Parallelizing QR Decompositions with the R-Stream Compiler

Allen Leung, Nicolas Vasilache, Benoît Meister,

David Wohlford, Richard Lethin

Reservoir Labs, Inc.

Unclassified, Distribution Statement A: Approved for public release; distribution is unlimited.

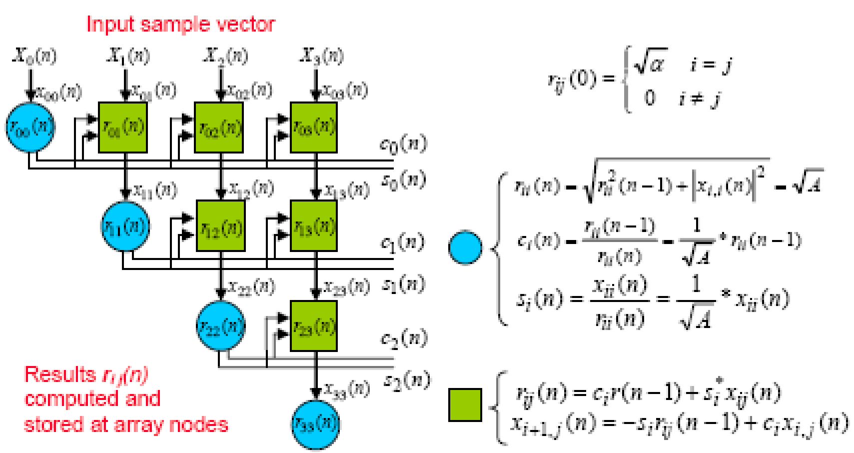
This material is based upon works supported by the Department of Defense under contract numbers F30602-03-0033, W31P4Q-07-C-0147, W9113M-07-C-0072, W9113M-08-C-0146 and W31P4Q-08-C-0319. Any opinions, findings and conclusions expressed in this material are those of Reservoir Labs, and do not necessarily reflect the views of the Department of Defense.

Outline

- QR decompositions
- Architectures
- The R-Stream compiler
- The polyhedral model and the scheduling algorithm
- Unified tradeoff between parallelization and locality
- R-Stream QR decompositions:
  - Givens
  - Modified Gram-Schmidt
  - Householder
- Current weaknesses and future work
- Conclusion and remarks

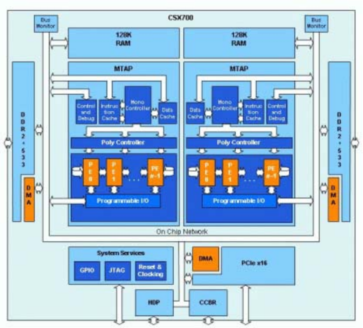
QR Decompositions

- 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:

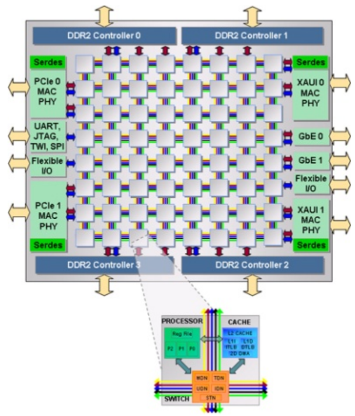


Efficiencies of the systolic form come from multidimensional, wavefront parallelism and high degrees of locality

Next Generation Multi-Core Processors/Accelerators

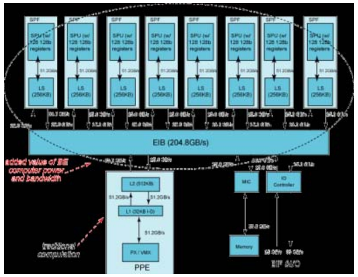


ClearSpeed CSX700

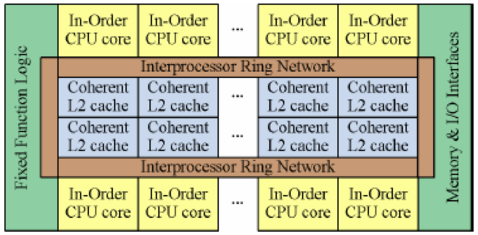


Tilera TILE64

Efficient execution on such devices requires finding mixed coarse, fine, wavefront parallelism and high degrees of locality

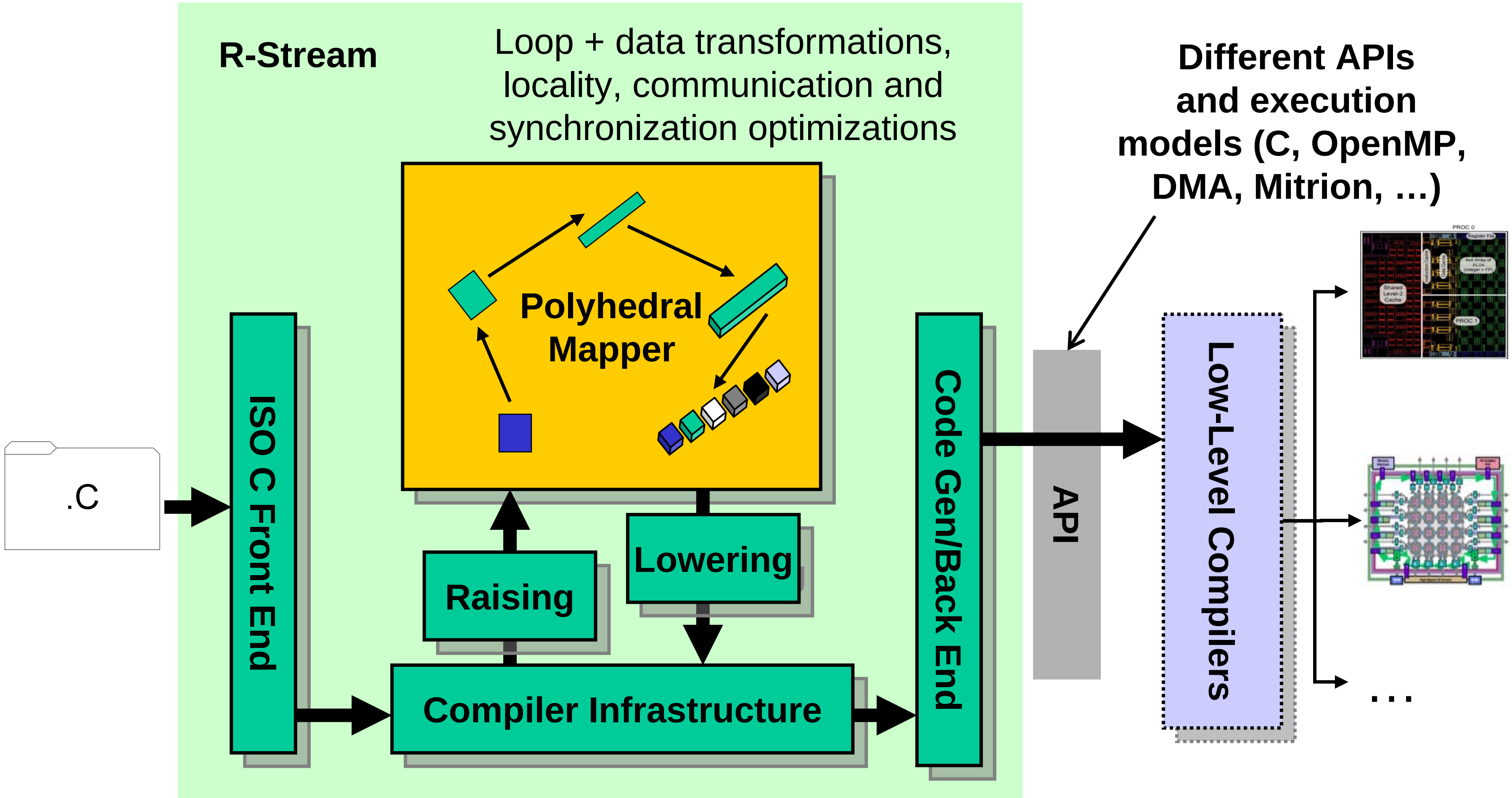


Sony, Toshiba, IBM Cell



Intel Larrabee

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

$\textbf{Affine scheduling}$: given statements $\mathrm{S}_1,\ldots,\mathrm{S}_\mathrm{n}$ and dependence relations $\mathbf{R}_{ij}$,

Find statement-wise affine schedule $\Theta = (\Theta_{\mathrm{S}_1},\ldots,\Theta_{\mathrm{S}_\mathrm{n}})$

$\Theta_{S_i}(x)$ 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

Our Scheduling Algorithm

- Computes an affine schedule that *simultaneously*
  - maximizes the amount of coarse-grained parallelism (both synchronization-free and pipelined)
  - maximizes the amount of locality
- New integer linear programming formulation, based on ideas from Bondhugula et al. [PLDI'08] and Megiddo and Sarkar [SPAA '97]
- Our algorithm maximizes

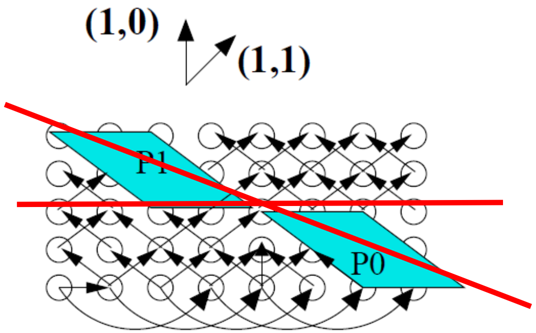
$$\sum_{l \in \text{loops}} w_l \, p_l + \sum_{e \in \text{loop edges}} u_e \, f_\text{e}$$

benefits of improved locality

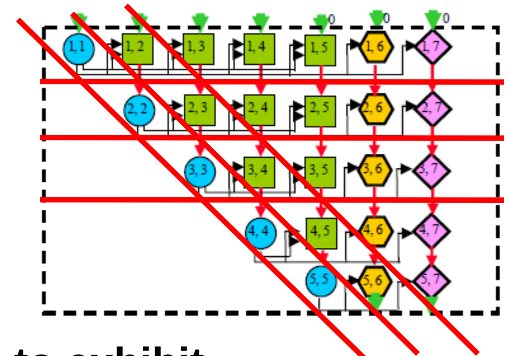
benefits of parallel execution

- $P_l = 1$ iff loop l can be executed in parallel
- $f_e = 1$ iff loop edge e can be legally fused
- w_l and u_e are problem/architecture specific parameters

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;
```

←

New optimization frames the tradeoffs between *parallelism and locality* in a single ILP

Givens QR

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

$$
G(i,j,\theta) = \begin{array}{cc}
\begin{bmatrix}
1 & \ldots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & & \vdots & & \vdots \\
0 & \cdots & \cos(\theta) & \cdots & \sin(\theta) & \ddots & 0 \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
0 & \cdots & -\sin(\theta) & \cdots & \cos(\theta) & \cdots & 0 \\
\vdots & & \vdots & & & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{bmatrix} & \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \\
\begin{array}{ccccccc} & & i & & j & & \end{array} &
\end{array}
$$

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)

Applying the New 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

$$\Theta_{S0}(i,j) = [i, i+j]$$

$$\Theta_{S1}(i,j) = [i, i+j]$$

$$\Theta_{S2}(i,j) = [i, i+j]$$

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

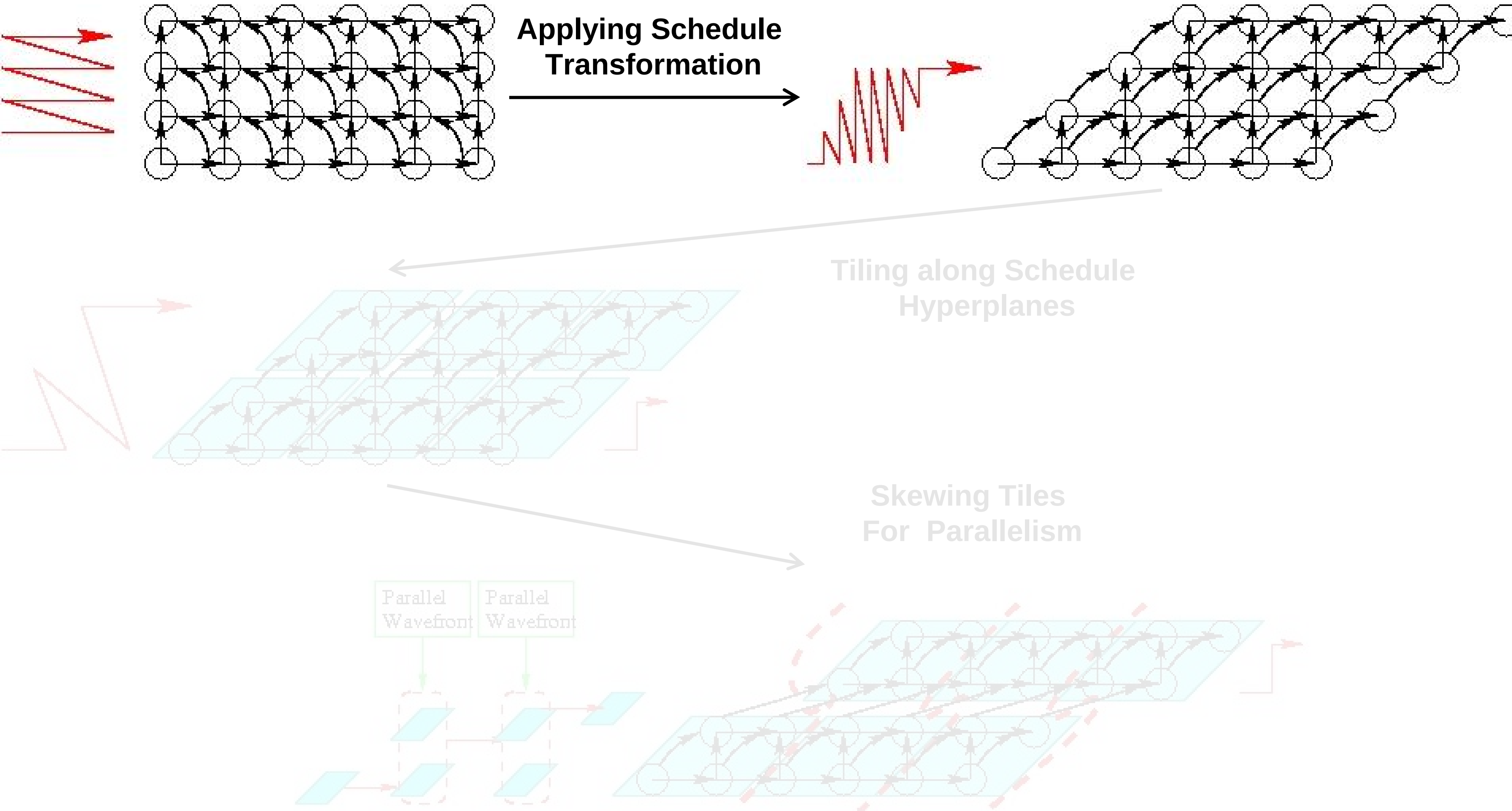
Schedule

```
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
  }
}
```

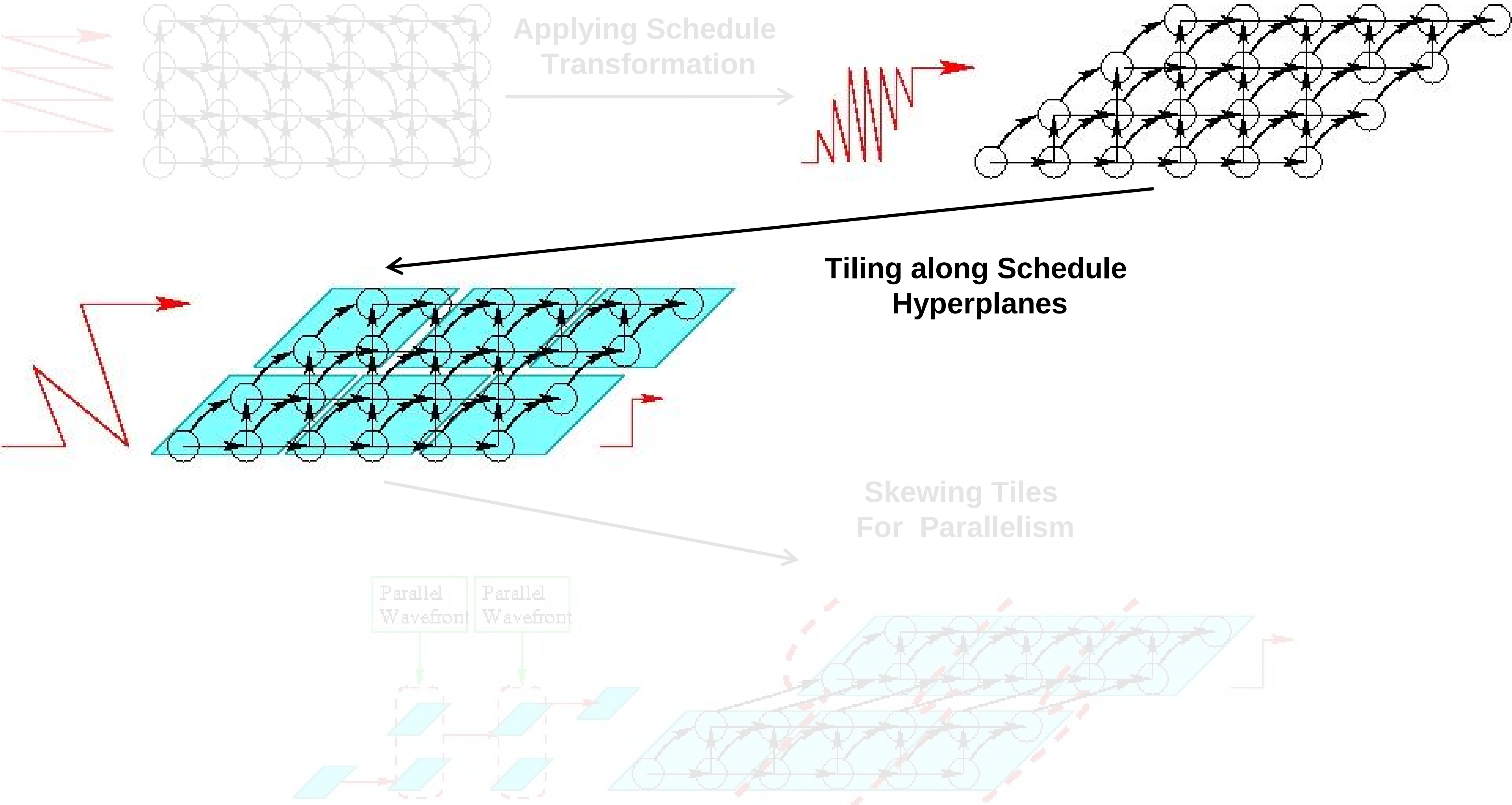
Before

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)

```
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
                 }
              }
           }
        }
     }
}
```

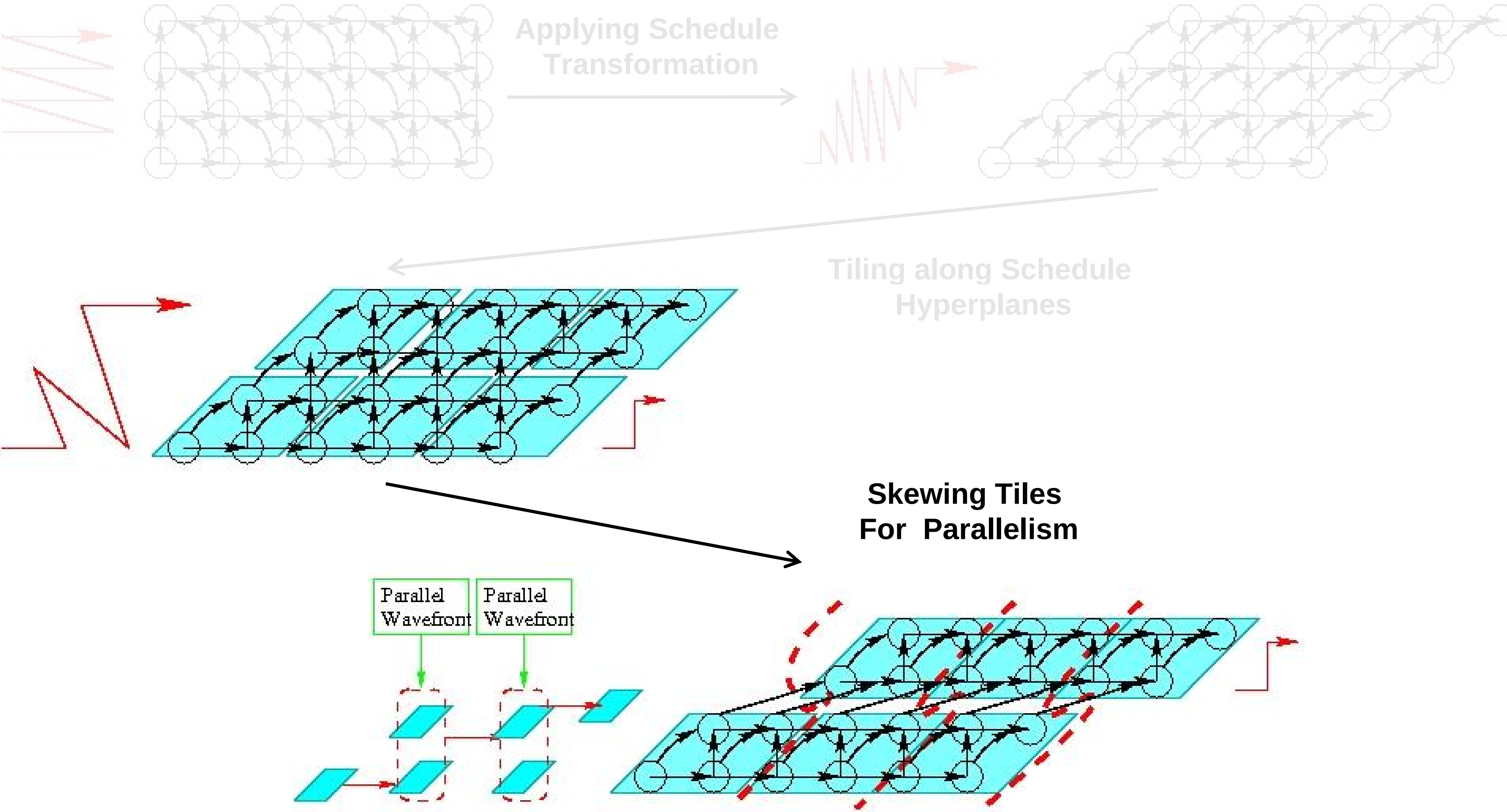
Before

After

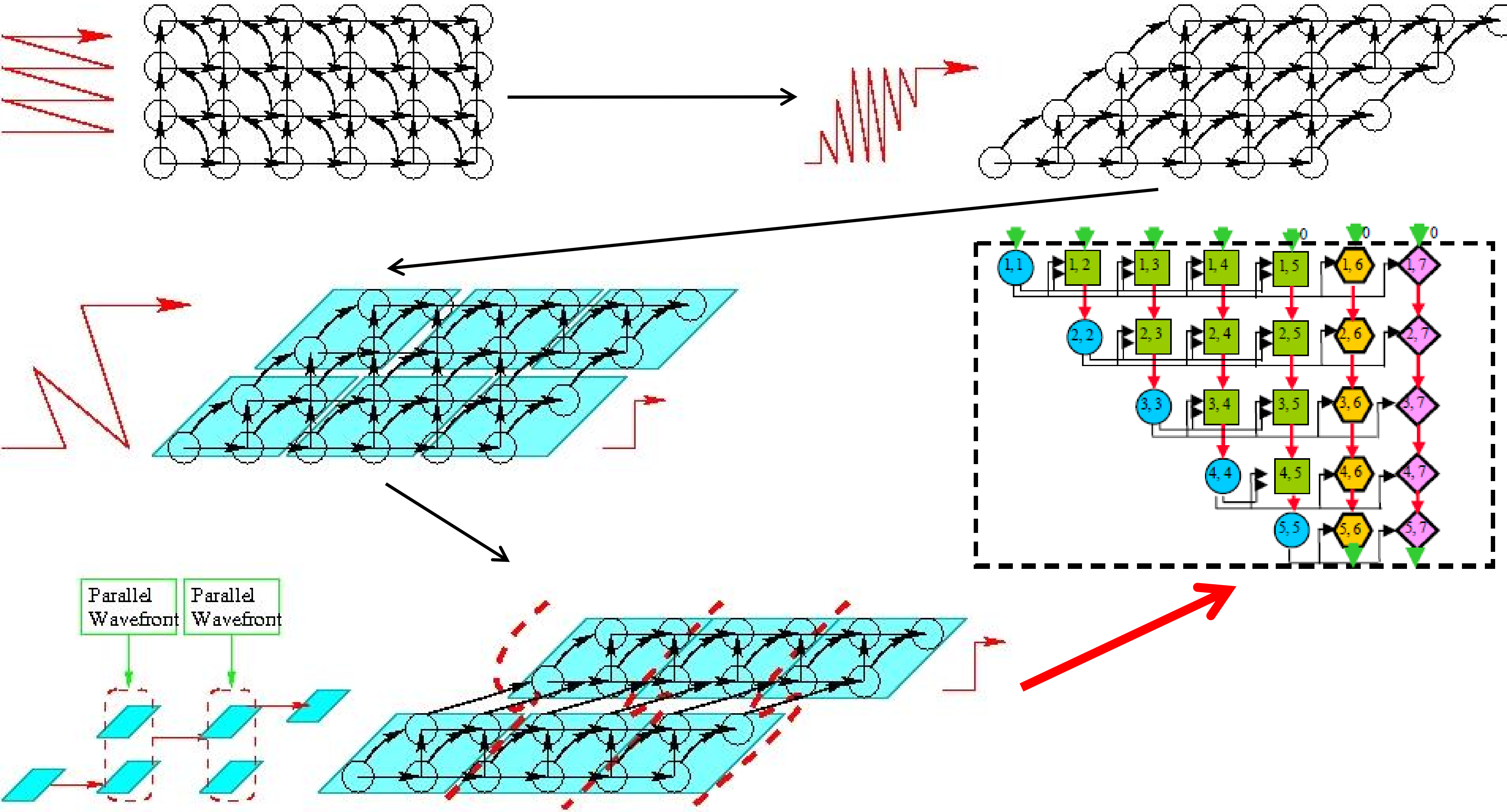
```
for (int i = 0; i <= 78; i++) {
  doall (int j = max(i-15, (4*i+ 4) / 5); j <= min(i, 63); j++) {
    // Tiled loops for S1, S2, S3 omitted
    doall (k = 0; k <= min(7, ( - i + j + 15) / 2); k++) {
      for (l = 64 * i -64 * j;
           l <= min(64*i-64*j+63, 16*j+15, 1022); l++) {
        for (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

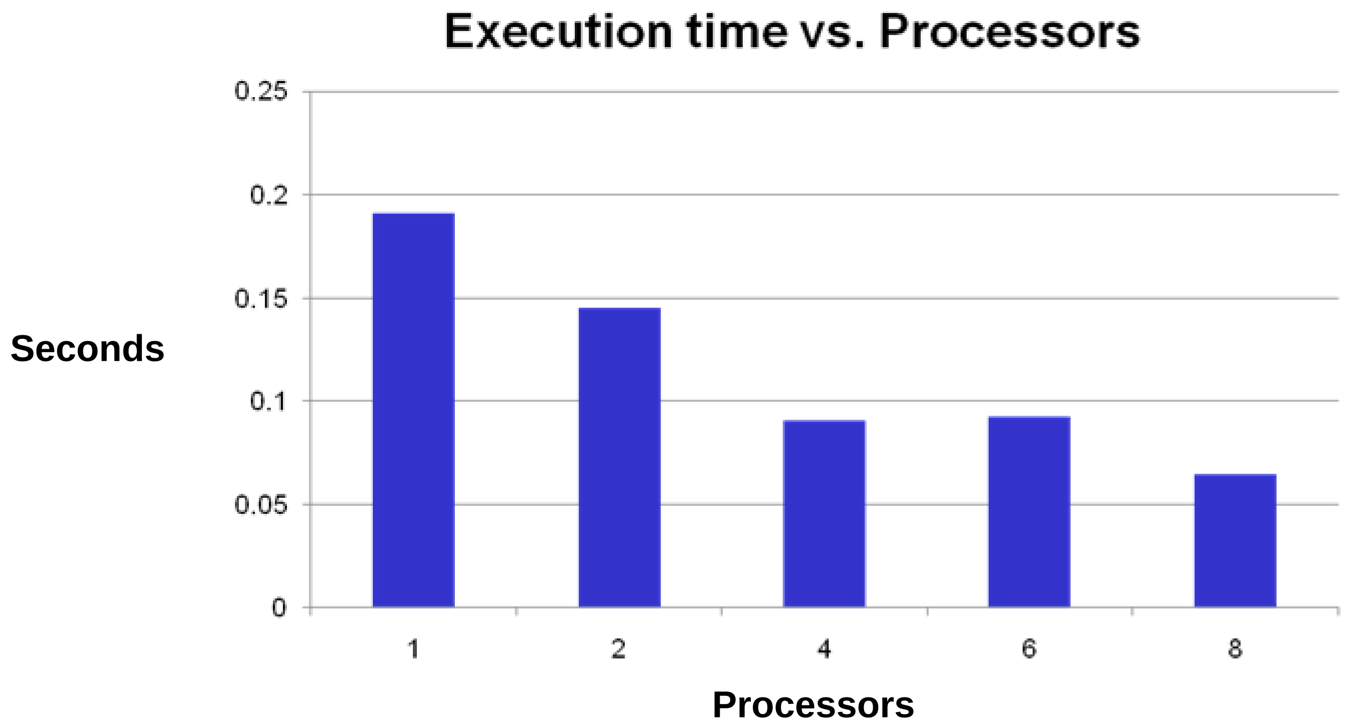
2-D Analogy (Skewing the Tile Space)



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
```

Here, the parallelization algorithm finds fine-grained parallelism

Result, after scheduling and tiling

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!

Current Weaknesses and Future Work

- Array expansion is key to removing false dependencies
  - Current implementation cannot fully remove all
  - Better algorithms known and are in implementation
  - E.g., demand-driven array expansion
- Capacity of ILP solvers
- Tuning, capability of downstream phases
- Making the LLC “sing” (e.g., SIMDization)

- More detailed comparisons with hand-mapped versions in the literature
  - E.g., vs. known systolic forms for Givens, Gram-Schmidt

Conclusion and Remarks

- Polyhedral form admits more complex algorithms than classic optimizers admit
  - Including these three QR algorithms
- New scheduling algorithm has the ability to extract relevant complex schedules trading parallelism and locality
- Addresses new multicore and accelerator architectures
- Input programs expressed in plain C, without maps, etc.
- General representation and methods - a route for global optimization of even more complex codes
  - e.g., an entire filter
- Targets different execution models (distributed, hetero, SMP, …)
- R-Stream provides implementation of the mapping sequence

Enables tackling more advanced compiler research challenges