



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)*tau,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+1))
                    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)) {
```

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 ceildiv(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+1] + A[read,x-1,y+1] +
               A[read,x-1,y-1] + A[read,x+1,y-1] +
               A[read,x+1,y+1] + A[read,x-1,y+1] +
               A[read,x+1,y-1] + A[read,x,y-1] +
               A[read,x,y+1] + A[read,x-1,y] +
               A[read,x+1,y] + A[read,x,y] * 8) * 0.5
          }
        }
      }
    }
  }
}
```

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;
            // note: t%2 stores two time steps
```

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/compiler 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;
```


1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 26

- ▶ 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 $T-1 \geq 1$ and $N-2 \geq 5$ and $N-2 \geq 10$

```
// 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
```

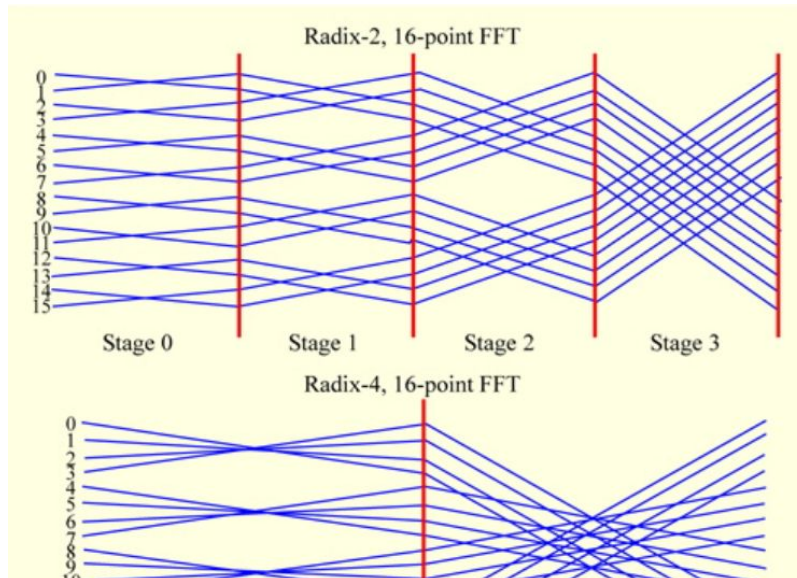
```
for t in 0..T-1 do  
  for x in 1..N-2 do  
    for y in 1..N-2 do  
      A[(t+1)%2, x, y] =  
        (A[t%2,x-1,y] + A[t%2,x,y-1] +  
         A[t%2,x ,y] + A[t%2,x,y+1] +  
         A[t%2,x+1,y]) / 5;
```

```
// 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
```

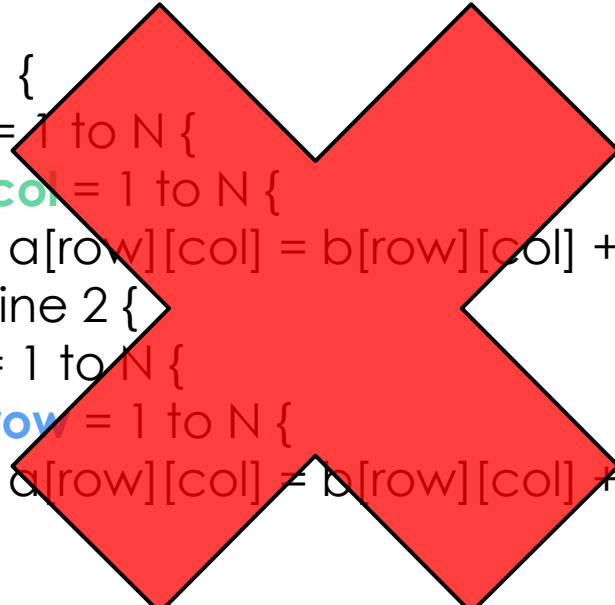

Varying Execution Order without Iterators

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*imgln[i,j] + a2*xm1  
              + b1*ym1 + b2*ym2;  
    xm1 = imgln[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*imgln[i,j] + a2*xm1  
              + b1*ym1 + b2*ym2;  
    xm1 = imgln[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]

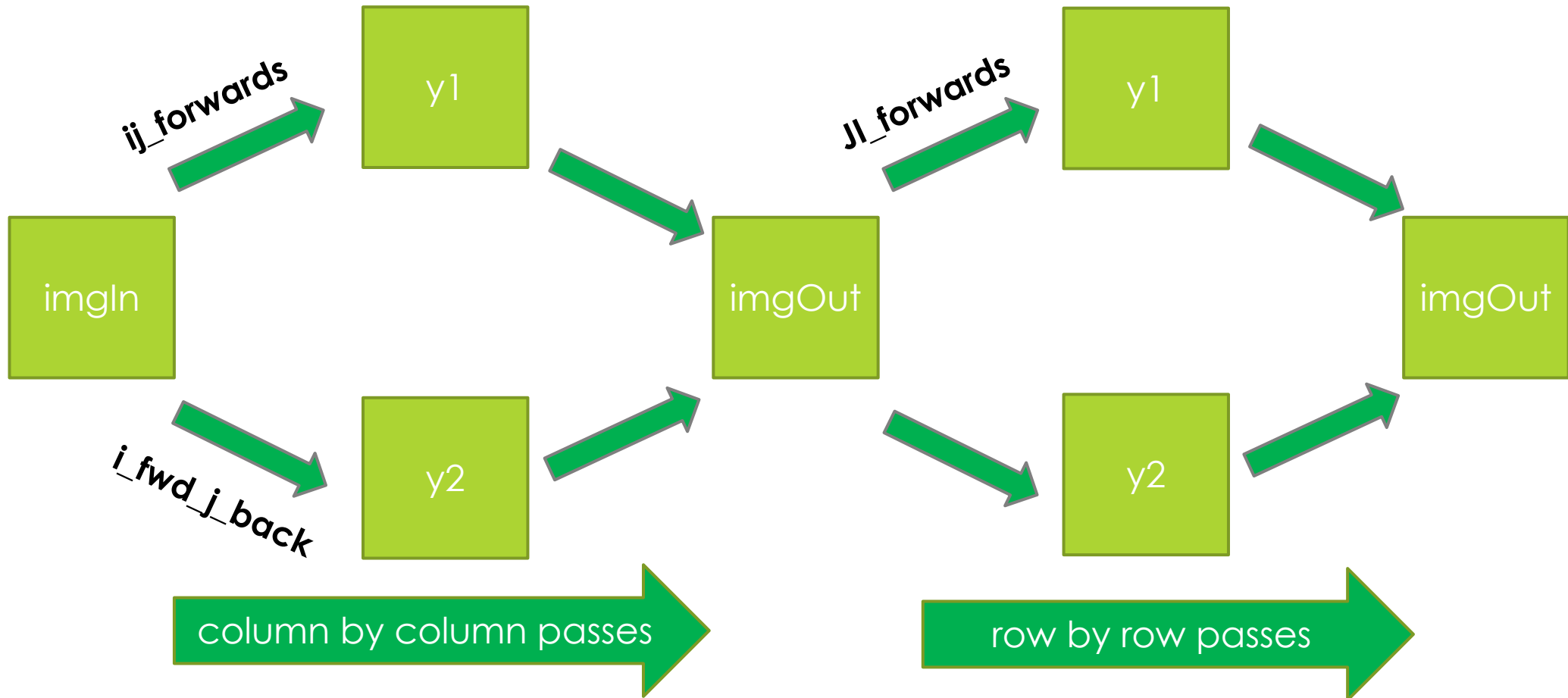
for i in 0..w-1 {
  ym1 = 0.0;  ym2 = 0.0;  xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] = a1*imgIn[i,j]+a2*xm1+b1*ym1+b2*ym2;
    xm1=imgIn[i,j]; ym2=ym1; ym1=y1[i,j]; }}

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

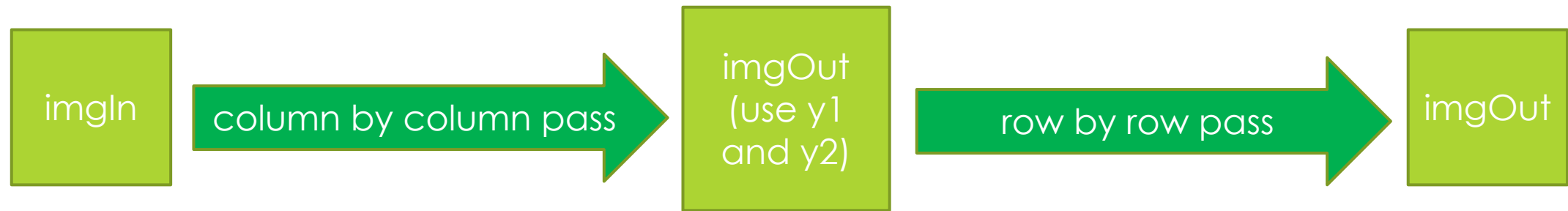
for i in 0..w-1
  for j in 0..h-1
    imgOut[i,j] = c1 * (y1[i,j] + y2[i,j]);

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

Deriche Data Flow



Two Phases

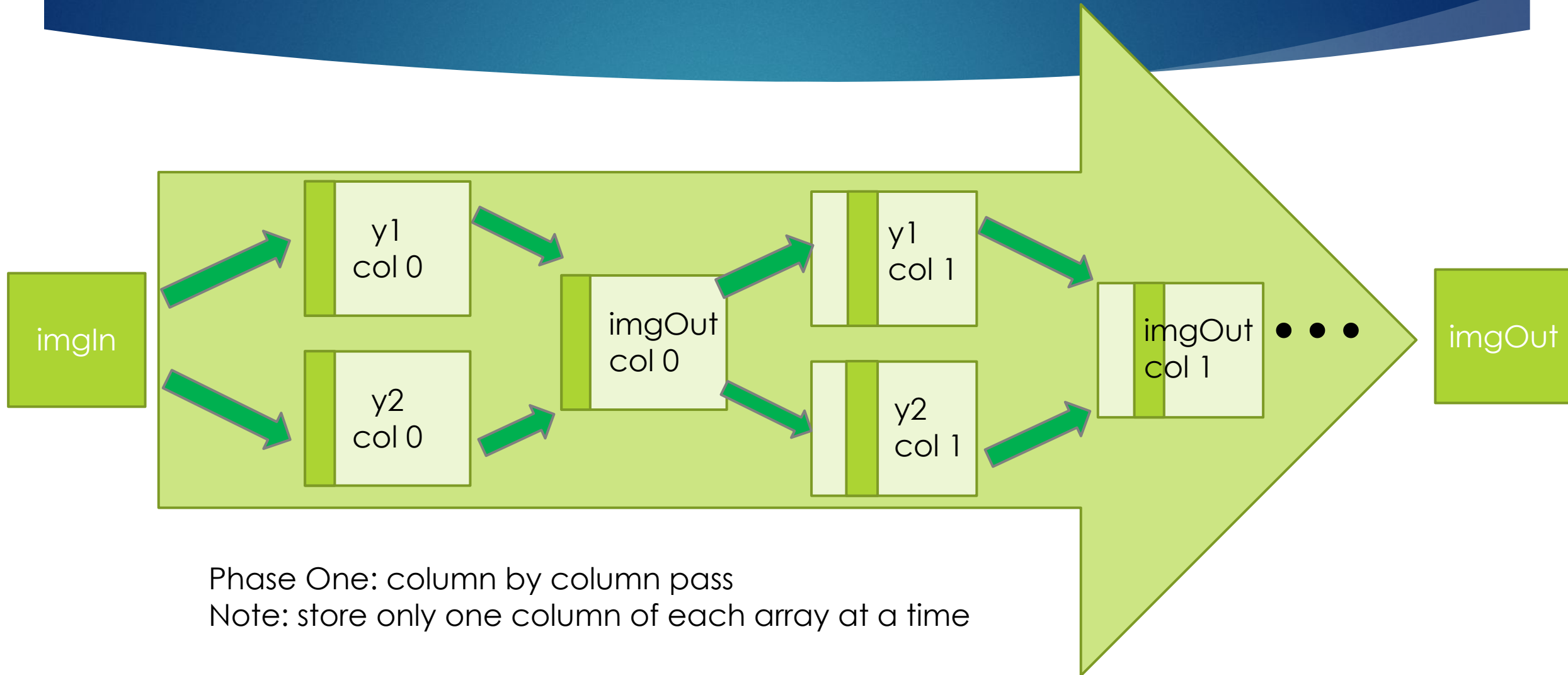


```
iter deriche_iterations(w: int, h: int): (int, int, int) {
```

```
  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)



Storage Transformations: Chapel Records

y1 **abstraction:**

A 2-dimensional image

(y2, x1, x2 are similar)

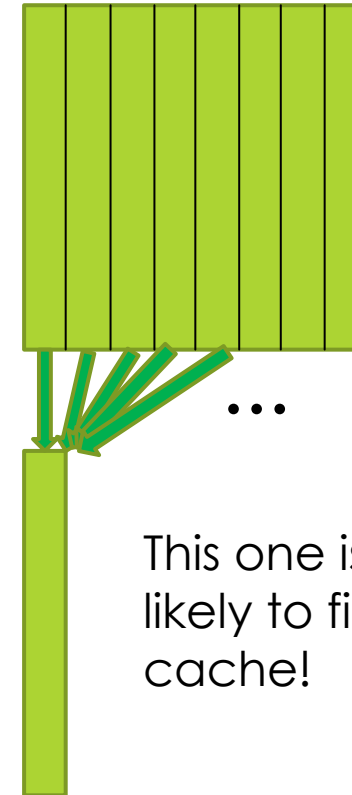
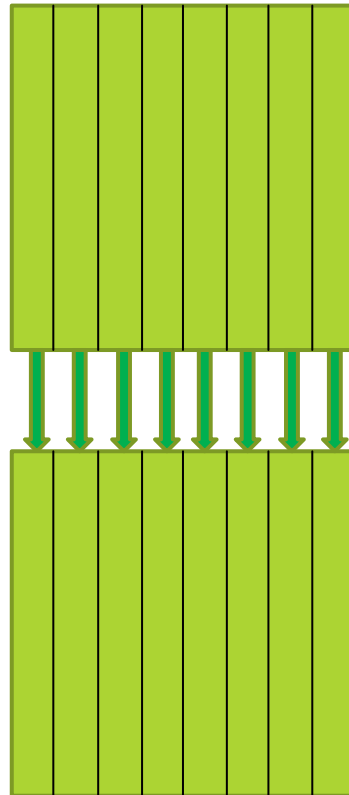
y1 **representation:**

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

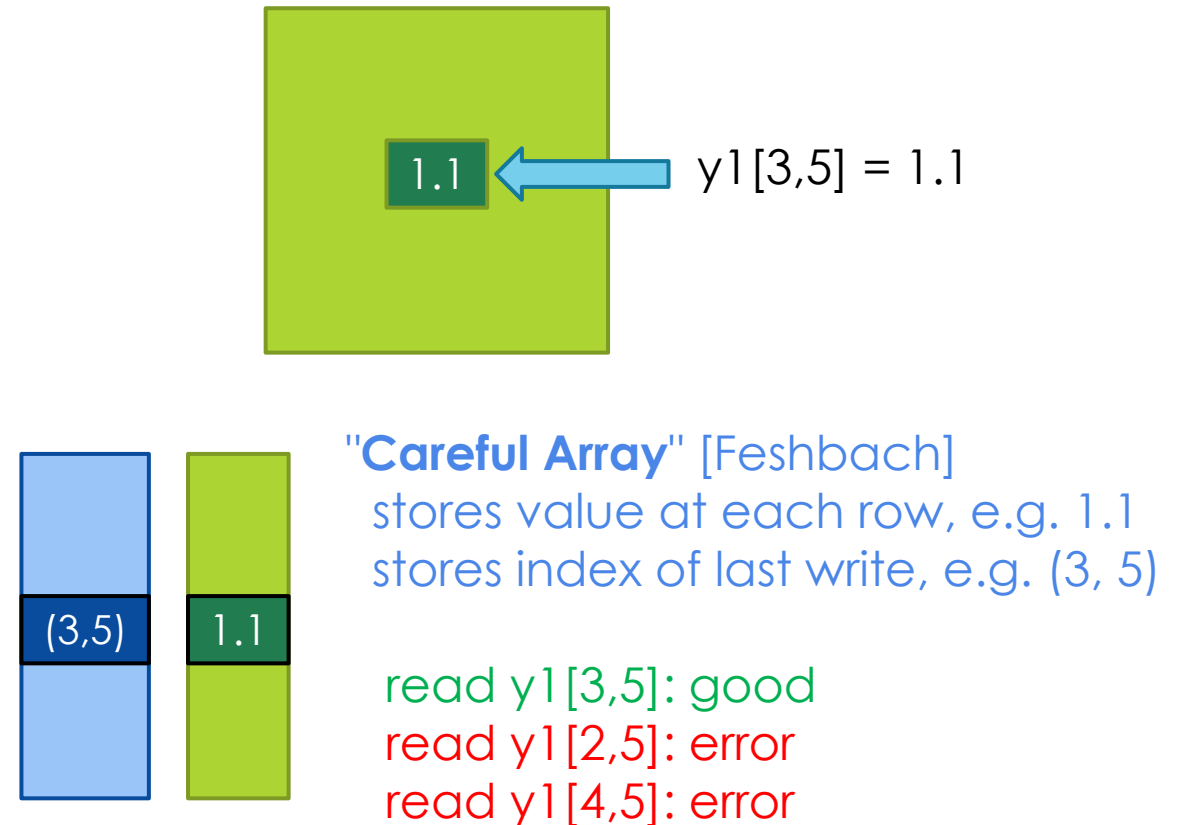
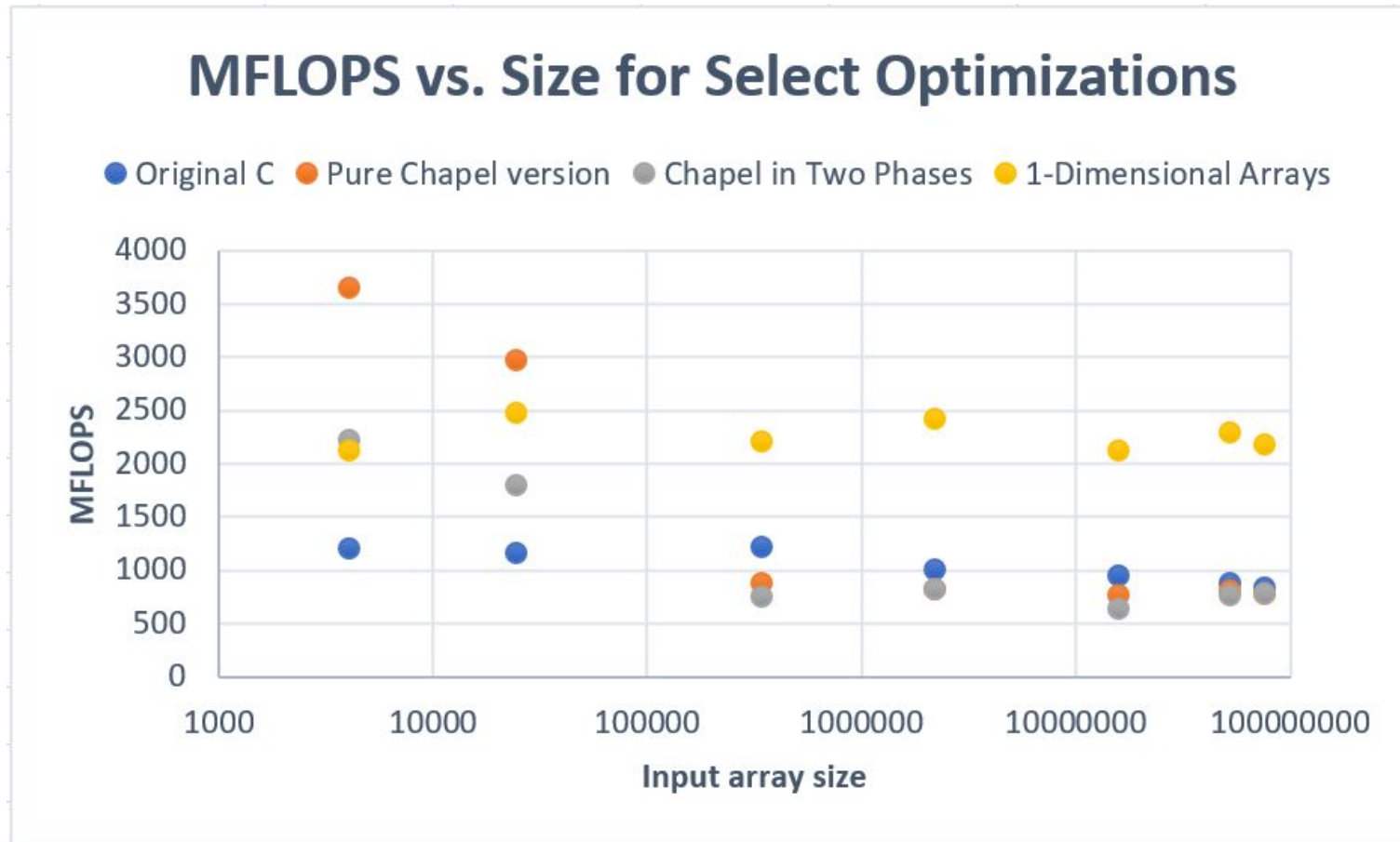


Table of Results (See Mary Glaser's thesis)

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?