Iterator-Based Optimization of Imperfectly-Nested Loops

DANIEL FESHBACH, MARY GLASER, MICHELLE STROUT, DAVID WONNACOTT

Overview

- Motivation: Approaches to Performance Tuning
- Quick overview of Polyhedral Model
- Quick review of Chapel Iterators
- Detailed Discussion of Deriche Image Processing Example
- Details of Nussinov are in paper (and past work)
- Details of FFT may be in future paper (we hope)

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...

```
// Example (benchmark, simplified Physics)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
  - Replace % operation with bit-mask, or hoist out of loop
  - Perform loop tiling to improve memory performance
  - Perform loop skewing to ensure loop tiling is legal
  - Also introduce vector instructions

```
// Example (benchmark, simplified Physics)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...

- Compiler-writer: this is a compiler problem, fix compiler
  - Replace % operation with bit-mask, or hoist out of loop
  - Perform loop tiling to improve memory performance
  - Perform loop skewing to ensure loop tiling is legal
  - Also introduce vector instructions
  - Then, update compiler to tile for multicore systems
  - Then, write another compiler for distributed memory
  - Then, write another compiler for GPGPU's

```
// Example (benchmark, simplified Physics)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
  - Replace % operation with bit-mask, or hoist out of loop
  - Perform loop tiling to improve memory performance
  - Perform loop skewing to ensure loop tiling is legal
  - Also introduce vector instructions
  - Then, update compiler to tile for multicore systems
  - Then, write another compiler for distributed memory
  - Then, write another compiler for GPGPU's

```
// Example (benchmark, simplified Physics)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
         (A[t%2,x-1,y] + A[t%2,x,y-1] +
          A[t%2,x  ,y] + A[t%2,x,y+1] +
          A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...

- Compiler-writer: this is a compiler problem, fix compiler
  - Replace % operation with bit-mask, or hoist out of loop
  - Perform loop tiling to improve memory performance
  - Perform loop skewing to ensure loop tiling is legal
  - Also introduce vector instructions
  - Then, update compiler to tile for multicore systems
  - Then, write another compiler for distributed memory
  - Then, write another compiler for GPGPU's

```
// Example (benchmark, simplified Physics)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student

```
// Example (actual code is more complex)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student
  - Grad student replaces or hoists %

```
// Example (actual code is more complex)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[t&1, x, y] =  // t&1 == t%2
        (A[1-(t&1),x-1,y]+A[1-(t&1),x,y-1]+
         A[1-(t&1),x  ,y]+A[1-(t&1),x,y+1]+
         A[1-(t&1),x+1,y]) / 5;
      // note: t%2 stores two time steps
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code…
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student
  - Grad student replaces or hoists %
  - Grad student may have heard of loop tiling, may try it

```
// Example (actual code is more complex)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

// Loop over tile wavefronts.
for kt in ceild(3,tau) .. floord(3*T,tau) {
  // The next two loops iterate within a tile wavefront.
  // Assumes a square iteration space.
  var k1_lb: int = floord(3*L+2+(kt-2)*tau, tau*3);
  var k1_ub: int = floord(3*U+(kt+2)*tau-2, tau*3);
  var k2_lb: int = floord((2*kt-2)*tau-3*U+2, tau*3);
  var k2_ub: int = floord((2+2*kt)*tau-3*L-2, tau*3);

  // Loops over tile coordinates within a parallel wavefront of
  forall k1 in k1_lb .. k1_ub {
    for x in k2_lb .. k2_ub {
      var k2 = x-k1;
      // Loop over time within a tile.
      for t in max(1,floord(kt*tau,3)) .. min(T,floord((3+kt)*ta
        write = t & 1; // equivalent to t mod 2
        read = 1 - write;
        // Loops over the spatial dimensions within each tile.
        for i in max(L,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2
              .. min(U,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau
          for j in max(L,max(tau*k1-t,t-i-(1+k2)*tau+1))
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student
  - Grad student replaces or hoists %
  - Grad student may have heard of loop tiling, may try it
  - Grad student spends nights reading about vectorization

```
// Example (actual code is more complex)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

// Loop over tile wavefronts.
for kt in ceild(3,tau) .. floord(3*T,tau) {
  // The next two loops iterate within a tile wavefront.
  // Assumes a square iteration space.
  var k1_lb: int = floord(3*L+2+(kt-2)*tau, tau*3);
  var k1_ub: int = floord(3*U+(kt+2)*tau-2, tau*3);
  var k2_lb: int = floord((2*kt-2)*tau-3*U+2, tau*3);
  var k2_ub: int = floord((2+2*kt)*tau-3*L-2, tau*3);

  // Loops over tile coordinates within a parallel wavefront of tiles.
  forall k1 in k1_lb .. k1_ub {
    for x in k2_lb .. k2_ub {
      var k2 = x-k1;
      // Loop over time within a tile.
      for t in max(1,floord(kt*tau,3)) .. min(T,floord((3+kt)*tau-3,3))
        write = t & 1; // equivalent to t mod 2
        read = 1 - write;
        // Loops over the spatial dimensions within each tile.
        for i in max(L,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2))
              .. min(U,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)) {
          for j in max(L,max(tau*k1-t,t-i-(1+k2)*tau+1))
                .. min(U,min((1+k1)*tau-t-1,t-i-k2*tau)){
            A[write, x, y] =
              (A[read,x-1,y] + A[read,x,y-1] +
```

Basic Approaches to Code Optimization

- Performance tuning of compute-intensive code...
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student
  - Grad student replaces or hoists %
  - Grad student may have heard of loop tiling, may try it
  - Grad student spends nights reading about vectorization
  - Next grad students can work on multicore, cluster, and GPGPU versions
  - Formal or Ad-Hoc approach to software management

```
// Example (actual code is more complex)
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//     previous values of it and neighbors

// Loop over tile wavefronts.
for kt in ceild(3,tau) .. floord(3*T,tau) {
  // The next two loops iterate within a tile wavefront.
  // Assumes a square iteration space.
  var k1_lb: int = floord(3*L+2+(kt-2)*tau, tau*3);
  var k1_ub: int = floord(3*U+(kt+2)*tau-2, tau*3);
  var k2_lb: int = floord((2*kt-2)*tau-3*U+2, tau*3);
  var k2_ub: int = floord((2+2*kt)*tau-3*L-2, tau*3);

  // Loops over tile coordinates within a parallel wavefront of tiles.
  forall k1 in k1_lb .. k1_ub {
    for x in k2_lb .. k2_ub {
      var k2 = x-k1;
      // Loop over time within a tile.
      for t in max(1,floord(kt*tau,3)) .. min(T,floord((3+kt)*tau-3,3)){
        write = t & 1; // equivalent to t mod 2
        read = 1 - write;
        // Loops over the spatial dimensions within each tile.
        for i in max(L,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2))
              .. min(U,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)) {
          for j in max(L,max(tau*k1-t,t-i-(1+k2)*tau+1))
                .. min(U,min((1+k1)*tau-t-1,t-i-k2*tau)){
            A[write, x, y] =
              (A[read,x-1,y] + A[read,x,y-1] +
               A[read,x  ,y] + A[read,x,y+1] +
               A[read,x+1,y]) / 5;
```

Goal: Best of Both Worlds (in Chapel)

- Performance tuning of compute-intensive code…
- Compiler-writer: this is a compiler problem, fix compiler
- Physicist: this is a coding problem, give to grad student
- Can we combine the best elements of both worlds?
  - Think of compiler optimizer as a way to make *clean* code run fast (rather than a way to make *some* code run fast)
  - Not primarily a language/complier comparison, but Chapel does seem appealing
  - first, a quick review of technologies…

```
// Example
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for (t, x, y) in Jacobi2d(T, N, N) do


      A_2s[t+1, x, y] =
        (A_2s[t,x-1,y] + A_2s[t,x,y-1] +
         A_2s[t,x  ,y] + A_2s[t,x,y+1] +
         A_2s[t,x+1,y]) / 5;
      // note: A_2s stores two time steps
```

"Polyhedral Model" and Optimization

- Linear constraints on integer variables
  - Integer Linear Programming/Presburger Arithmetic
  - Efficient for **simple subscripts and loop bounds**

```
// Example success:
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
```

"Polyhedral Model" and Optimization

- Linear constraints on integer variables
  - Integer Linear Programming/Presburger Arithmetic
  - Efficient for **simple subscripts and loop bounds**

- Exact ***Instance-wise*** array dataflow
  - For any *execution* of a line, which execution(s) of which line(s) produce the value under what conditions?

    e.g., iteration (1, 5, 10), i.e. when t=7, x=5, y=10
      the algorithm writes to A[1, 5, 10]
      using values from iterations (0, 4, 10), (0, 5, 9),
                                   (0, 5, 10), (0,5,11),
                               and (0, 6, 10)
      if T-1 >= 1 and N-2 >= 5 and N-2 >= 10

```
// Example success:
// iterative Jacobi stencil

// Repeatedly update each A[i,j], based on
//    previous values of it and neighbors

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;
```

"Polyhedral Model" and Optimization

- Linear constraints on integer variables
  - Integer Linear Programming/Presburger Arithmetic
  - Efficient for **simple subscripts and loop bounds**

- Exact ***Instance-wise*** array dataflow
  - For any *execution* of a line, which execution(s) of which line(s) produce the value under what conditions?

- Extends beyond unimodular loop transformations by allowing **imperfectly nested loops**

```
// Example success:
// iterative Jacobi stencil

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2, x-1, y] + …)/5

// Equivalently, two arrays
for t in 0..T-1 do {
  for x in 1..N-2 do
    for y in 1..N-2 do
      new_A[x, y] =
           (A[x-1, y] + …)/5
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[x, y] = new_A[x, y]
```

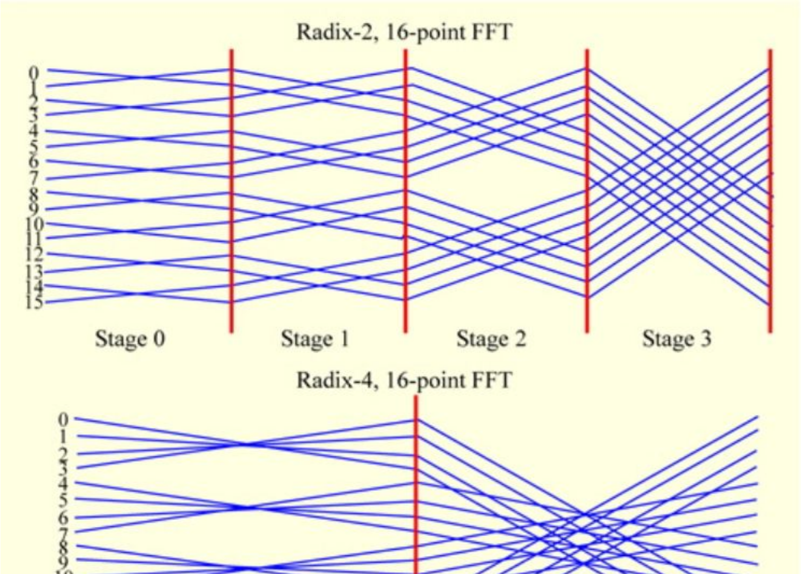
"Polyhedral Model" and Optimization

- Linear constraints on integer variables
  - Integer Linear Programming/Presburger Arithmetic
  - Efficient for **simple subscripts and loop bounds**
- Exact ***Instance-wise*** array dataflow
  - For any *execution* of a line, which execution(s) of which line(s) produce the value under what conditions?
- Extends beyond unimodular loop transformations by allowing **imperfectly nested loops**
- Allows search for/deduction of efficient schedules for *many* codes, **but**…

```
// Example success:
// iterative Jacobi stencil

for t in 0..T-1 do
  for x in 1..N-2 do
    for y in 1..N-2 do
      A[(t+1)%2, x, y] =
        (A[t%2,x-1,y] + A[t%2,x,y-1] +
         A[t%2,x  ,y] + A[t%2,x,y+1] +
         A[t%2,x+1,y]) / 5;


// Also handles imperfectly-nested code
//  that builds new_A[i,j] from A[i,j]
//  and neighbors (identical dataflow)
```

"Polyhedral Model" Limitations

- Allows search for/deduction of efficient schedules for many codes, but limited success with
  - **non-linear** subscript expressions,
    - e.g. fast fourier transform (Dataflow figure from Polat, Gokhan, et al., 2015, ETRI Journal, vol. 37, no. 4,)




```
// FFT
proc butterfly(wk1, wk2, wk3, X:[?D]) {
  const i0 = D.low,
        i1 = i0 + D.stride,
        i2 = i1 + D.stride,
        i3 = i2 + D.stride;
  var x0 = X(i0) + X(i1),
      x1 = X(i0) - X(i1),
      x2 = X(i2) + X(i3),
      x3rot = (X(i2) - X(i3))*1.0i;

  X(i0) = x0 + x2;
  x0 -= x2;
  X(i2) = wk2 * x0;

/// etc...
```

"Polyhedral Model" Limitations

- Allows search for/deduction of efficient schedules for many codes, but limited success with
  - **non-linear** subscript expressions,
    - e.g. fast fourier transform
  - cases **outside assumptions/search-space** of optimizer
    - e.g. Nussinov's Algorithm (or Zuker's)

```
// Nussinov's Algorithm
//  for predicting RNA secondary structure

for i in 1..(size-1) do
  for j in (size-i)..(size-1) do {
    for k in (size-1-i)..(j-1) do
      N[size-1-i,j] = max(N[size-1-i,j],
                          N[size-1-i,k]+N[k+1,j]);
    N[size-i-1,j] = max(N[size-i-1,j],
                        N[size-i,j-1]+ /* ... */);
  }
```

"Polyhedral Model" Limitations

- Allows search for/deduction of efficient schedules for many codes, but limited success with
  - **non-linear** subscript expressions,
    - e.g. fast fourier transform
  - cases **outside assumptions/search-space** of optimizer
    - e.g. Nussinov's Algorithm (or Zuker's)
  - cases that require **data-space** transformation
    - e.g. Deriche image filtering algorithm

```
// Deriche.c [YP15], Chapel-ized [Glaser '18]
for i in 0..w-1 {
  ym1 = 0.0;  ym2 = 0.0; xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] =
a1*imgIn[i,j]+a2*xm1+b1*ym1+b2*ym2;
    xm1=imgIn[i,j]; ym2=ym1; ym1=y1[i,j]; }}

// for i in 0..w-1
//   for j in 0..h-1 by -1
//      build y2, similarly to above

for i in 0..w-1
  for j in 0..h-1
     imgOut[i,j] = c1 * (y1[i,j] + y2[i,j]);

// three j/i loop nests for horizontal sweep
```

Varying Execution Order without Iterators

Faster in Machine 1:

```
for row = 1 to N {
    for col = 1 to N {
        a[row][col] = b[row][col] + c[row][col]; }}
```

Faster in Machine 2:

```
for col = 1 to N {
    for row = 1 to N {
        a[row][col] = b[row][col] + c[row][col]; }}
```

```
if Machine 1 {
    for row = 1 to N {
        for col = 1 to N {
            a[row][col] = b[row][col] + c[row][col]; }}}
else if Machine 2 {
    for col = 1 to N {
        for row = 1 to N {
            a[row][col] = b[row][col] + c[row][col]; }}}
```

Iterators to Abstract Execution Order

```
iter rowMajor(N) {
    for row = 1 to N {
        for col = 1 to N {
            yield (row, col); }}}

iter colMajor(N) {
    for col = 1 to N {
        for row = 1 to N {
            yield (row, col); }}}
```

```
if Machine 1 {
    bestMatrixOrder = rowMajor;
}
else if Machine 2 {
    bestMatrixOrder = colMajor;
}

for (row, col) in bestMatrixOrder(N) {
    a[row][col] = b[row][col] + c[row][col]; }
```

Iterators for Imperfect Loop Nests

```
for i in 0..w-1 {
  ym1 = 0.0; ym2 = 0.0; xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] = a1*imgIn[i,j] + a2*xm1
              + b1*ym1 + b2*ym2;
    xm1 = imgIn[i,j];   ym2 = ym1;   ym1 = y1[i,j];   }}
```

```
iter ij_forwards(w: int, h: int): (int, int, int) {
  for i in 0..w-1 {
    yield (i, 0, -9999);
    for j in 0..h-1
      yield (i, 1, j);   }}
```

```
for (i, statement, j) in ij_forwards(w, h) {
  if (statement == 0) {
    ym1 = 0.0; ym2 = 0.0; xm1 = 0.0;  }
  else if (statement == 1) {
    y1[i,j] = a1*imgIn[i,j] + a2*xm1
      + b1*ym1 + b2*ym2;
    xm1 = imgIn[i,j];   ym2 = ym1;   ym1 = y1[i,j];  }}
```

Iteration-Space Performance Tuning

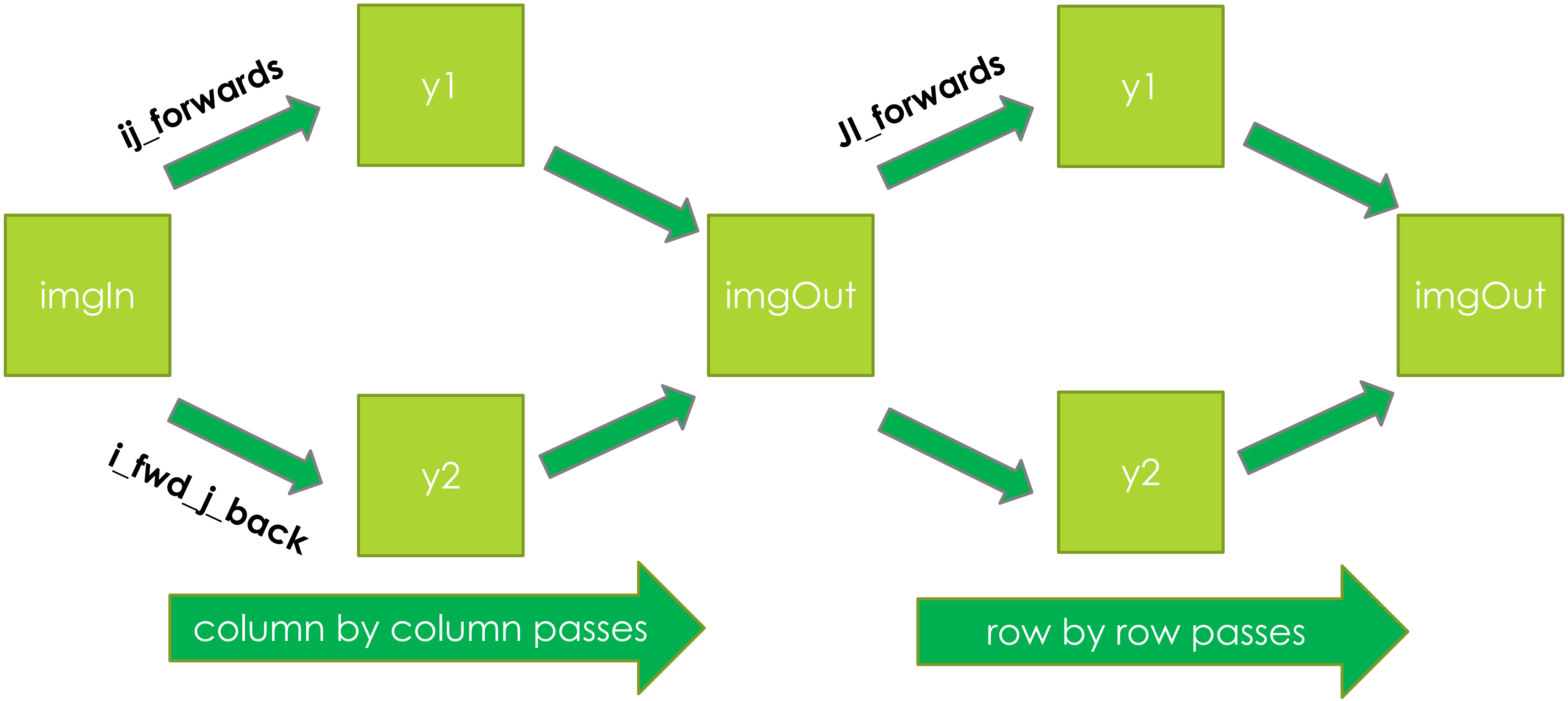
- **Loop body** expresses core computation
- **Iterator** expresses the loop transformation
- **Record** (or class) expresses the storage transformation
- This lets the **programmer** explore possible optimizations
- It needs limited help from the **compiler**
  - Enables performance (good basic optimizations, emphasis on a few specifics)
  - Could confirm legality of transformations in most cases (easier than optimizing)

Deriche Image Processing Algorithm

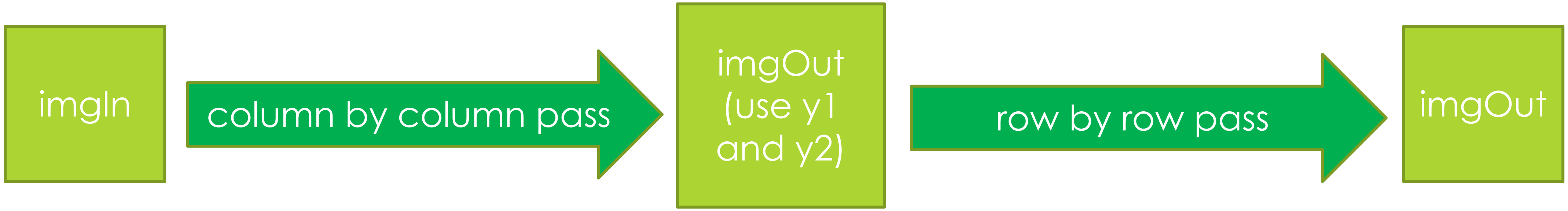
- PolyBench suite: challenge problems for Polyhedral compilers
- Edge detection and smoothing to 2D images
- Computational core: six doubly-nested loops
- Challenges:
  - (No problem with non-affine subscripts.)
  - May be hard to find best iteration order?
  - Best iteration order requires data transform

```
// Deriche.c [YP15], Chapel-ized [Glaser '18]

for i in 0..w-1 {
  ym1 = 0.0;  ym2 = 0.0; xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] = a1*imgIn[i,j]+a2*xm1+b1*ym1+b2*ym2;
    xm1=imgIn[i,j]; ym2=ym1; ym1=y1[i,j]; }}

// for i in 0..w-1
//   for j in 0..h-1 by -1
//      build y2, similarly to above

for i in 0..w-1
  for j in 0..h-1
    imgOut[i,j] = c1 * (y1[i,j] + y2[i,j]);

// three j/i loop nests for horizontal sweep
```

Deriche Data Flow

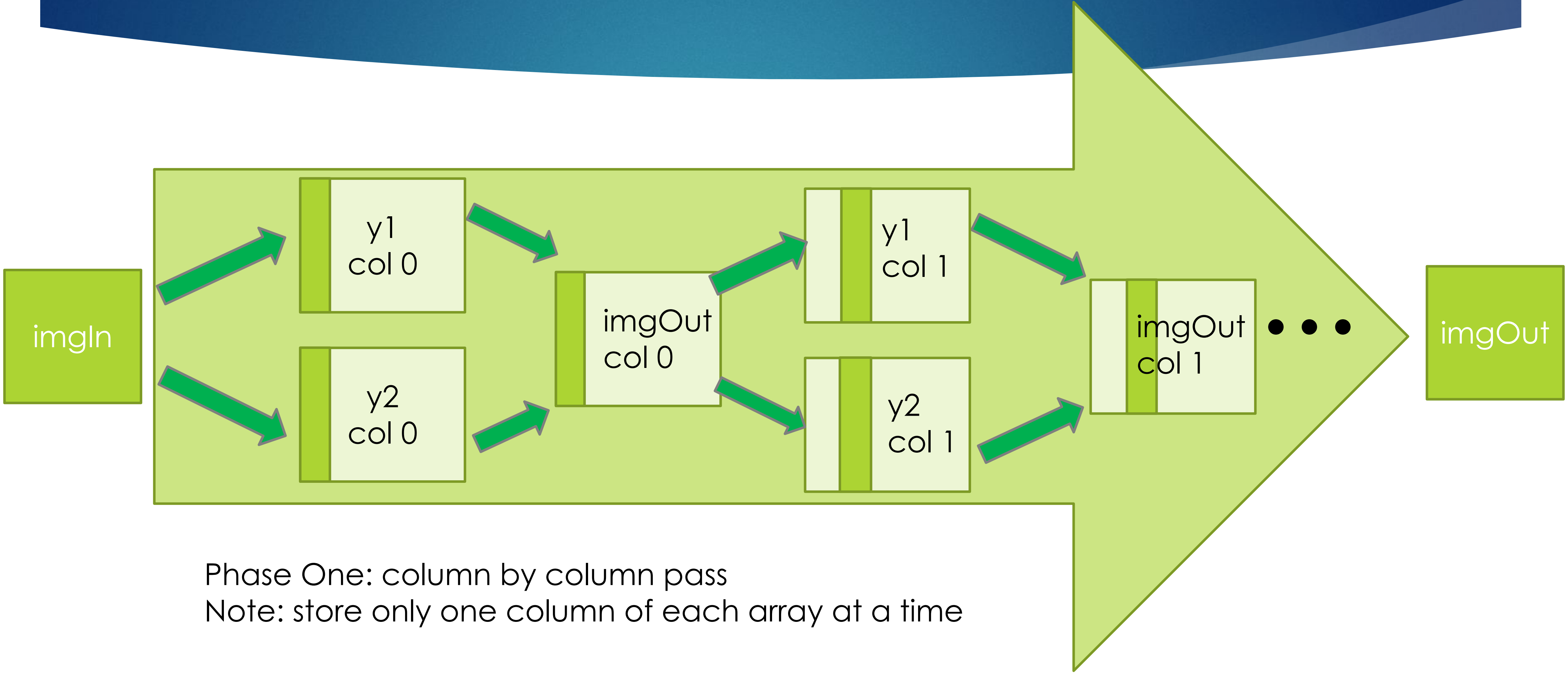


Two Phases




```
iter deriche_iterations(w: int, h: int): (int, int, int) {
 for i in 0..w-1 {
   for j in 0..h-1
     yield (0, i, j);
   for j in 0..h-1 by -1
     yield (1, i, j);
   for j in 0..h-1
     yield (2, i, j);
 }
 for j in 0..h-1 {
   for i in 0..w-1
     yield (3, i, j);
   for i in 0..w-1 by -1
     yield (4, i, j);
   for i in 0..w-1
     yield (5, i, j);
 }
}
```

Two Phases (Detail)



Phase One: column by column pass
Note: store only one column of each array at a time

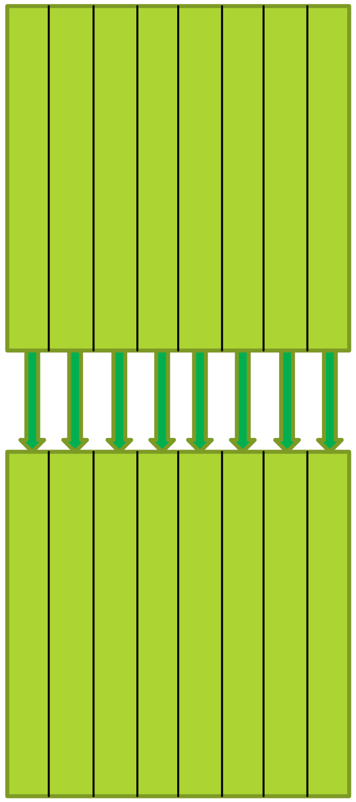
Storage Transformations: Chapel Records

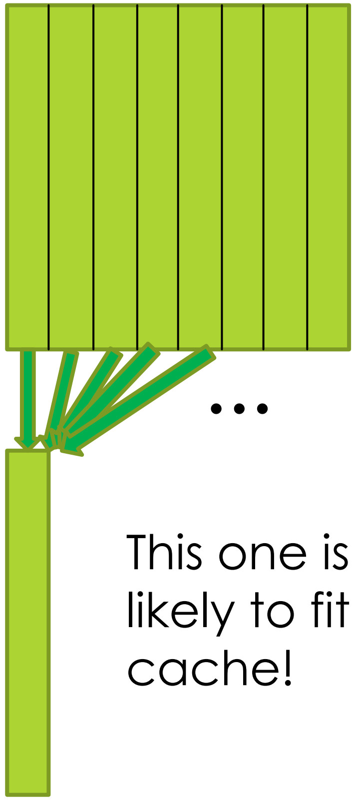
y1 **abstraction**:
A 2-dimensional image

(y2, x1, x2 are similar)

y1 **representation**:
Could be 2-d array
Could be 1-d vector
Could be *per-core* vector

(y2, x1, x2 can be similar)





This one is more likely to fit in cache!

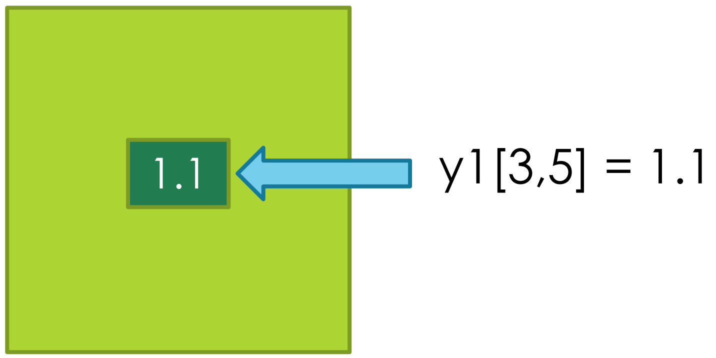
Main Code for Optimizable Deriche

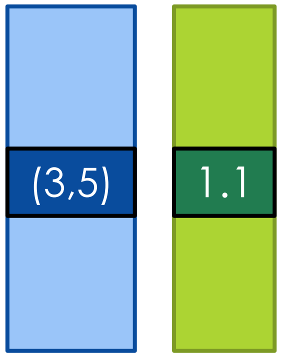
```
// Deriche.c [YP15], Chapel-ized [Glaser '18]


for i in 0..w-1 {
  ym1 = 0.0;  ym2 = 0.0;  xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] = a1*imgIn[i,j]+a2*xm1+b1*ym1+b2*ym2;
    xm1=imgIn[i,j]; ym2=ym1; ym1=y1[i,j]; }}

// for i in 0..w-1
//   for j in 0..h-1 by -1
//      build y2, similarly to above

for i in 0..w-1
  for j in 0..h-1
    imgOut[i,j] = c1 * (y1[i,j] + y2[i,j]);


// three j/i loop nests for horizontal sweep
```

```
// Optimizable Deriche [Glaser '18]

for (statement,i,j) in deriche_iterations(w,h) {

  if (statement == 0)
    y1.set(i,j, a1*imgIn.get(i,j) +
                a2*imgIn.jlower(i,j) +   // i, j-1
                b1*y1.jlower(i,j) +
                b2*y1.jlowerlower(i,j));


  else if (statement == 1)
    y2.set(i,j, a3*imgIn.jhigher(i,j) + ...


  else if (statement == 2)
    imgMid.set(i,j, c1*(y1.get(i,j)+y2.get(i,j)));

// stmts 3,4 build intermediate arrays from imgMid
// stmt    5 then builds final imgOut from those
}
```

Checking for Correctness

- Static Check *possible* in Compiler
  - Apply Polyhedral Model's tests
  - Assume simplest iterator/storage is correct
- Dynamic check via "careful arrays"
  - Check correctness at runtime
- See paper for details





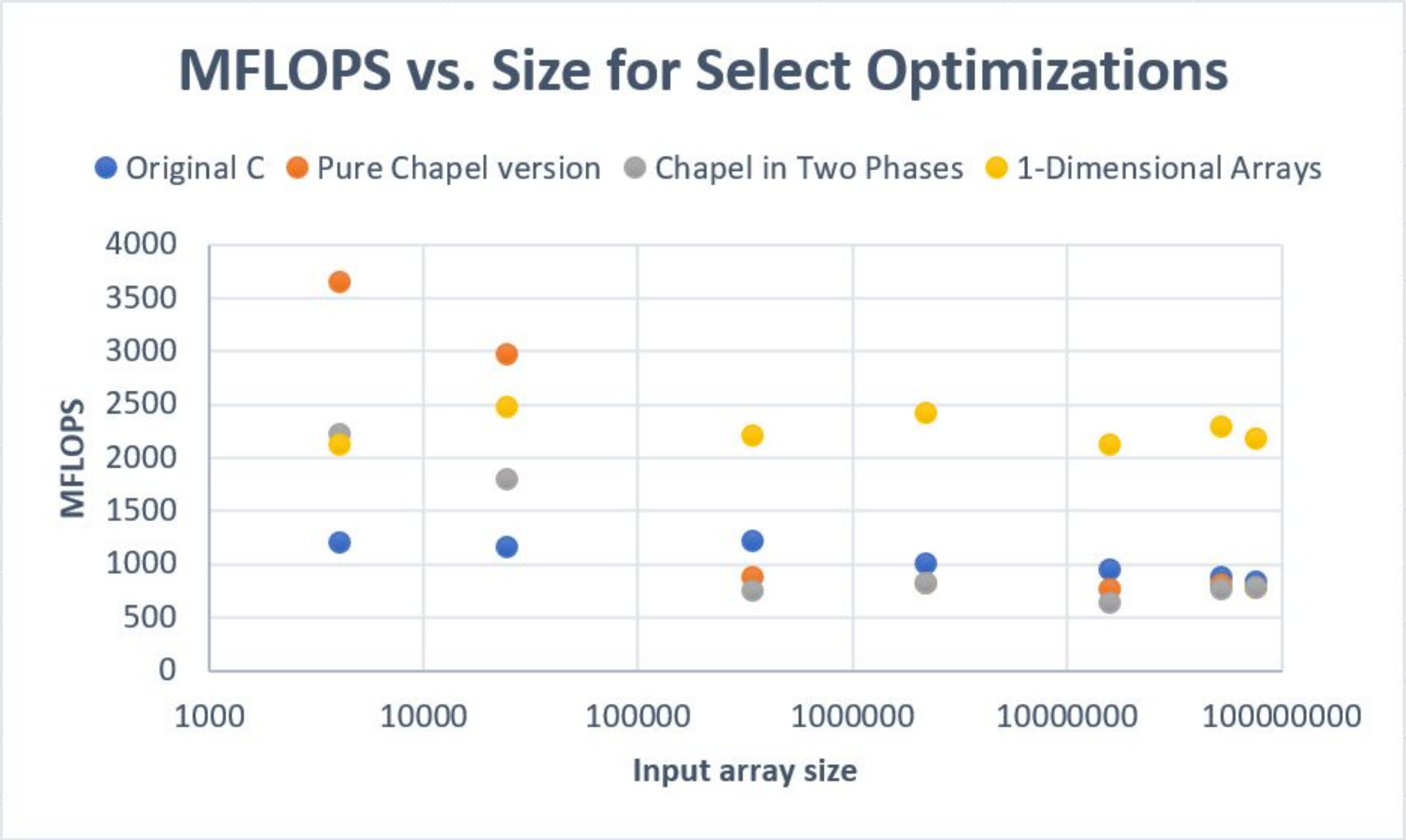
"**Careful Array**" [Feshbach]
stores value at each row, e.g. 1.1
stores index of last write, e.g. (3, 5)

read y1[3,5]: good
read y1[2,5]: error
read y1[4,5]: error

Table of Results (See Mary Glaser's thesis)

Array sizes	Original C		Original Chapel		Two phases, scalars		Two phases, vectors	
wxh	**Seconds**	**MFLOPS**	**Seconds**	**MFLOPS**	**Seconds**	**MFLOPS**	**Seconds**	**MFLOPS**
64x64	0.000110	1192.53	0.000036	3641.71	5.90e-5	2221.56	6.20e-5	2114.06
192x128	0.000682	1152.93	0.000265	2966.91	4.40e-4	1787.35	3.19e-4	2465.30
720x480	0.009089	1216.77	0.012636	875.23	1.49e-2	741.03	5.01e-3	2209.19
1680x1320	0.071262	995.81	0.087314	812.74	8.74e-2	812.32	2.94e-2	2415.85
4000x4000	0.544893	939.63	0.670132	764.03	8.10e-1	631.93	2.41e-1	2122.64
8000x6600	1.922052	879.06	2.124119	795.44	2.22	762.17	7.37e-1	2293.51
7700x9900	2.922805	834.60	3.177009	767.82	2.59	778.59	9.30e-1	2167.27

MFLOPS Graph (see Mary Glaser's Thesis)



Conclusions/Take-Away Messages

- Iterator-based Performance Tuning of Dense Array Codes:
  - When polyhedral approach works, **Chapel matches C** using best known tiling
    - [Bertolacci et. al, ICS '15]
  - Works fine for **imperfectly nested loops**
  - Allows manual search for **good (best?) iteration order** in Deriche
  - Associated record definition can express **data transformation**
  - (In principle, should work for non-affine subscripts … work in progress)
  - Iterator/record abstractions allow static or run-time **correctness checking**
  - Our Iterator/record combination runs **faster than automatically-optimized C**

Related and Future Work

- Related Work
  - Improving polyhedral compilers
    - good, but not done yet, at least three distinct major challenges
  - Programmer-directed tools (AlphaZ, CHILL, etc.)
    - good, but requires tool-specific learning, additional software to update
- Future Work
  - More benchmarks, including FFT
  - Implement static correctness check
  - More optimizations: multi-core, distributed/cluster computing, vectorizing
  - Sparse computations
- Questions?