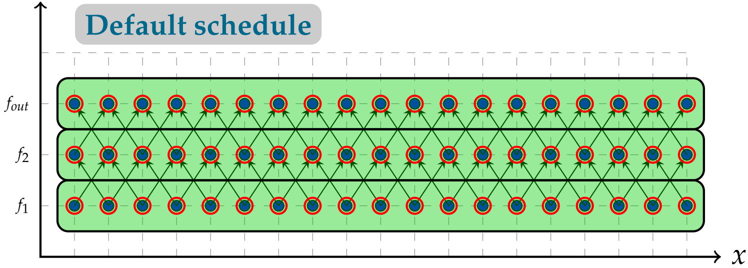


Parallelism

Locality

Storage

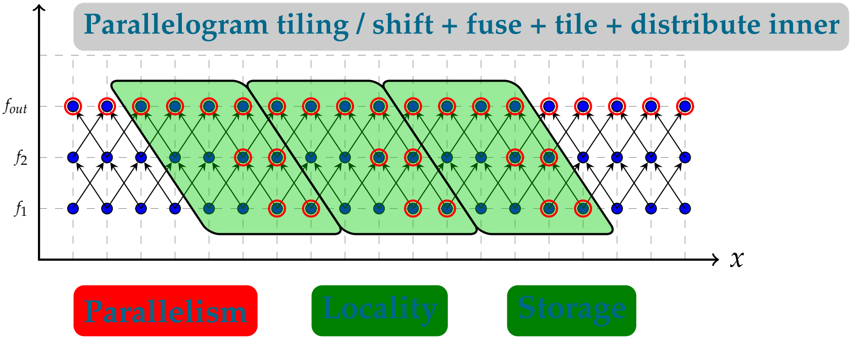
Scheduling Techniques



Parallelism Locality Storage

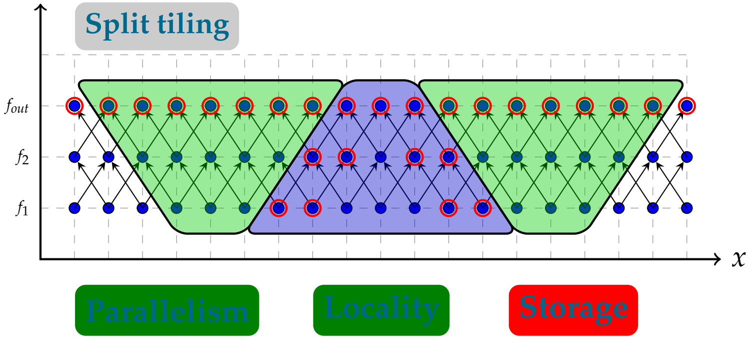
- Load balanced parallelization
- But does not exploit locality

Scheduling Techniques



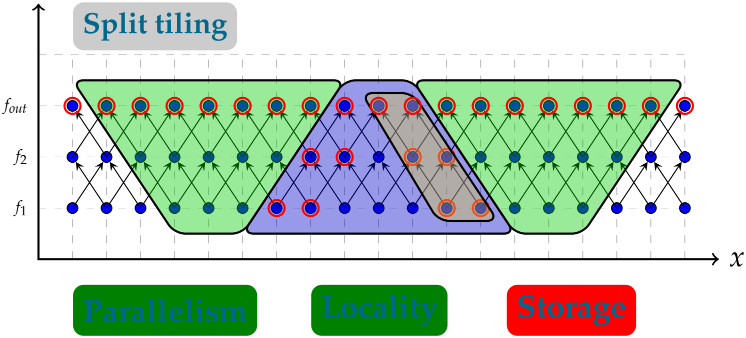
- Loss of parallelism (for a coarse-grained mapping)
- (or) High synchronization ($\frac{3N}{32}$ synchronizations!) for a fine-grained one

Scheduling Techniques



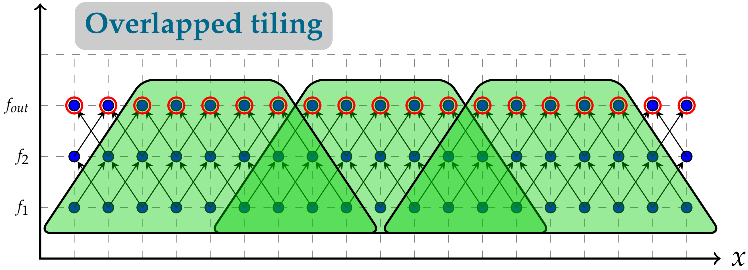
- *Split tiling for GPUs*: Grosser et al. GPGPU 2013
- Similar scheme also used in Pochoir [Tang et al. SPAA 2011]

SCHEDULING TECHNIQUES



- Data is live out of left and right boundaries (in addition to top)
  - Local buffering (scratchpads for tiles) is difficult!

Scheduling Techniques



Parallelism

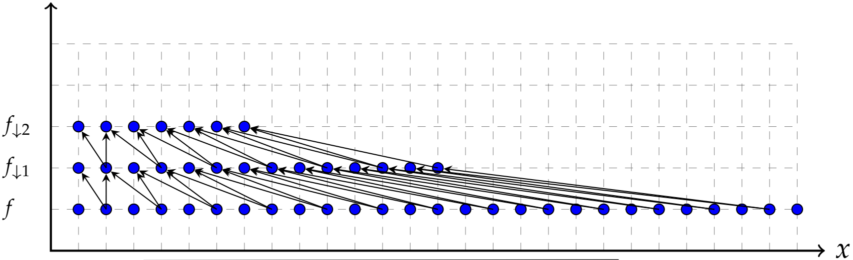
Locality

Storage

Re-computation

- Break dependence at boundaries through redundant computation

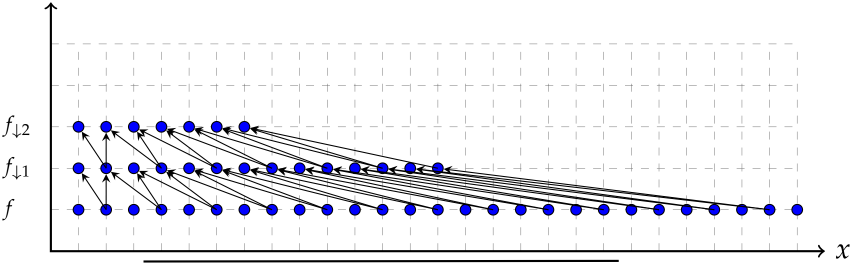
Overlapped Tiling for Heterogeneous Functions



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

- **Some approaches to overlapped tiling only consider homogeneous time-iterated stencils**

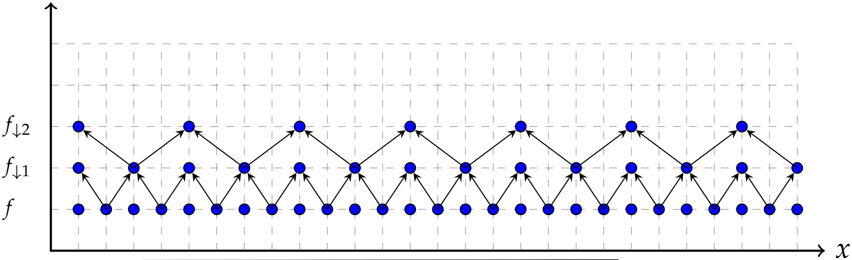
Overlapped Tiling for Heterogeneous Functions



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**

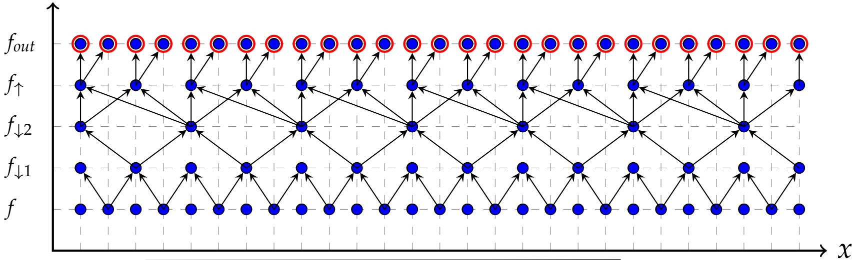
Overlapped Tiling for Heterogeneous Functions



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

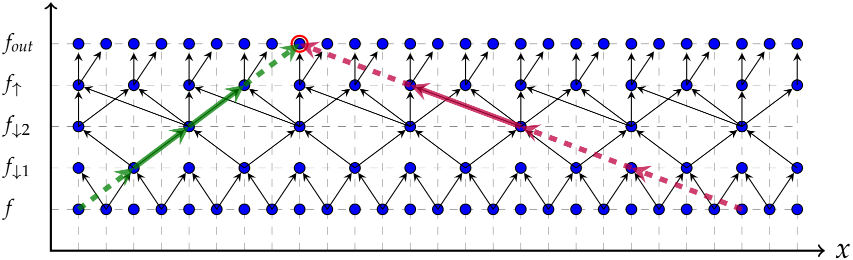
- **Scaling and aligning the schedules**

Overlapped Tiling for Heterogeneous Functions



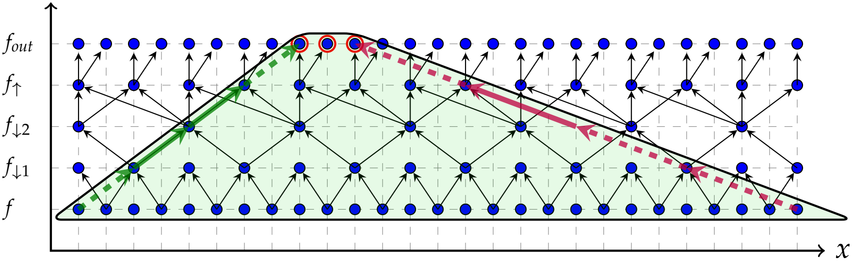
Function	**Schedule**
$f_{out}(x) = f_{\uparrow}(x/2)$	$(x) \to (4, x)$
$f_{\uparrow}(x) = f_{\downarrow 2}(x/2) \cdot f_{\downarrow 2}(x/2 + 1)$	$(x) \to (3, 2x)$
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x - 1) \cdot f_{\downarrow 1}(2x + 1)$	$(x) \to (2, 4x)$
$f_{\downarrow 1}(x) = f(2x - 1) \cdot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

Overlapped Tiling for Heterogeneous Functions



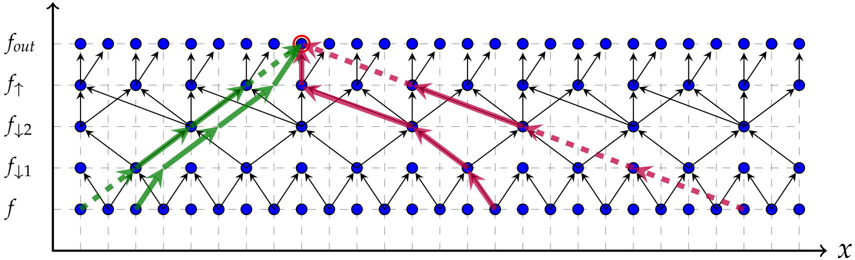
- **Determining tile shape**
- **Conservative vs precise bounding faces**

Overlapped Tiling for Heterogeneous Functions



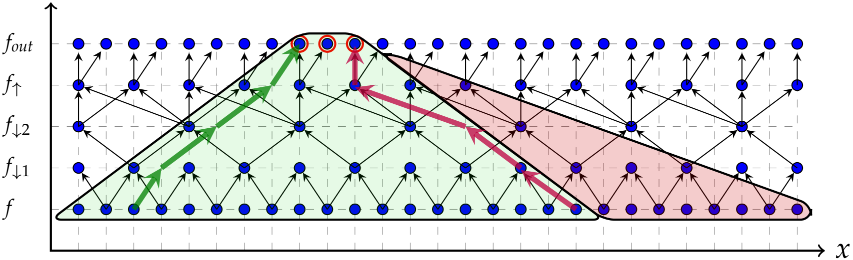
- **Determining tile shape**
- **Conservative vs precise bounding faces**

Overlapped Tiling for Heterogeneous Functions



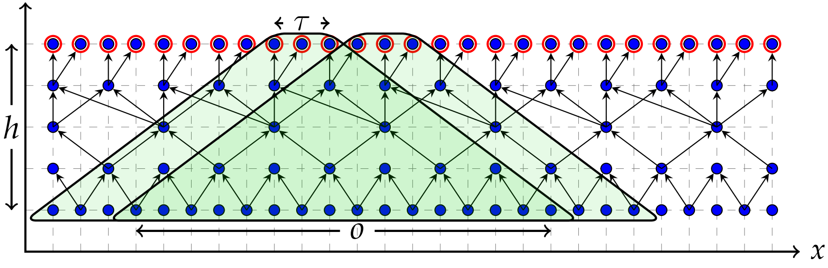
- **Determining tile shape**
- **Conservative vs precise bounding faces**

Overlapped Tiling for Heterogeneous Functions



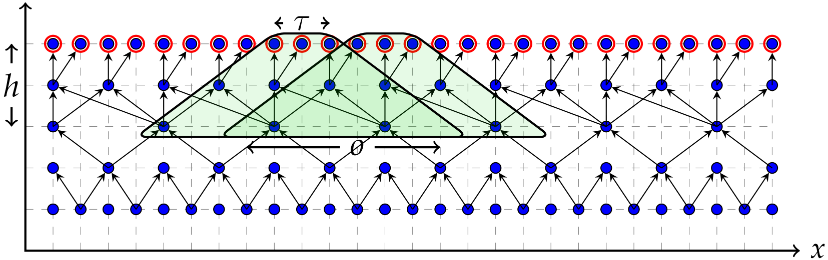
- **Significant reduction in redundant computation**

Overlapped Tiling for Heterogeneous Functions



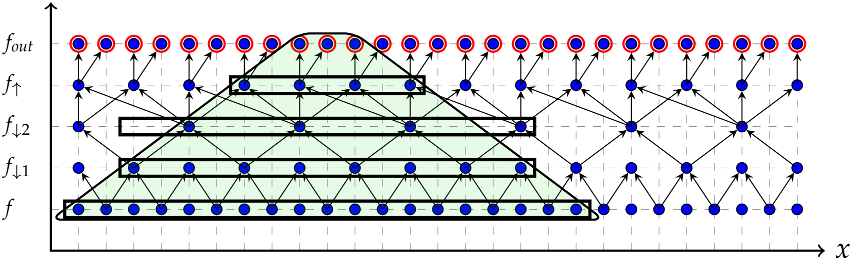
- **Tile size τ, overlap O, height h**
- **Trade-off between fusion height and overlap**
- **More fusion provides more locality, but also a greater fraction of redundant computation**

Overlapped Tiling for Heterogeneous Functions



- **Tile size τ, overlap O, height h**
- **Trade-off between fusion height and overlap**
- **More fusion provides more locality, but also a greater fraction of redundant computation**

Overlapped Tiling for Heterogeneous Functions



Scratchpads

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

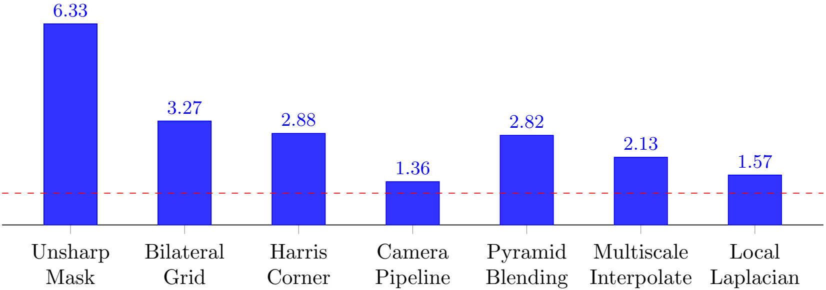
Some Benchmarks in this Domain

Seven 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

- Video demo

Speedup of grouped and tiled implementations over naively parallelized and vectorized ones

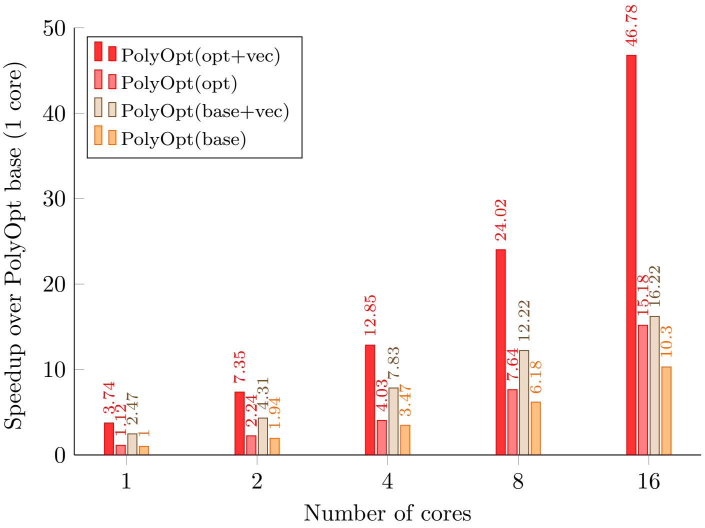


16 threads and vectorization enabled

On a 2-socket 16-core Intel Xeon SandyBridge

Source: [Mullapudi et al. ASPLOS 2015 PolyMage]

A Deeper Look: Harris Corner Detection



Source: PolyMage, Mullapudi et al. ASPLOS 2015

References

- Delite: A compiler/runtime framework for embedded DSLs
  `http://stanford-ppl.github.io/Delite/` (read papers)
- Halide `http://halide-lang.org` (tutorial and code)
  `http://halide-lang.org/cvpr2015.html`
- PolyMage:
  `http://mcl.csa.iisc.ernet.in/polymage.html` (code, slides, and paper)
  Mullapudi et al. Automatic Optimization of Image Processing Pipelines, ASPLOS 2015.

OUTLINE

1. Introduction, Motivation, and Foundations
2. Optimizations for Parallelism, Locality and More
   - Polyhedral Framework
   - Affine Transformations
   - Tiling
   - Concurrent Start in Tiled Spaces
3. High-Performance DSL Compilation
   - Image Processing Pipelines
   - Solving PDEs Numerically
   - Deep Neural Networks
4. Conclusions

Solving Partial Differential Equations Numerically

- A number of science and engineering problems involve solving a partial differential equation (PDE)
- Numerous techniques exist varying in computational complexity, convergence properties, amenability to optimization
- A discretization strategy is chosen first
  1. **Finite difference**
  2. Finite volume
  3. Finite element

Example: Poisson's Equation

Poisson's equation – the mother of all PDEs:

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

Example: Poisson's Equation

Poisson's equation – the mother of all PDEs:

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

- Approximate the second derivative (Laplacian) using finite difference. Eg: for a 2-d grid,

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

Example: Poisson's Equation

Poisson's equation – the mother of all PDEs:

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

- Approximate the second derivative (Laplacian) using finite difference. Eg: for a 2-d grid,

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

- We are solving $y = Ax$, where A is a sparse banded matrix (x is a linearization of the unknown on the multi-dimensional grid)
- 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

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

Hierarchical Mesh Structure

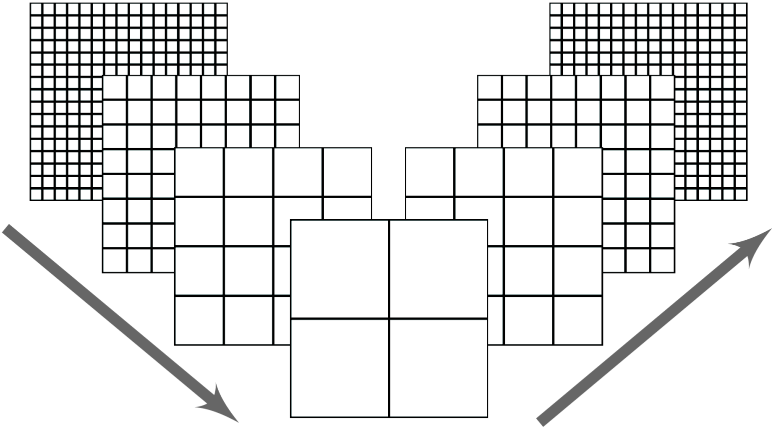


Figure: Hierarchical mesh structure for Multigrid levels

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*

$e^{2h} \leftarrow 0$

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

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

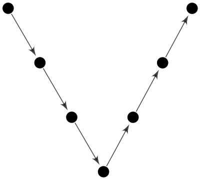
$v^h \leftarrow v^h + e^h$ *// correction*

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

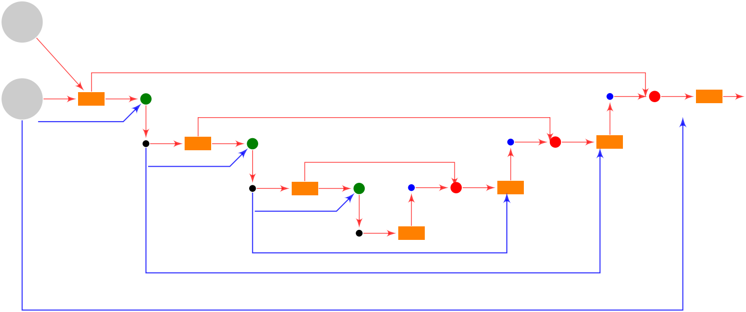
return v^h

- Animation

MULITIGRID V-CYCLE

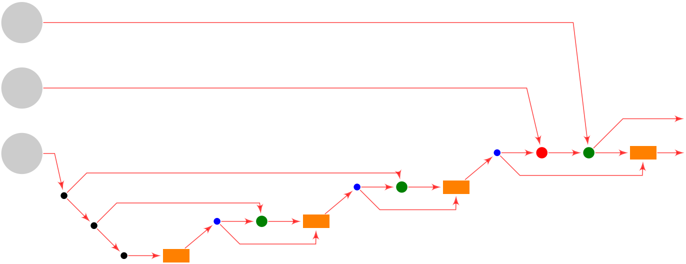


(a) V-cycle



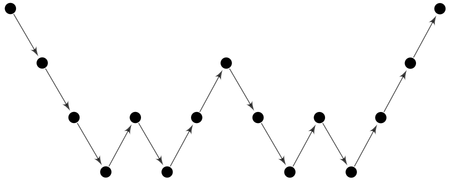
(b) V-cycle: complete DAG

NAS MG V-CYCLE

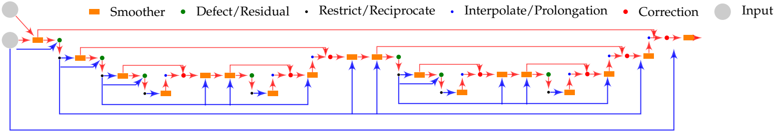


(c) NAS-PB MG V-cycle

Multigrid W-cycle



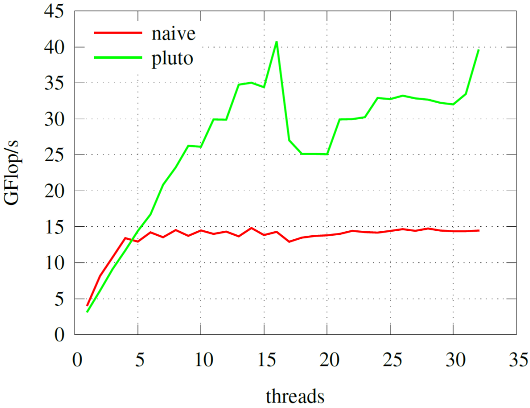
(d) W-cycle



(e) W-cycle: complete DAG

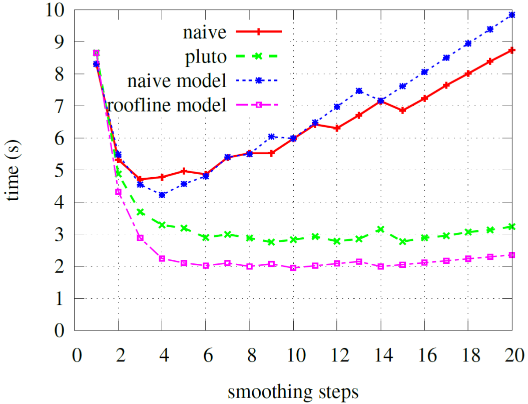
Figure: DAG representation of (a) V-cycle and (b) W-cycle

GMG: Smoother Scaling



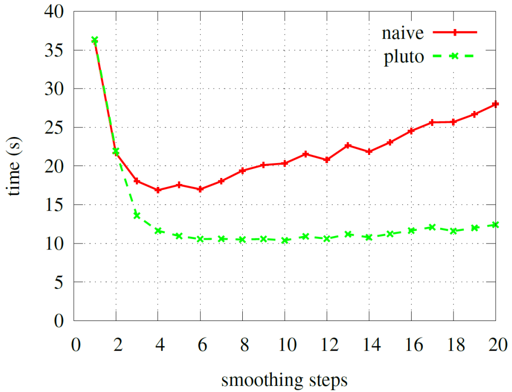
Scalability of 10 iterations of the Jacobi smoother on an 8000^2 domain on a 16-core Intel Sandy Bridge
Source: Ghysels (LBNL) and Vanroose (University of Antwerp) SIAM J. Scientific Computing 2015

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 (University of Antwerp) SIAM J. Scientific Computing 2015

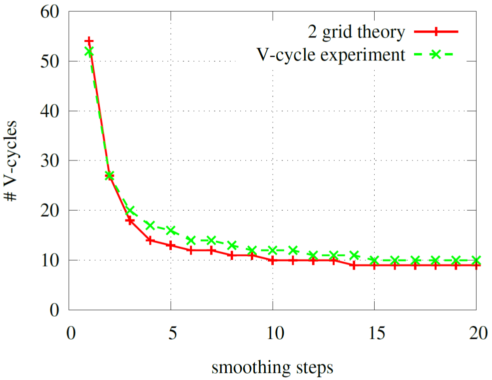
GMG: EXECUTION TIME (3-D)



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 (University of Antwerp) SIAM J. Scientific Computing 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 (University of Antwerp) SIAM J. Scientific Computing 2015

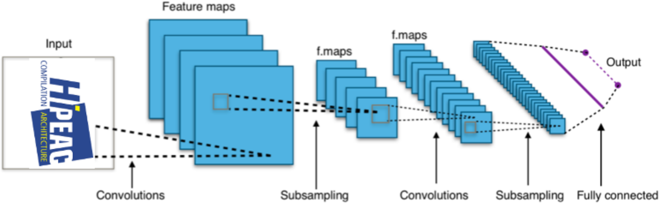
References

- P. Ghysels and W. Vanroose, Modeling the performance of geometric multigrid on many-core computer architectures, SIAM J. Scientific Computing (2015).
- Knabner P, Angerman L. Numerical Methods for Elliptic and Parabolic Partial Differential Equations. Texts in Applied Mathematics, Springer, 2003.
- Saad Y. Iterative Methods for Sparse Linear Systems, Second Edition. SIAM: Philadelphia, 2003.

OUTLINE

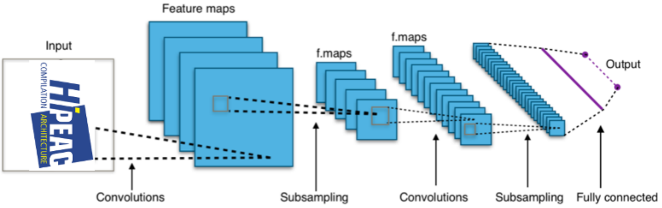
Deep Convolutional Neural Networks

- Shown to be effective in image classification, speech recognition, and at many more tasks
- A domain currently of high interest



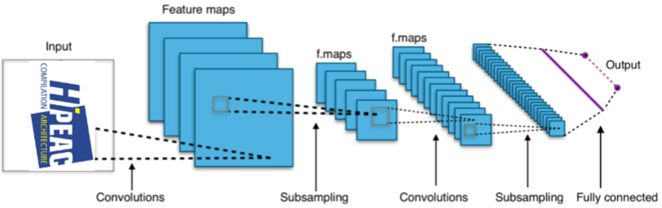
Deep Convolutional Neural Networks

- Shown to be effective in image classification, speech recognition, and at many more tasks
- A domain currently of high interest
- **Training these networks requires HPC**!
- **Inference requires high performance or real-time** response



Deep Convolutional Neural Networks

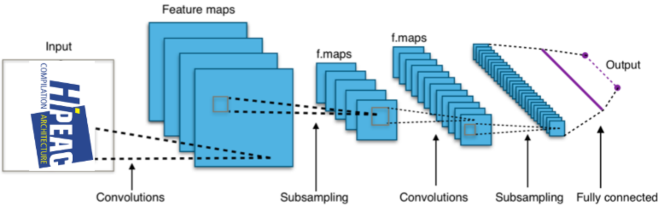
- **Training these networks requires HPC!**
- **Inference requires high performance or real-time** response



- The network is trained by sending through training data (in batches) forward and then backward, multiple times

Deep Convolutional Neural Networks

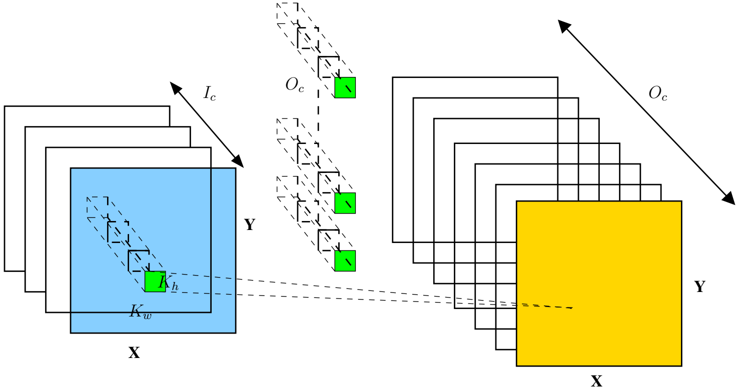
- **Training these networks requires HPC!**
- **Inference requires high performance or real-time** response



- The network is trained by sending through training data (in batches) forward and then backward, multiple times
- Extremely compute intensive!
- Think about running numerous matrix-matrix multiplications in parallel (with all of them sharing data along multiple dimensions)

CNN Convolution as a loop nest

```
for (n = 0; n < N; n++) /* Samples in a batch */
 for (o = 0; o < Oc; o++) /* Output feature channels */
  for (i = 0; i2 < Ic; i++) /* Input feature channels */
   for (y = 0; i3 < Y; i3++) /* Layer height */
    for (x = 0; i4 < X; i4++) /* Layer width */
     for (kh = 0; i5 < Kh; i5++) /* Convolution kernel height */
      for (kw = 0; i6 < Kw; i6++) /* Convolution kernel width */
       output[n, o, y, x] += input[n, i, y+kh, x+kw] * weights[o, i, kh, kw];
```



CNN Convolution as a loop nest

```
for (n = 0; n < N; n++) /* Samples in a batch */
  for (o = 0; o < Oc; o++) /* Output feature channels */
    for (i = 0; i2 < Ic; i++) /* Input feature channels */
      for (y = 0; i3 < Y; i3++) /* Layer height */
        for (x = 0; i4 < X; i4++) /* Layer width */
          for (kh = 0; i5 < Kh; i5++) /* Convolution kernel height */
            for (kw = 0; i6 < Kw; i6++) /* Convolution kernel width */
              output[n, o, y, x] += input[n, i, y+kh, x+kw] * weights[o, i, kh, kw];
```

1. **Abundant parallelism**
   - Batch-level parallelism (N)
   - Parallelism from feature channels and layer (Y, X, Oc)
   - Parallelism when using BLAS calls?
2. **Locality?**
   - *output*: reuse along i, kh, kw
   - *input*: reuse along o (along kh, kw as well if no replicate)
   - *weights* (reuse along n, y, x)
   - In addition, multiple convolutions performed successively
3. **Data allocation, layout, and management?**

Optimizing CNNs

- High-dimensional iteration spaces, high-dimensional arrays
- **A playground for optimization**
- Parallelization, locality optimization, data allocation / layout optimization, computation reduction?
- Take advantage of existing vendor libraries (MKL, CuDNN)
- New CNN and other DNN architectures, very deep neural networks, upcoming parallel architectures

CNNs: State-of-the-Art

- GPUs are used: NVIDIA CuDNN provides tuned primitives for well-known/widely used layers (convolutions, max pooling)
- Caffe (C++-based), Torch (Lua), Theano (Python), TensorFlow (Python) are library-based approaches that wrap around calls to libraries (CuDNN)

CNNs: State-of-the-Art

- GPUs are used: NVIDIA CuDNN provides tuned primitives for well-known/widely used layers (convolutions, max pooling)
- Caffe (C++-based), Torch (Lua), Theano (Python), TensorFlow (Python) are library-based approaches that wrap around calls to libraries (CuDNN)
- State-of-the-art implementations sustain excellent performance on GPUs
  On an NVIDIA GeForce Titan X with a peak of 6.97 TFLOPS (single-precision), VGGNet network E with fp32 data, NVIDIA CuDNN v3 obtains 44% and 90% of machine peak respectively for N=1 and N=64.
- **What will the role of DSL compilers and code generators be?**

References

1. *Coarse grain parallelization of deep neural networks*, Marc Gonzalez Tallada, PPoPP 2016
2. *Latte: a language, compiler, and runtime for elegant and efficient deep neural networks*, Truong et al. PLDI 2016
3. *Fast Algorithms for Convolutional Neural Networks*, Andrew Lavin, Scott Gray, Nov 2015 `http://arxiv.org/abs/1509.09308`

Outline

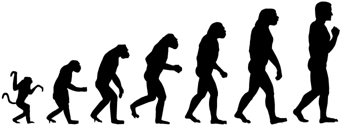
Takeaways for the Domains Presented

- The presented domains have abundant parallelism, reuse, and optimization opportunity
- There is more parallelism than the number of processors
- One may be ultimately memory bandwidth bound (even after optimization) on a large number of cores
- A naive parallelization is often easy
- **But while parallelizing**, pay attention to:
  - Tiling for locality
  - Fusion
  - Synchronization costs
  - Local buffering (easier/feasible in DSL compilation)

Big Picture: Role of Compilers

General-purpose: EVOLUTIONARY

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Limited improvements but wide impact



Domain-specific: REVOLUTIONARY

- Build new **domain-specific languages and compilers**
- Dramatic speedups

1. **Important to pursue both**
2. **Need to build reusable infrastructure to share among various DSLs**
3. **Reduce multiplicity of DSL environments**

CONCLUSIONS

- **Tremendous opportunities in high-performance compilation — both domain-specific and general-purpose**
- **Several emerging domains that require high-performance compilation**
  - **will impact both embedded and big data crunching architectures**
- **These domains are a perfect fit for HiPEAC (eg: high-performance embedded vision)**

Thank You!