Observations from Developing a Streaming Compiler for Polymorphous Computer Architectures

Richard Lethin

(credit for the good ideas here to colleagues at Reservoir and in the Morphware Forum, and with significant gratitude to our funding agencies)

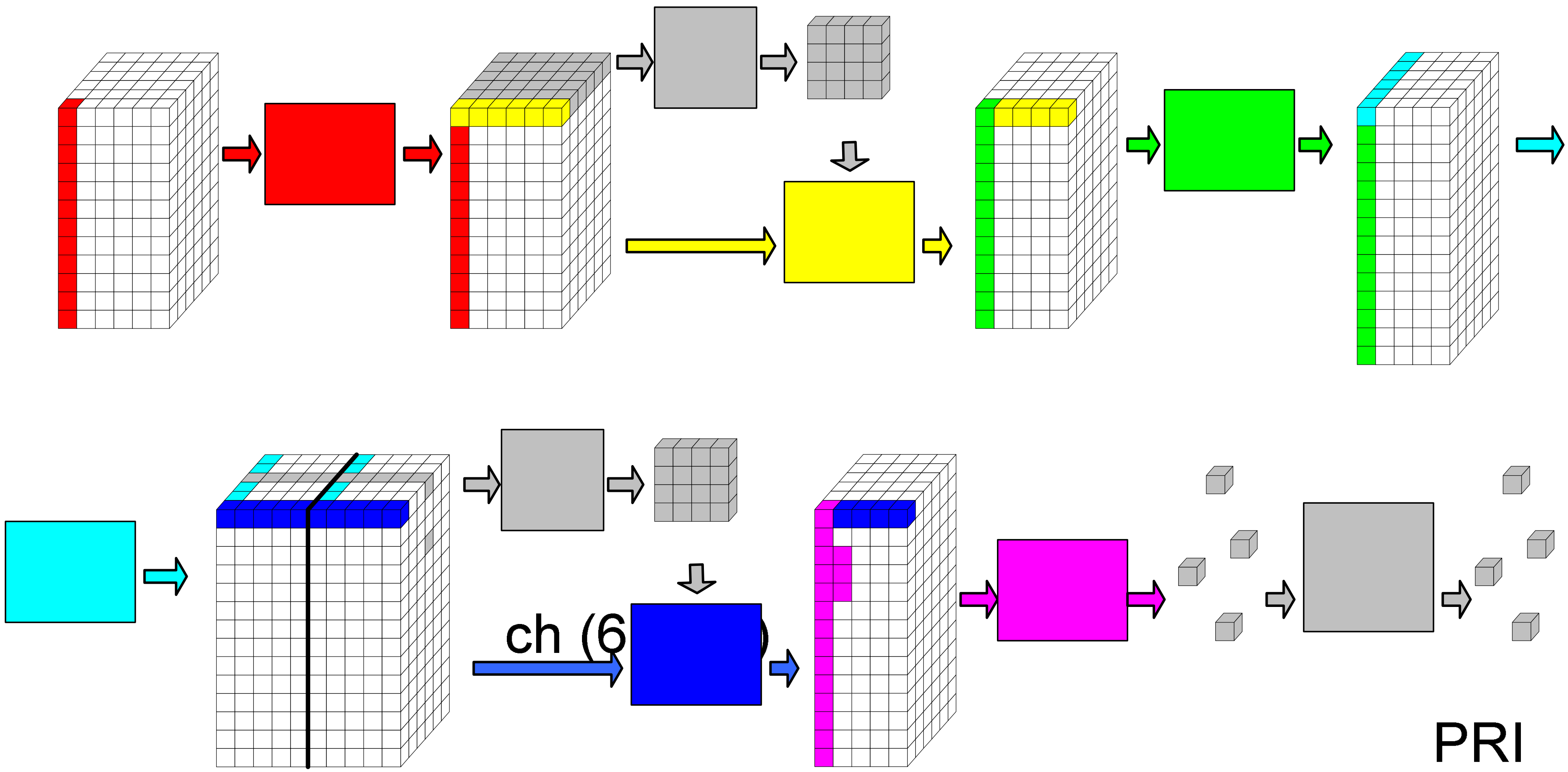
Outline

- Context: HPEC applications, Polymorphous Computer Architectures, Streaming Compilers, Streaming Virtual Machines
- Streaming language, or just do it from C?
- Streaming programming model vs. streaming execution model
- Streaming API for cores, chips chassis, boards
- Start with the compiler technology
- What we've got now
- Some forward research opportunities

Polymorphous Computer Architectures

- DARPA/IPTO 2001-2007 research program
- Architectures and software for *versatile* HPEC
- UT Austin, MIT, Raytheon/USC-ISI, Stanford, Reservoir, GT, …
- Hardware architectures: TRIPS, RAW, Smart Memories, Monarch
- High Level Compiler: R-Stream
- Low Level Compilers: Scale, StreamIT, Monarch
- Morphware: Machine Models, Streaming VM, Threaded VM, UVM, HAL
- Streaming Languages: StreamIT, Brook

HPEC Application Focus, Radar App



Time Delay &
Equalization
FIR

R-Stream Project

HPEC Application in C Language

R-Stream 3.0 Compiler

Raising
High-Level Analysis
Parallelization
Partitioning
Tiling
Locality Enhancement
Placement
Communication Generation
…

Capabilities beyond previous compilers

Input programs: increased scope of optimizations
- Imperfect loop nests.
- Parametric affine static control programs.

Out: increased scope of targets
- Optimizations and transformations for PCA.

Based on mathematics:
- Unique implementation of capabilities that would otherwise just be theory.
- Less “magic”
- First opportunity to use theoretical techniques on C

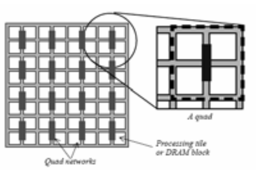
Leveraged Investment

One high-level optimizer leveraged across multiple architectures and implementations.

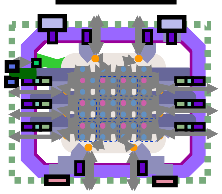
Programming Challenges

Virtual Machine Abstractions

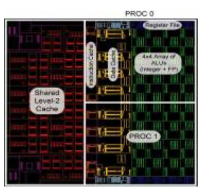
LLC | LLC | LLC | LLC | LLC | LLC | LLC



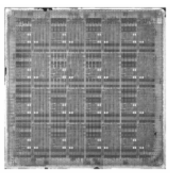
Smart Memories



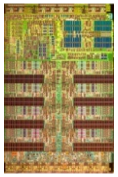
Monarch



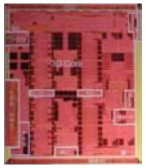
TRIPS



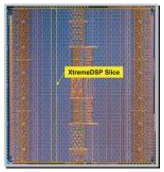
RAW



Cell



GPU



FPGA

Polymorphous Computer Architectures

COTS Streaming Architectures

PCA Hardware:
10-100x FLOPS/Watt advantage
- Parallelism
- SIMD
- Heterogeneous Functional Units
- Coarse grain communication ops
- Streaming Engines
- Tiled Chip Multiprocessing
- Distributed local memories
- Starved for pin bandwidth

You have to take it as given

- (And you should because it's true)… that **in order to get high FLOPS/W from next generation hardware, you need to choreograph a tight execution with:**

- High degrees of concurrency
- Multiple types of concurrency (coarse, ILP, SIMD, …)
- Explicitly controlled communication (DMA, RDMA, message passing)
- Overlapping communications with computations
- Simple pipelines
- Arithmetic intensity – high FLOPS/IO - with very high locality

- The reward is that you might be able to get high percentage of a peak 100 GFLOPS/Watt performance …modulo software

Streaming languages in PCA program

- Brook (circa 2003, now adopted by AMD)
  - Comes from early GPGPU, Imagine project at Stanford
  - Syntactic extensions to C
  - Stream data abstraction, kernel data abstraction
  - “Guide/force programmer” to write in 1-D form
  - “Stream Operators”

- StreamIT
  - Filters, Pipelines, Split/Joins, etc.
  - Elegant language, Java bindings, synchronous dataflow
  - Ask Saman, Rodric (no time here)

Sample Brook Code

Kernels perform computations on streams. This kernel computes pair-wise sum.

```
void kernel streamAdd(stream float s1, stream float s2, out stream float s3) {
   s3 = s1 + s2;
}
```

Represents *s3.push(s1.pop() + s2.pop()).*

```
void kernel weightedSum(stream float image_in[3][3], out stream float image_out) {
  image_out = 1.0 / 9.0 *
     (image_in[0][0] + image_in[0][1] + image_in[0][2] +
      image_in[1][0] + image_in[1][1] + image_in[1][2] +
      image_in[2][0] + image_in[2][1] + image_in[2][2]);
}

int brookMain() {
   float image[100][100][3][3];
   float imageOut[100][100];

   stream float st[3][3];
   stream float stOut;
   stream float stOutDouble;
   …
   streamRead(st, image, 0, 99, 0, 99);
   weightedSum(st, stOut);
   streamAdd(stOut, stOut, stOutDouble);
   streamWrite(imageOut, stOutDouble, 0, 99, 0, 99);
   …
}
```

Stream is 1D, but elements can be arrays. This is a stream of 3x3 arrays.

Use of stream operator to read from array into stream.

Use of streamAdd kernel to double stream.

Observations on (our) Brook

- Fitting streaming into C execution model
  - Expression had to be strip mined
- Could revert back to C, but this is a double edged sword
  - “Stuff that couldn’t be streamed” expressed in C
  - Transpose strictly in Brook – Puzzler, then 2 pages of code!
    - (Same in StreamIt)
    - One character – ‘ – in MATLAB, by the way
    - This is a corner turn!
- To get results on GPU for “hard stuff” a suite of “stream operators” were defined.
  - Language spec became increasingly baroque, situation-specific

Puzzled by how to compile Brook

- Objective is to get parallelism, locality, to distributed memory architectures, etc.
- We needed to undo the bindings in the input program
- We needed to compile C anyway
- We needed a way to express the semantics of stream operators
- …

- Stepping back:
- The claim was that streams abstraction helped avoid C language issues like aliasing, etc.
- But even the one dimensional abstraction was limiting
  - Modern radar algorithms want N-Dimensional constructs
  - STAP, STRAAP, MIMO, …

Our next step: Abstract Array C

- Let's just make it easier to express abstract arrays in C
  - Help avoid the need for alias analysis heroics
  - Get the N-dimensional abstractions we need
  - Light touch on the language
- Solution was a syntactic indication that an array is abstractible
  - A[[i]][[j]]   (an easy modification in the EDG front end)
  - Tells the compiler that the array's layout is undetermined
    - In contrast to regular C
- Reality intrudes
  - How do you pass abstract arrays in functions?
  - A few more "little" language features sneak in (doall, etc.)

Abstract Array C

- Who's going to write in that new syntax anyway?
- Meanwhile, we started to better understand next-generation mapping technologies (polyhedral "stuff") – that was a problem we could get our arms around.
- We're trying to raise the level of abstraction, divorce from physical considerations…
- Finally, we reach the conclusion that defining new language features is energy that we could just put into implementing the analyses that would allow us to take a subset of C programs and "abstract" them.
  - We punt abstract array C, just use C, write the analyses

At the output side, the Streaming Virtual Machine

- LOTS of effort goes into defining an abstraction layer that can encompass TRIPS, RAW, Monarch, Smart Memories
- Streams, Kernels, binding in C, accepted by “low level compiler”
- Push, pop, EOS tokens, stream contexts
- An accompanying Morphware Machine Model
  - Describing capacities, operations, throughputs, topology
  - Lots of stuff…
- Resulting specification available, www.morphware.org

Learning from trying …

- HLC only is able to target a few of the primitives in SVM
- R-Stream 2.0 gets locked into a specific “big VLIW” execution model
  - Shaped by Imagine
  - But what about RAW, Monarch, TRIPS?
  - Is it the compiler limitation or an API limitation?
- …
- Very difficult, and hard to give answers without a mapper
- It’s not so hard.   A TI320cXX… has a clear definition
    - DMA, SIMD, tasks, parallelism, …
    - Maybe it’s not sufficient for PCAs, but it’s not bad…

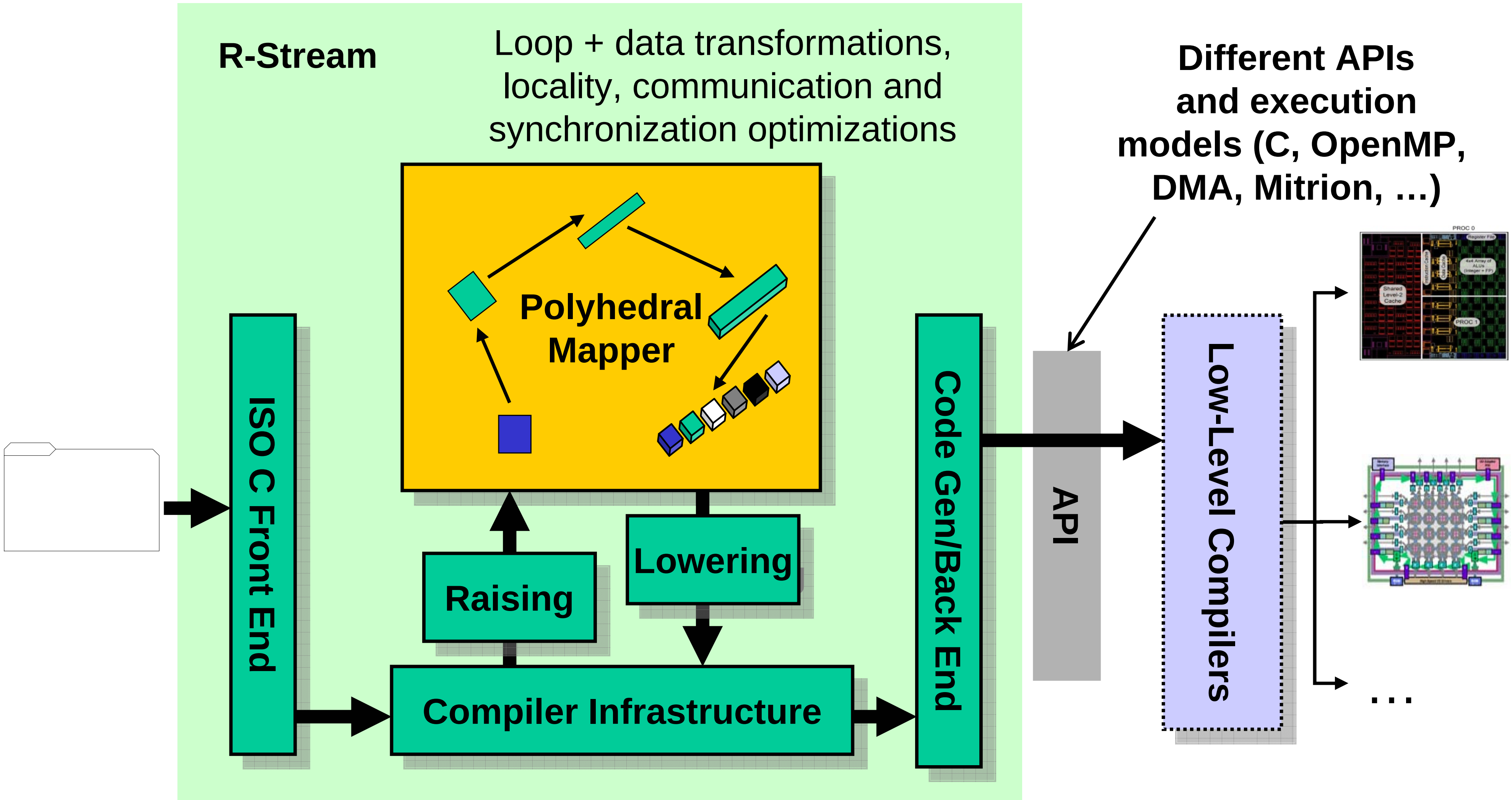
- Oh, and the machine model: the compiler only uses a subset of the primitives in the Machine Model spec.

The conclusion is that the mapper is central

- How do you know what language features help or don't help?
- How do you know what execution models are feasible?
- You only need machine model features that affect the mapper

- So we come up with this…

R-Stream 3.0 Compiler Flow

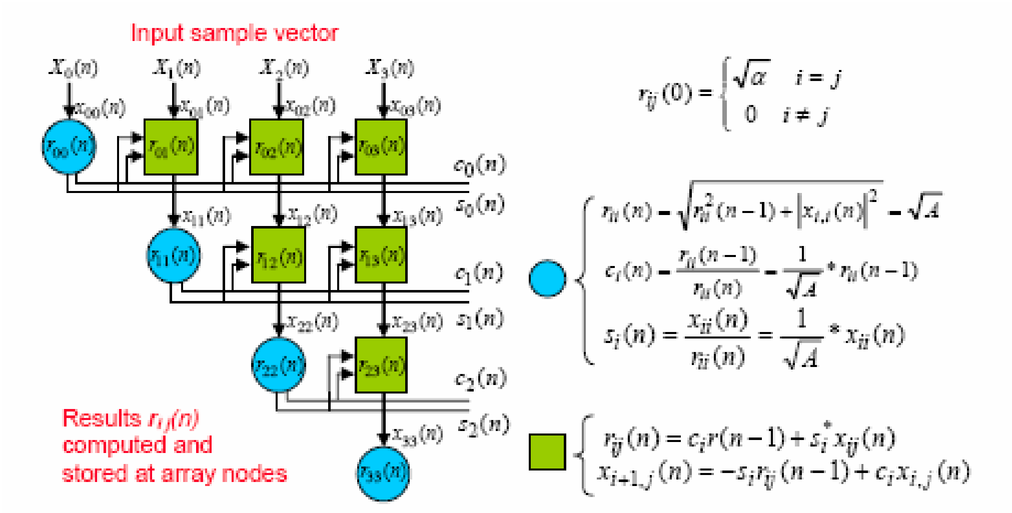


Oh, and then QR decomposition

- The first algorithms that the HPEC users try is QR decomposition…
- R-Stream 2.0 blows up.   Most people don't write QR in a streaming style.
- And there are a few different algorithms for computing QR
  - (Gram-Schmidt, Householder, Givens, …)
  - And the guys who build real HPEC systems know that some are better than others depending on the target
    - Shared memory: Householder or GS
    - Systolic: Givens
    - Decision shaped by shape

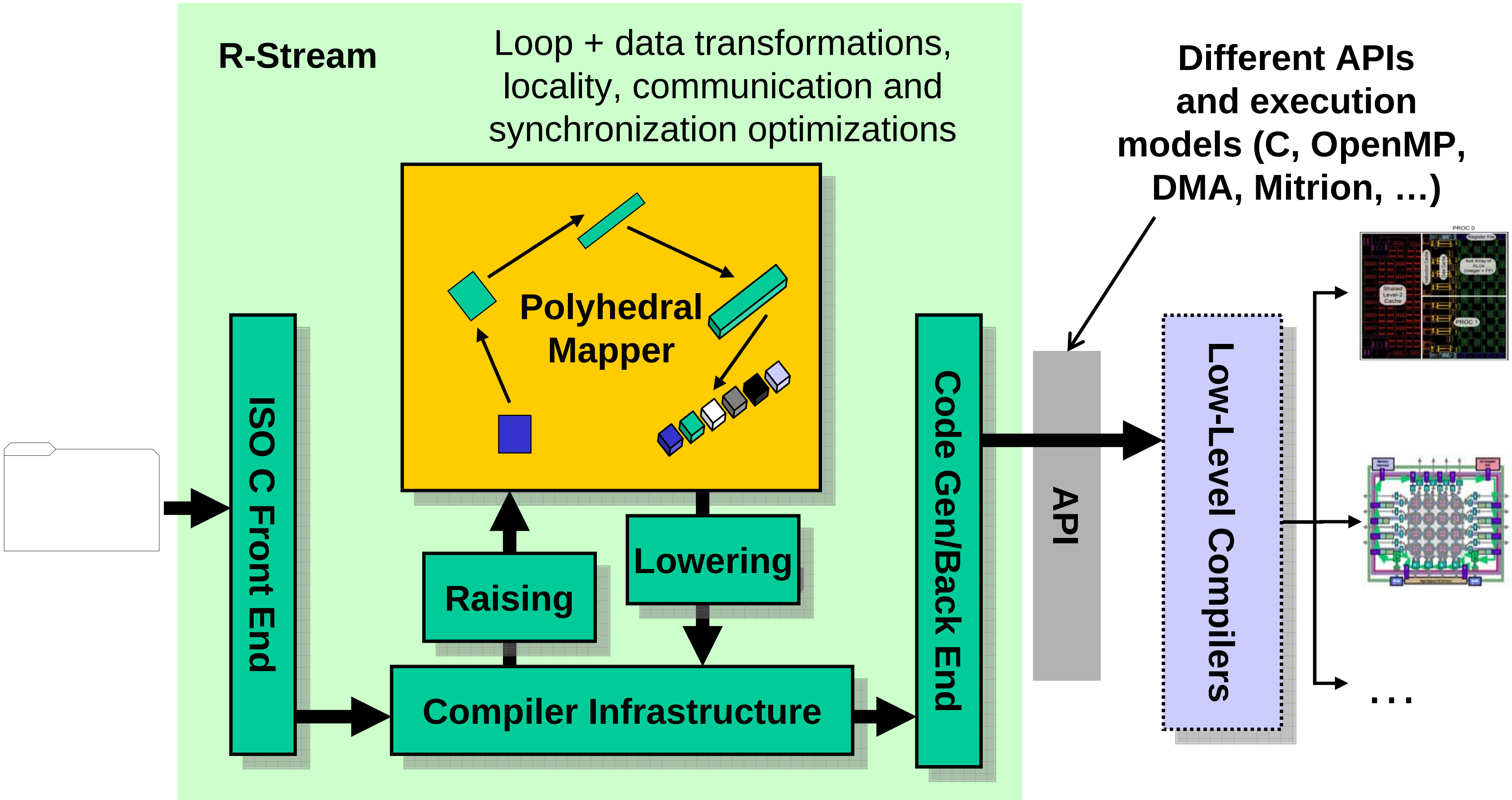
QR Decompositions

- Decompose $\mathbf{X} = \mathbf{QR}$, where $\mathbf{Q}$ is orthonormal ($\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$) and $\mathbf{R}$ is upper triangular
- High performance of QR decomposition is crucial to many HPEC applications, e.g., QR Recursive Least Squares (QR-RLS) in a Space Time Adaptive Processing (STAP) radar
- Very efficient "hand crafted" systolic implementations exist, e.g., Nguyen et. al., HPEC 2005:



Efficiencies of the systolic form come from multidimensional, wavefront parallelism and high degrees of locality

R-Stream Compiler Flow



The Polyhedral Model

- Linear algebraic model for representing loops
- Iteration spaces as polyhedra. Dependencies as polyhedral relations
- Statement-wise schedules: when + where a statement is executed
- Advantages:
  - Greater scope of programs optimized
  - Parametric programs optimized
  - Common representation for all mapping steps
  - Optimizations framed as (relatively) efficient problems for common mathematical solvers
- This allows compiler to optimize QR algorithms
  - in a way that is not possible with “classic” optimizers.
- Not specific to QR (i.e., not a “fastest QR in the West” library)
  - Allows high-level optimization of QR jointly with other kernels

Polyhedral Representation in a Nutshell

```
for (i=2; i<=M; i++) {
   for (j=0; j<=N; j+=2)
      A[i,N-j] = C[i-2,4*i+j/2];
   for (j=i; j<=N; j++)
      B[i,N-j] = A[i,j+1];
}
```

Iteration domains as polyhedra

$$\{(i,j) \mid 2 \le i \le M, i \le j \le N\}$$

Affine schedules determine the execution order and place

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

Variables and access functions as polyhedra

B

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

A

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

Dependence relations as polyhedra tie these components together

Affine Scheduling

Affine scheduling : given statements $S_1,\ldots,S_n$ and dependence relations $\mathbf{R}_{ij}$,

Find statement-wise affine schedule $\Theta = (\Theta_{S_1},\ldots,\Theta_{S_n})$

$\Theta_{S_i}(x)$ maps iteration x of statement S_i to its execution time

A schedule is legal **iff** $\Theta_{S_i}(x) \succ \Theta_{S_j}(y), \quad (x,y) \in \mathbf{R}_{ij}$ for all i,j

"after"

"Iteration x of S_i depends on iteration y of S_j"

Affine Scheduling and Space-Time Mappings

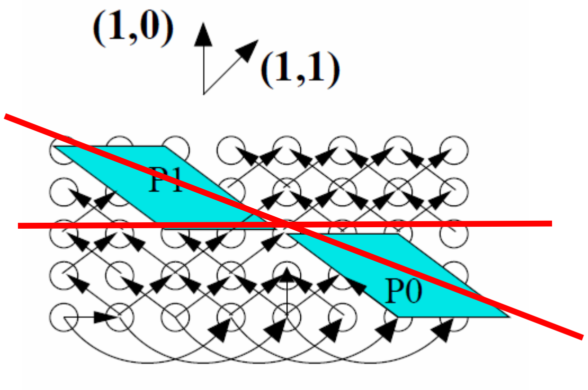
Generalization from schedules to **space - time** mappings:

$$\Theta_{S_i}(x) = \begin{bmatrix} s_1(x) \\ t_2(x) \\ s_3(x) \\ \vdots \\ t_k(x) \end{bmatrix}$$

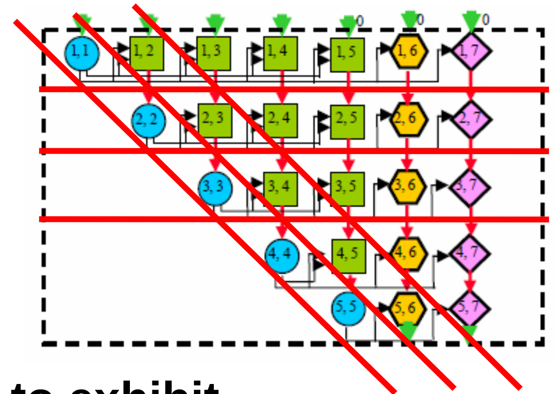
Space dimensions (can be interpreted as processor element coordinates)

Time dimensions determine execution order

Parallelism Types and Loop Transformations



⇔



Parallelism not always <u>that</u> trivial to exhibit

- Automatically exhibits *wavefront hyperplanes* essential for:
  - *Communication-free* parallelism
  - Pipelined parallelism with *near-neighbor communications* thanks to *permutable loops* (i.e. all dependences are forward)
  - Tiling for *data locality* and task aggregation (register reuse)
- Finds hyperplanes automatically for *whole programs*, not just QR
- Enables *hierarchical parallelism* exploitation (FPGA, SMP, MPI …)
- General formulation only available since 2007; R-Stream improves it

Tradeoff between Parallelism and Locality

Maximizing locality

```
for (i=0; i < N; i++) {
  for (j=0; j < N; j++) {
    B[j][i] = A[j][i] + u1[j] * v1[i] +
              u2[j] * v2[i];
    x[i] = x[i] + B[j][i] * y[j] * beta
  }
  x[i] = x[i] + z[i];
  doall (j = 0; j < N; j++)
    w[j] = w[j] + B[j][i] * x[i] * alpha;
}
```

Maximizing coarse-grained parallelism

```
doall (i = 0; i <= N + -1; i++)
   doall (j = 0; j <= N + -1; j++)
      B[i][j] = A[i][j] + u1[i] * v1[j] +
                u2[i] * v2[j];
doall (i = 0; i <= N + -1; i++)
   for (j = 0; j <= N + -1; j++)
      x[i] = x[i] + B[j][i] * y[j] * beta;
doall (i = 0; i <= N + -1; i++)
   x[i] = x[i] + z[i];
doall (i = 0; i <= N + -1; i++)
   for (j = 0; j <= N + -1; j++)
      w[i] = w[i] + B[i][j] * x[j] * alpha;
```

Maximizing a weighted sum of parallelism and locality

```
doall (i = 0; i < N; i++) {
   doall (j = 0; j < N; j++)
      B[j][i] = A[j][i] + u1[j] * v1[i] +
                u2[j] * v2[i];
   reduction_for (j = 0; j < N; j++)
      x[i] = x[i] + B[j][i] * y[j] * beta;
   x[i] = x[i] + z[i];
}
doall (i = 0; i < N; i++)
   reduction_for (j = 0; j <= N + -1; j++)
      w[i] = w[i] + B[i][j] * x[j] * alpha;
```

Optimization can frames the tradeoffs between *parallelism and locality*

Givens QR

- Uses Given's rotations to "locally" zero out elements

$$G(i,j,\theta)=\begin{bmatrix} 1 & \dots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos(\theta) & \cdots & \sin(\theta) & \ddots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin(\theta) & \cdots & \cos(\theta) & \cdots & 0 \\ \vdots & & \vdots & & & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ \\ \end{matrix}$$

$$\qquad\qquad\qquad i \qquad\qquad\qquad j$$

Givens QR in Plain Old Sequential C

```
#define N 1024

for (int k = 0; k < N-1; k++) {
  for (int i = N-2; i >= k; i--) {
    float a = A[i][k];        // S0
    float b = A[i+1][k];      // S1
    float d = sqrt(a*a+b*b);
    float c = a/d;
    float s = -b/d; // S2
    for (j = k; j < N; j++) {
      float t1 = A[i][j]*c + A[i+1][j]*s;
      float t2 = A[i+1][j]*c - A[i][j]*s;
      A[i][j]   = t1;
      A[i+1][j] = t2;  // S3
    }
  }
}
```

Array Expansion

- Creates additional storage to ensure parallelism exploitation
- Removes "*memory-based*" dependences
- Allows exclusive focus on *producer-consumer* relationships
  - Discarding *producer-producer* conflicts

```
#define N 1024

for (int k = 0; k < N-1; k++) {
  for (int i = N-2; i >= k; i--) {
    float a = A[i][k];        // S0
    float b = A[i+1][k];      // S1
    float d = sqrt(a*a+b*b);
    float c = a/d;
    float s = -b/d; // S2
    for (j = k; j < N; j++) {
      float t1 = A[i][j]*c + A[i+1][j]*s;
      float t2 = A[i+1][j]*c - A[i][j]*s;
      A[i][j]   = t1;
      A[i+1][j] = t2;  // S3
    }
  }
}
```

Before

```
for (int i = 0; i <= 1022; i++) {
  for (int j = 0; j <= - i + 1022; j++) {
    S0(a[i][j], A[1023-j][i]);
    S1(b[i][j], A[1022-j][i]);
    S2(a[i][j], b[i][j], c[i][j], s[i][j]);
    for (int k = 0; k <= - i + 1023; k++)
      S3(A[1022-j][i+k], A[1023-j][i+k],
         c[i][j], s[i][j]));
  }
}
```

After (simplified statement notation)

Parallelization Algorithm

```
for (int i = 0; i <= 1022; i++) {
  for (int j = 0; j <= - i + 1022; j++) {
    S0(a[i][j], A[1023-j][i]);
    S1(b[i][j], A[1022-j][i]);
    S2(a[i][j], b[i][j], c[i][j], s[i][j]);
    for (int k = 0; k <= - i + 1023; k++)
      S3(A[1022-j][i+k], A[1023-j][i+k],
         c[i][j], s[i][j]));
  }
}
```

Before

$$\Theta_{S0}(i,j) = [i, i+j]$$
$$\Theta_{S1}(i,j) = [i, i+j]$$
$$\Theta_{S2}(i,j) = [i, i+j]$$
$$\Theta_{S3}(i,j,k) = [i, i+j, k]$$

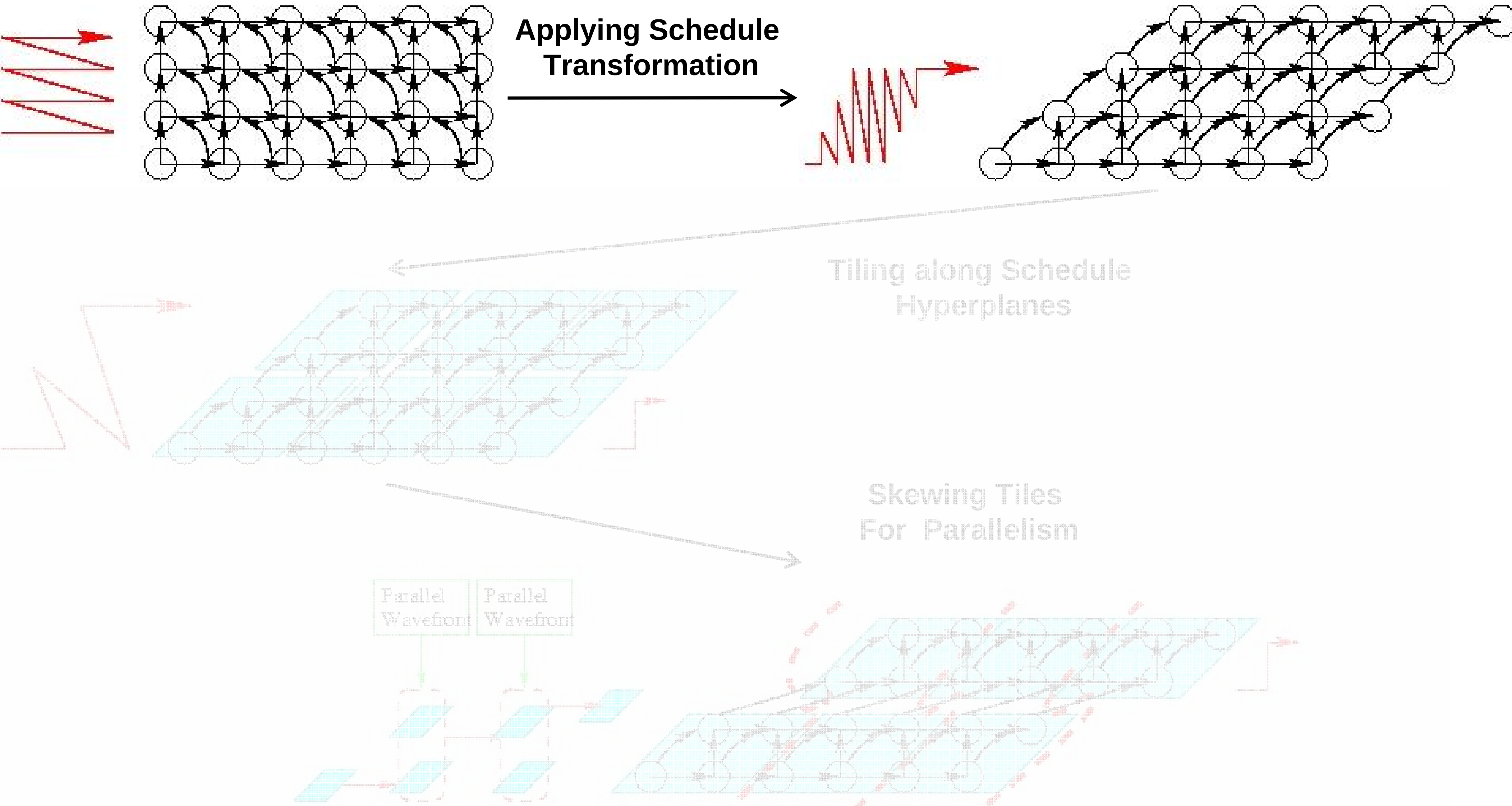
Schedule

```
for (int i = 0; i <= 1022; i++) {     // permutable
  for (int j = i; j <= 1022; j++) {   // permutable
    S0(a[i][-i+j], A[1023+i-j][i]);
    S1(b[i][-i+j], A[1022+i-j][i]);
    S2(a[i][-i+j], b[i][-i+j], c[i][-i+j], s[i][-i+j]);
    doall (int k = 0; k <= - i + 1023; k++)
      S3(A[1022+i-j][i+k],
         A[1023+i-j][i+k],
         c[i][-i+j], s[i][-i+j]);
  }
}
```

Wavefront parallelism and locality found (by virtue of "permutable" attribute), now exploitable in next steps …

After

2-D Analogy (Applying the Parallelization Algorithm)



Tiling

```
for (int i = 0; i <= 1022; i++) {     // permutable
  for (int j = i; j <= 1022; j++) {   // permutable
    S0(a[i][-i+j], A[1023+i-j][i]);
    S1(b[i][-i+j], A[1022+i-j][i]);
    S2(a[i][-i+j], b[i][-i+j], c[i][-i+j], s[i][-i+j]);
    doall (int k =
      S3(A[1022+i-
         A[1023+i-
         c[i][-i+j
  }
}
```

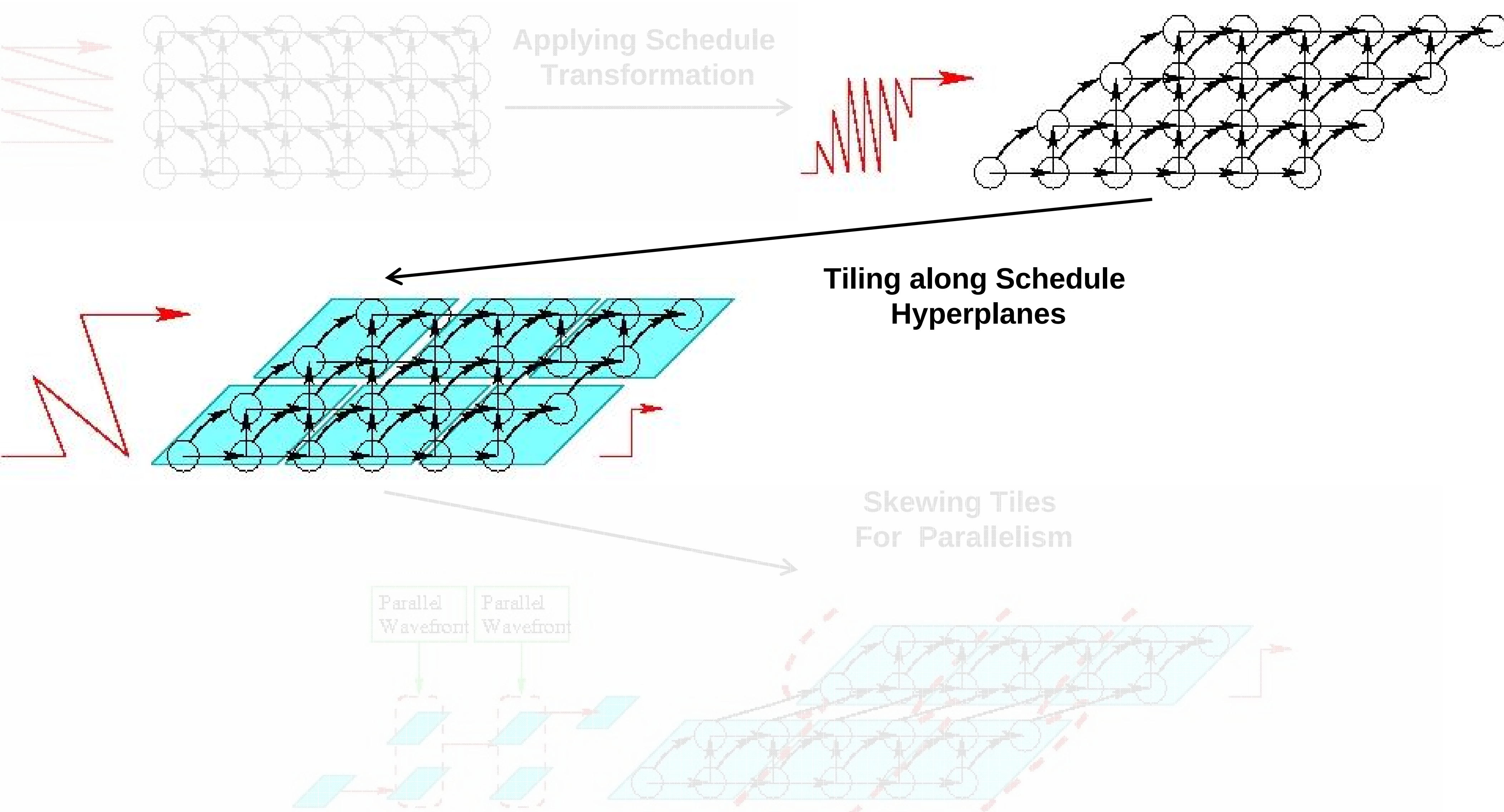
Before

After

```
for (i = 0; i <= 960; i += 64) { // permutable
   lo0 = max(0, i + -15);
   gap1 = - lo0 & 15;
   for (j = lo0 + gap1; j <= 1008; j += 16) { // permutable
     // tiled loops for S0, S1, S2 omitted
     doall(k=0; k <= min(-i+1023, 896); k += 128) {
       for (l=i; l <= min(i+63,1022,j+15,-k+1023); l++) {
         for (m = max(l, j); m <= min(1022, j + 15); m++) {
           doall (n = k; n <= min(k+127, -l+1023); n++) {
              S3(A[1022 + l – m][l + n],
                 A[1023 + l – m][l + n],
                 c[l][-l+m],s[l][-l+m]);
           }
        }
     }
   }
}
```

The locality implicit in the schedule permits a self-contained inner loop tile with a small, constrained memory footprint

2-D Analogy (Tiling)



Skewing the Tile Space (⬄ Pipelined Parallelism)

Before

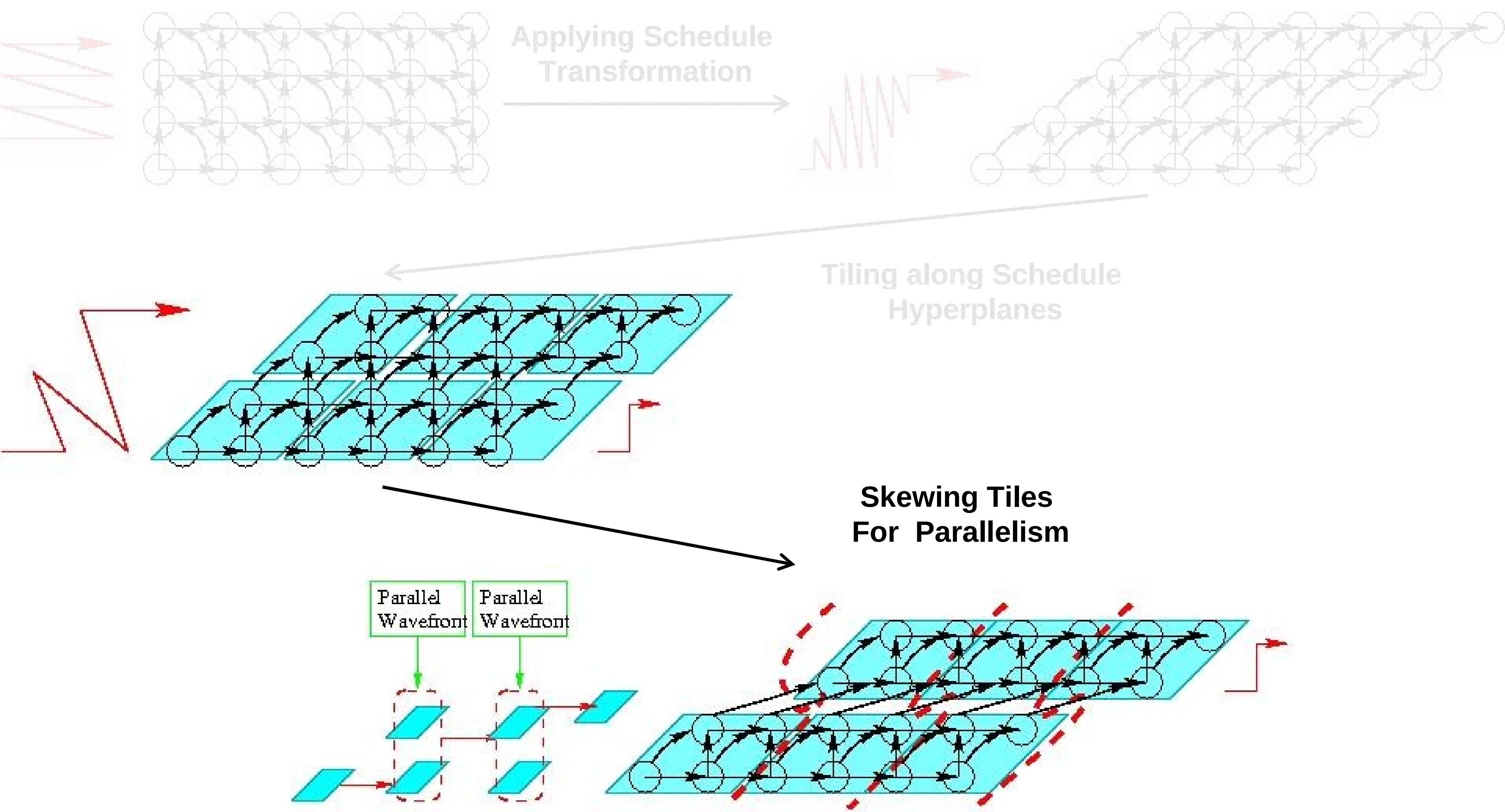
```
for (i = 0; i <= 960; i += 64) { // permutable
   lo0 = max(0, i + -15);
   gap1 = - lo0 & 15;
   for (j = lo0 + gap1; j <= 1008; j += 16) { // permutable
     // tiled loops for S0, S1, S2 omitted
     doall(k=0; k <= min(-i+1023, 896); k += 128) {
       for (l=i;
         for (m
           doall
              S3
           }
         }
       }
     }
   }
}
```

After

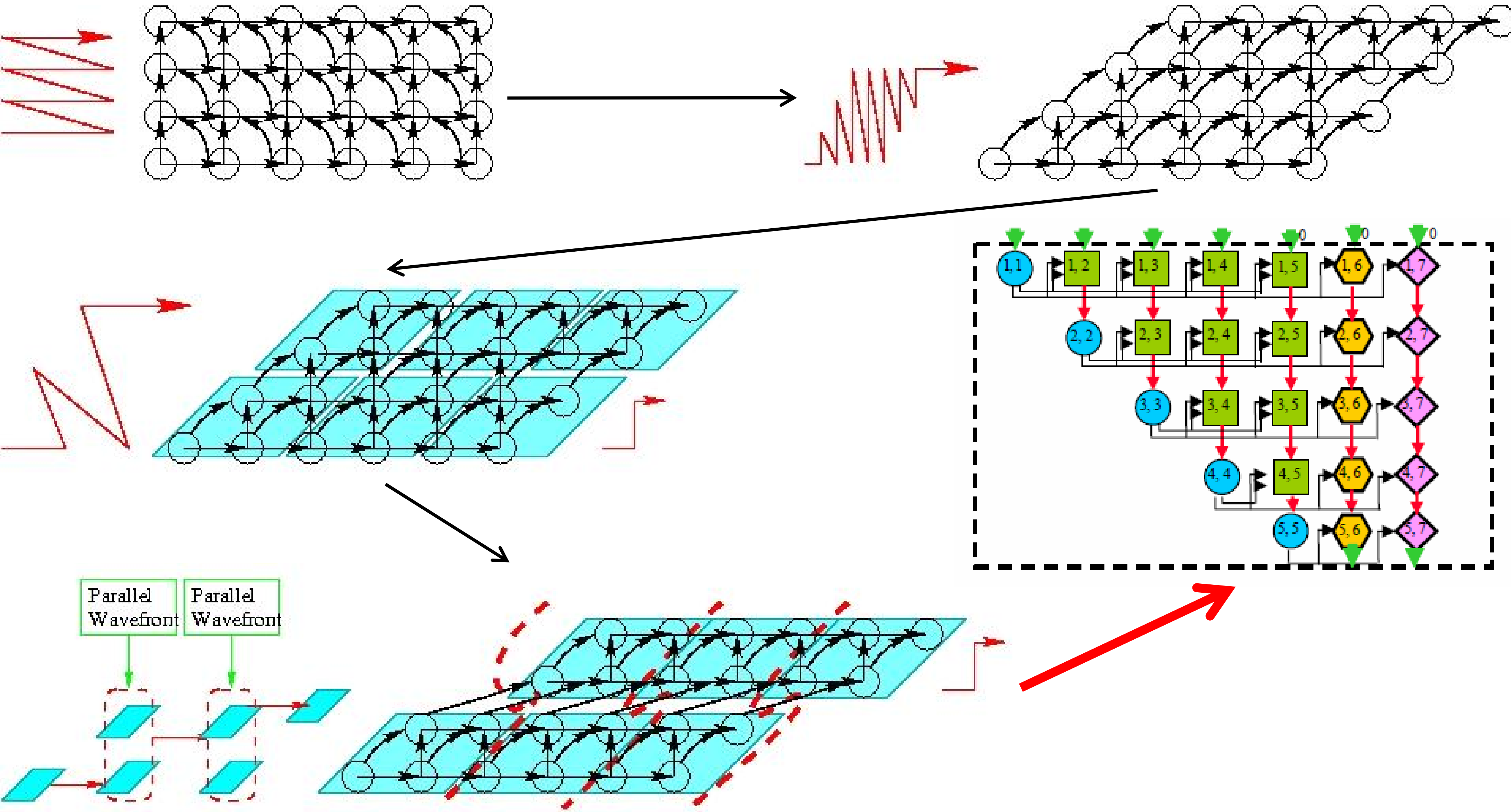
```
for (int i = 0; i <= 78; i++) {
  doall (int j = max(i-15, (4*i+ 4) / 5); j <= min(i, 63); j++) {
    // Tiled loops for S1, S2, S3 omitted
    doall (k = 0; k <= min(7, ( - i + j + 15) / 2); k++) {
      for (l = 64 * i -64 * j;
           l <= min(64*i-64*j+63, 16*j+15, 1022); l++) {
        for (m=max(l, 16*j); m <= min(1022, 16 * j + 15); m++) {
          doall (n = 128 * k; n <= min(128*k+127, -l+1023); n++)
{
              S3(A[1022 + l – m][l + n],
                 A[1023 + l – m][l + n],
                 c[l][-l+m],
                 s[l][-l+m]);
          }
        }
      }
    }
  }
}
```

The wavefront parallelism in the schedule (the permutable loops) is skewed to create pipeline parallelism

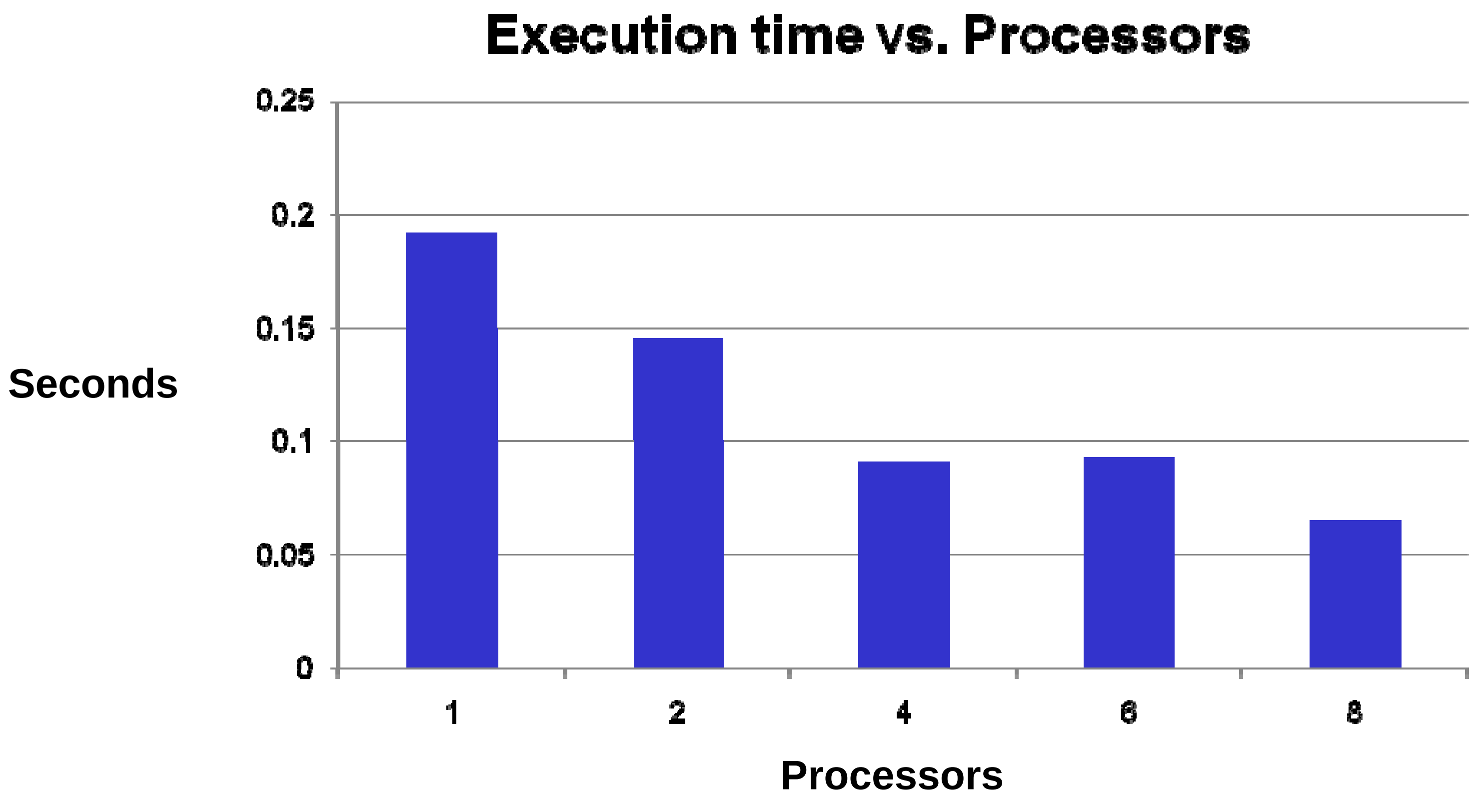
2-D Analogy (Skewing the Tile Space)



2-D Analogy (Summary)



Some Performance Results (Givens QR)



Automatically parallelized
Speedup with increasing # of processors

Xeon 8-core (bi quad core) Dell 2 GHz
512x512 matrix
OpenMP produced at back end
gcc 4.2.3 –O6 –SSE3

1 processor version is without R-Stream

Modified Gram-Schmidt QR

```
for (int k = 0; k < N; k++) {
  float nrm = 0;
  for (int i = 0; i < M; i++)
    nrm += A[i][k] * A[i][k];
  R[k][k] = sqrt(nrm);
  for (int i = 0; i < M; i++)
    Q[i][k] = A[i][k] / R[k][k];
  for (int j = k+1; j < N; j++) {
    R[k][j] = 0;
    for (int i = 0; i < M; i++)
      R[k][j] += Q[i][k] * A[i][j];
    for (int i = 0; i < M; i++)
      A[i][j] -= Q[i][k] * R[k][j];
  }
}
```

Plain Old Sequential C Input

This algorithm is also easy to raise to polyhedral representation

Modified Gram-Schmidt QR Parallelized

```
// prologue elided
for (int i = 0; i <= 1022; i++) {
  reduction_for (int j = 0; j <= 1023; j++)
    nrm += A[j][i] * A[j][i];
  nrm[i] = sqrt(R[i][i]);
  doall (int j = 0; j <= 1023; j++)
     Q[j][i] = A[j][i] / R[i][i];
  // barrier
  doall (int j = 0; j <= - i + 1022; j++) {
    for (int k = 0; k <= 1023; k++)
      R[i][1+i+j] += Q[k][i] * A[k][1+i+j];
    doall (int k = 0; k <= 1023; k++)
      A[i][j] -= Q[k][i] * R[i][1+i+j];
    // barrier
  }
  // barrier
}
// epilogue elided
```

Here, the scheduling algorithm finds coarse-grained parallelism

Result, after scheduling

Householder QR

```
#define M 1024
#define N 1024
void hh(float A[M][N], float Rdiag[N]) {
   int i, j, k;
   for (k = 0; k < N; k++) {
      float nrm = 0;
      for (i = k; i < M; i++)
         nrm = hypot(nrm, A[i][k]);
      if (nrm != 0) {
         if (A[k][k] < 0)
            nrm = -nrm;
         for (i = k; i < M; i++) {
            A[i][k] = A[i][k] / nrm;
         A[k][k] = A[k][k] + 1;
         for (j = k+1; j < N; j++) {
            float s = 0;
            for (i = k; i < M; i++)
               s = s + A[i][k]*A[i][j];
            s = -s/A[k][k];
            for (i = k; i < M; i++)
               A[i][j] = A[i][j] + s*A[i][k];
         }
      }
      Rdiag[k] = -nrm;
   }
}
```

Plain Old Sequential C Input

Raising Householder to polyhedral representation requires "if conversion" approximations, due to data-dependent predicates

Householder QR Parallelized

```
// prologue elided
for (int i = 0; i <= 1022; i++)
  for (int j = 0; j <= - i + 1023; j++)
     _hh_1(_v1[i],nrm[i]);
     _hh_2(A[i + j, i],_v1[i],_v2[i, j]);
     _hh_3(_v2[i, j],nrm[i]);
  _hh_4(nrm[i],_p1[i]);
  if (_p1[i])
     _hh_5(A[i, i],_v1[i],_v3[i]);
     _hh_6(nrm[i],_v3[i]);
     // barrier
     doall (int j = 0; j <= - i + 1022; j++)
        _hh_7(A[i + j, i],nrm[i]);
        _hh_9(s[i, j]);
     // barrier
     _hh_7(A[1023, i],nrm[i]);
     _hh_8(A[i, i]);
     // barrier
     doall (int j = 0; j <= - i + 1022; j++)
        _hh_11(A[i, i],_v4[i, j]);
        for (int k = 0; k <= - i + 1023; k++)
           _hh_10(A[i + k, i],A[i + k, 1 + i + j],<>s[i, j]);

        _hh_12(s[i, j],_v4[i, j],_v5[i, j]);
        doall (int k = 0; k <= - i + 1023; k++)
           _hh_13(A[i + k, 1 + i + j],A[i + k, i],_v5[i, j]);
// epilogue elided
```

Result, after scheduling and tiling

Here, the parallelization algorithm finds fine-grained parallelism

Various Downstream Transformations

- Tiling to match granularity of tasks to core (e.g., local memory size)
- Placing the tiles onto 1D and 2D arrays of cores
- Managing distributed local memories
- Generating explicit DMA and synchronization operations
- Multibuffering to overlap computation and communication
- Partitioning code for heterogeneous targets (hosts, accelerators)
- Unrolling and jamming for improved locality (enable SIMDization
- Converting to dataflow representation (for FPGA accelerators)
- Generating directives (e.g., OpenMP)

R-Stream also automates all of these transformations

Parallelization is only the first step!

So, what have we got?

- **A tool and algorithm for converting a sequential execution model into a streaming execution model!**
  - Particularly, to distributed memories and explicitly controlled architectures
  - Solved a "DARPA hard" problem - mapping
- And, it can emit to other execution models, e.g., we can emit to OpenMP!   (Various target architectures in progress).
- Disclaimer: various limitations (implementation, theory) need to be resolved.

What next (research)?

- Want to revisit the input language issue
  - Support higher levels of abstraction, algorithm exploration
- Need libraries of “raisable” BLASx
- Maybe we need to pick up SVM effort (SVM 2.0)
  - Many APIs (MCF, DACS/ALF, MPI-C, SPURS, QA, SCA, DRI…)
  - Extend to core/chip/board/chassis/cabinet level
  - Extend to other considerations (e.g., fault tolerance)
- Dynamism
- Mapping algorithms