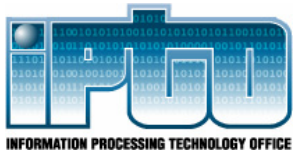


R-Stream: A Parametric High Level Compiler

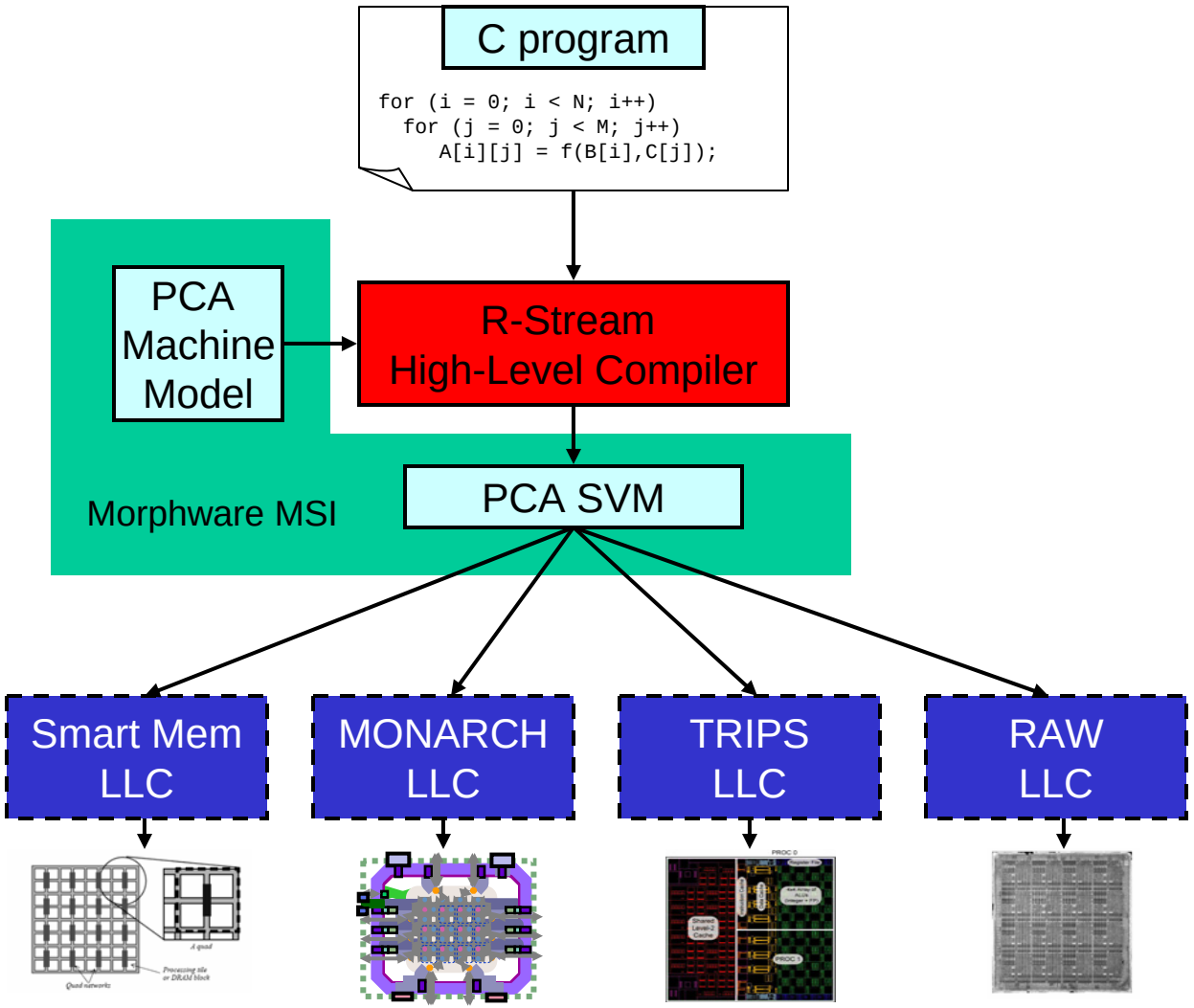
Eric Schweitz, Richard Lethin, Allen Leung, Benoit Meister
Reservoir Labs, Inc.

This work was sponsored by DARPA IPTO in the Polymorphous Computing Architectures program with Dr. William Harrod as Program Manager; contract number F30602-03-C-0033.

Outline

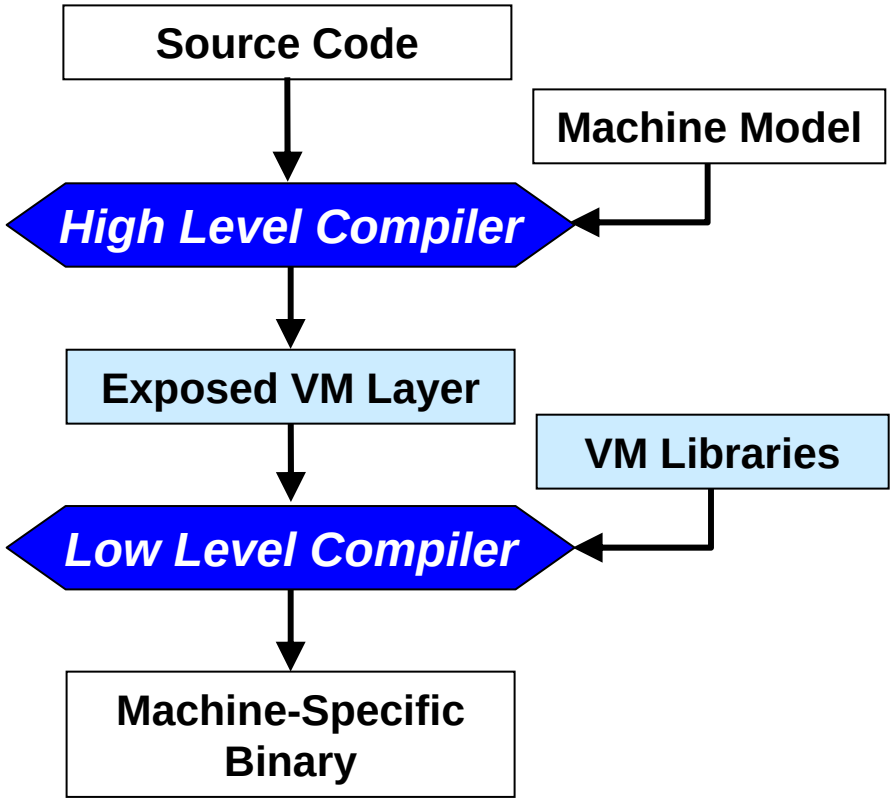
- **Introduction**
  - Morphware Forum and PCA
  - Applications
  - Challenges
- **R-Stream: High-Level Design**
- **Polyhedral Model**
- **Preliminary Results on IBM Cell**
- **Conclusion**

Morphware: Phased Compilation

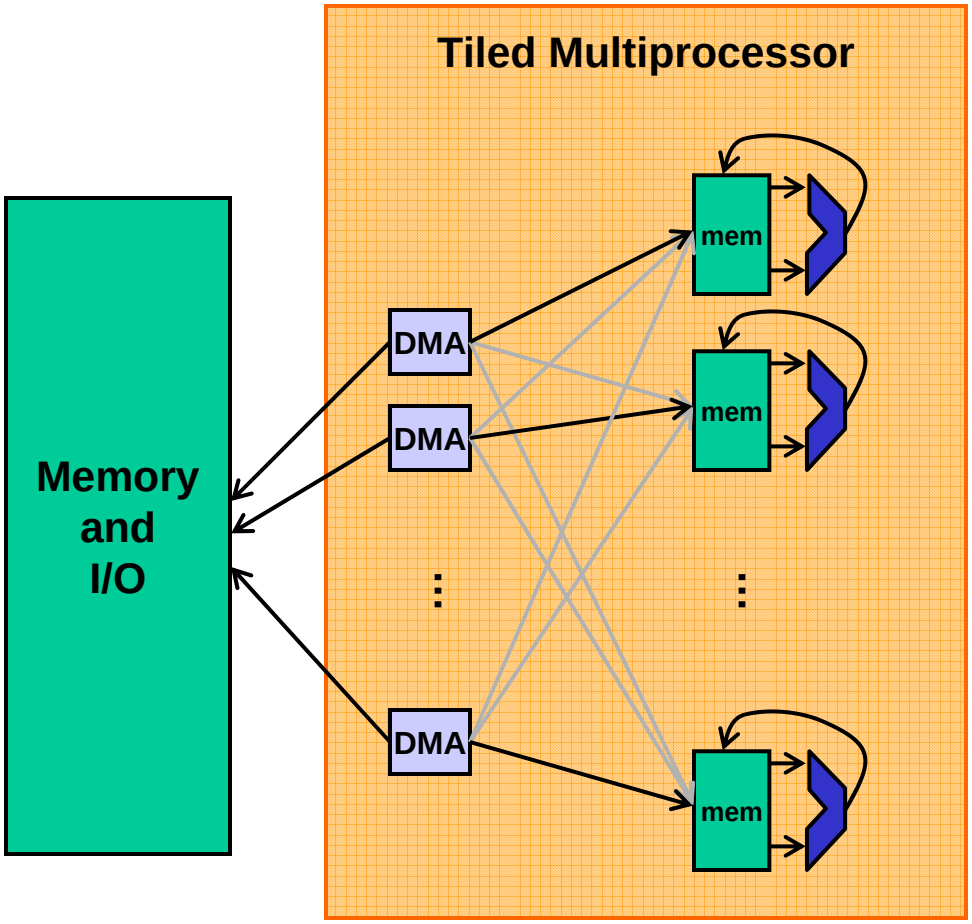


Morphware Stable Interface

- **MSI exposes an open layer between compile stages**
  - defines a computational model
  - provides a productivity layer
- ***Productivity* layer**
  - enables separate development of HLCs and LLCs
  - eases expansion to new targets
  - enables development of implementation libraries for use with LLCs

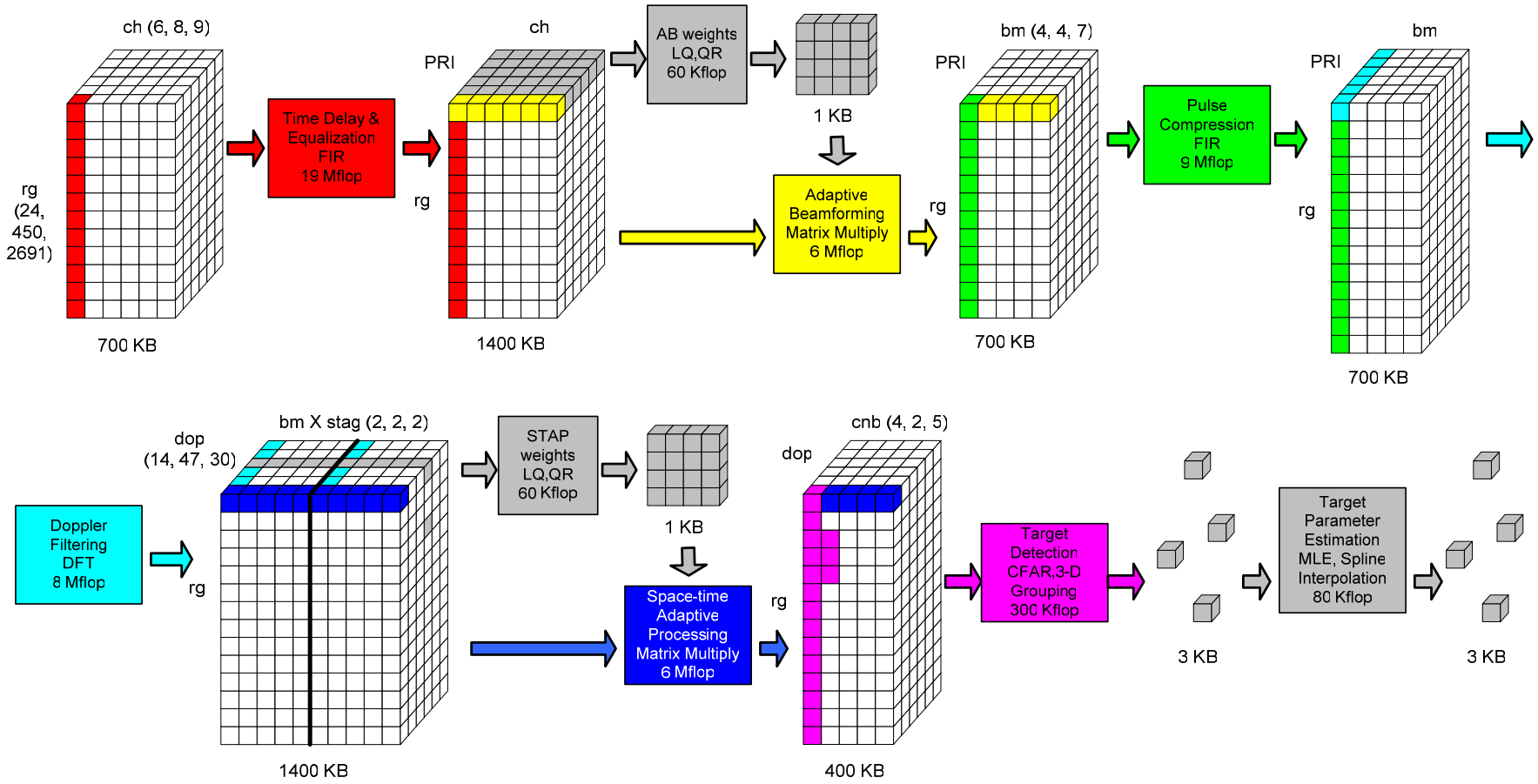


Exploiting Locality and Parallelism



- **High performance at low(er) power – FLOPS/W**

Applications: GMTI



Compiler Challenges

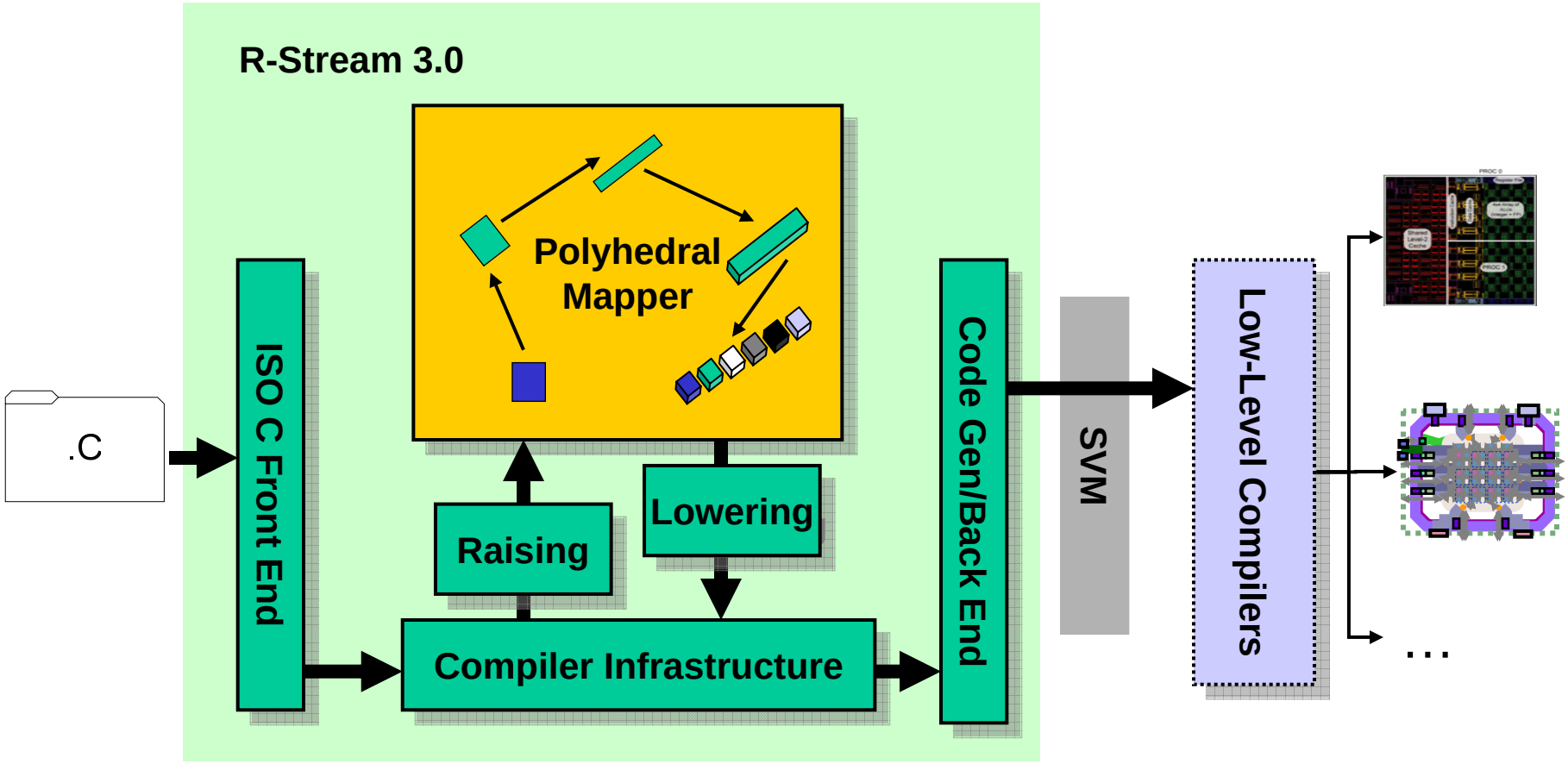
- **Automatic management of small on-chip scratchpad memories**
- **High performance requires locality of reference**
- **Exploiting parallelism from the source code**
  - Traditional approach: exploit the parallelism in loops
    - good source of data-parallelism
  - Large body of research to draw upon
    - including the polyhedral model; still, implementation lags research
- **Locality *and* parallelism**
  - a.k.a., Stream Processing
  - Pipelining data between different tasks across the PEs on chip

R-Stream: High-Level Design

R-Stream Design Goals

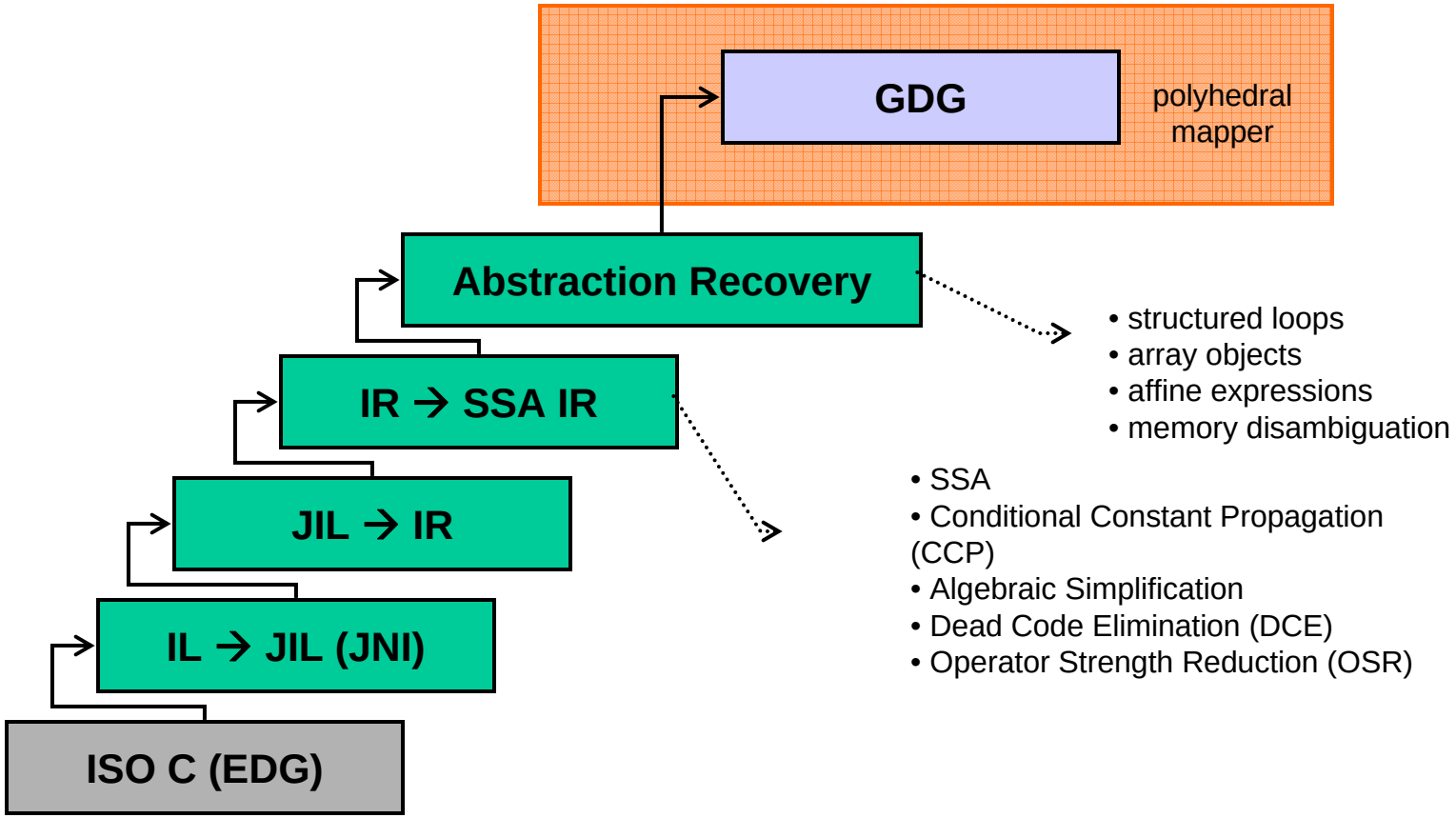
- **Static mapping**
- **Optimizations:**
  - Parallelism extraction
  - Loop transformations
  - Locality optimizations
  - Data layout transformations
  - DMA generation
  - Communication optimizations
  - Data distribution and processor mapping
- **Combining optimizations**
- **Flexible platform for experimentation**

High-Level Compiler

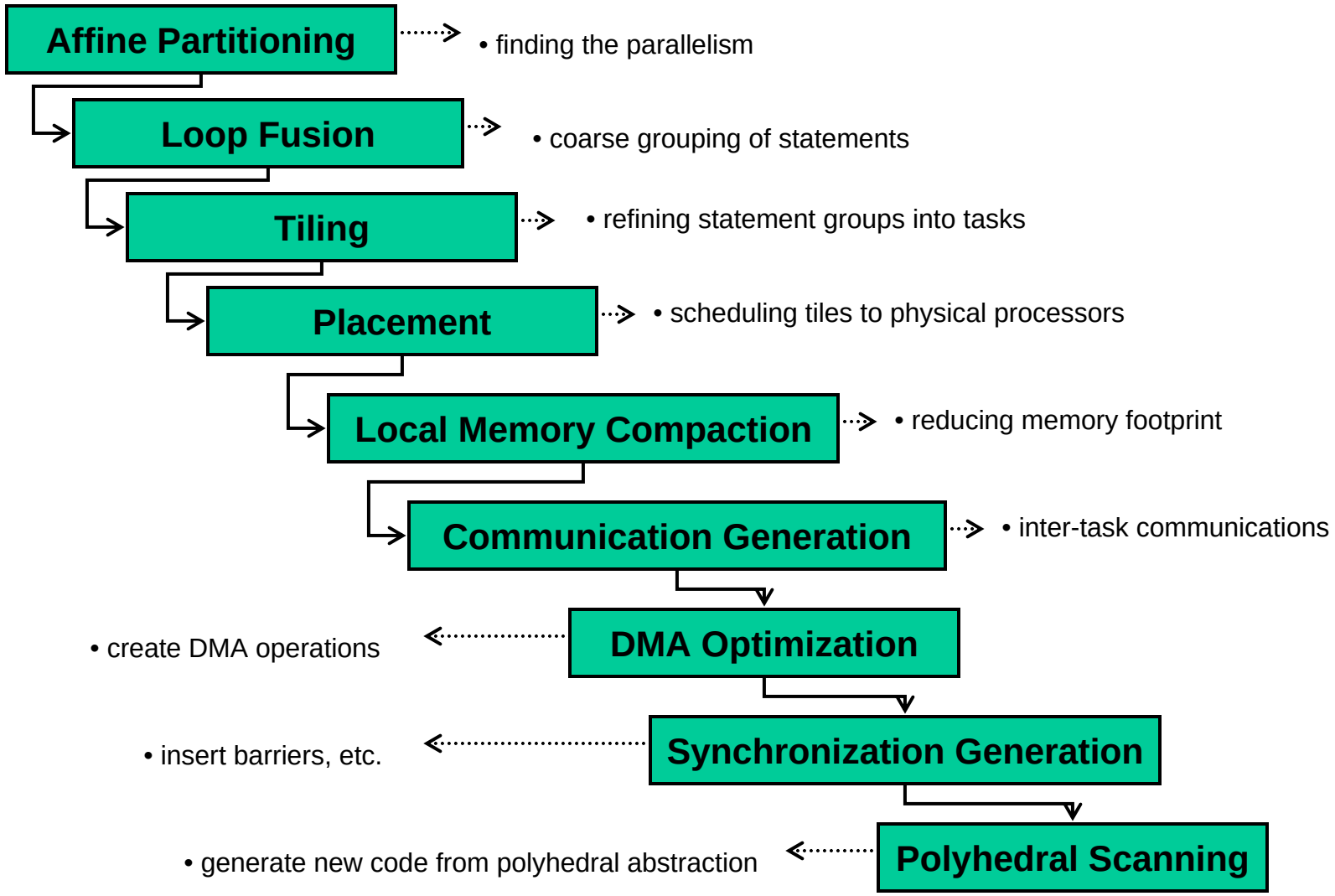


Compiler Infrastructure: Raising

- Very much like a traditional compiler in structure
- Goal is different: want to raise abstraction for mapping

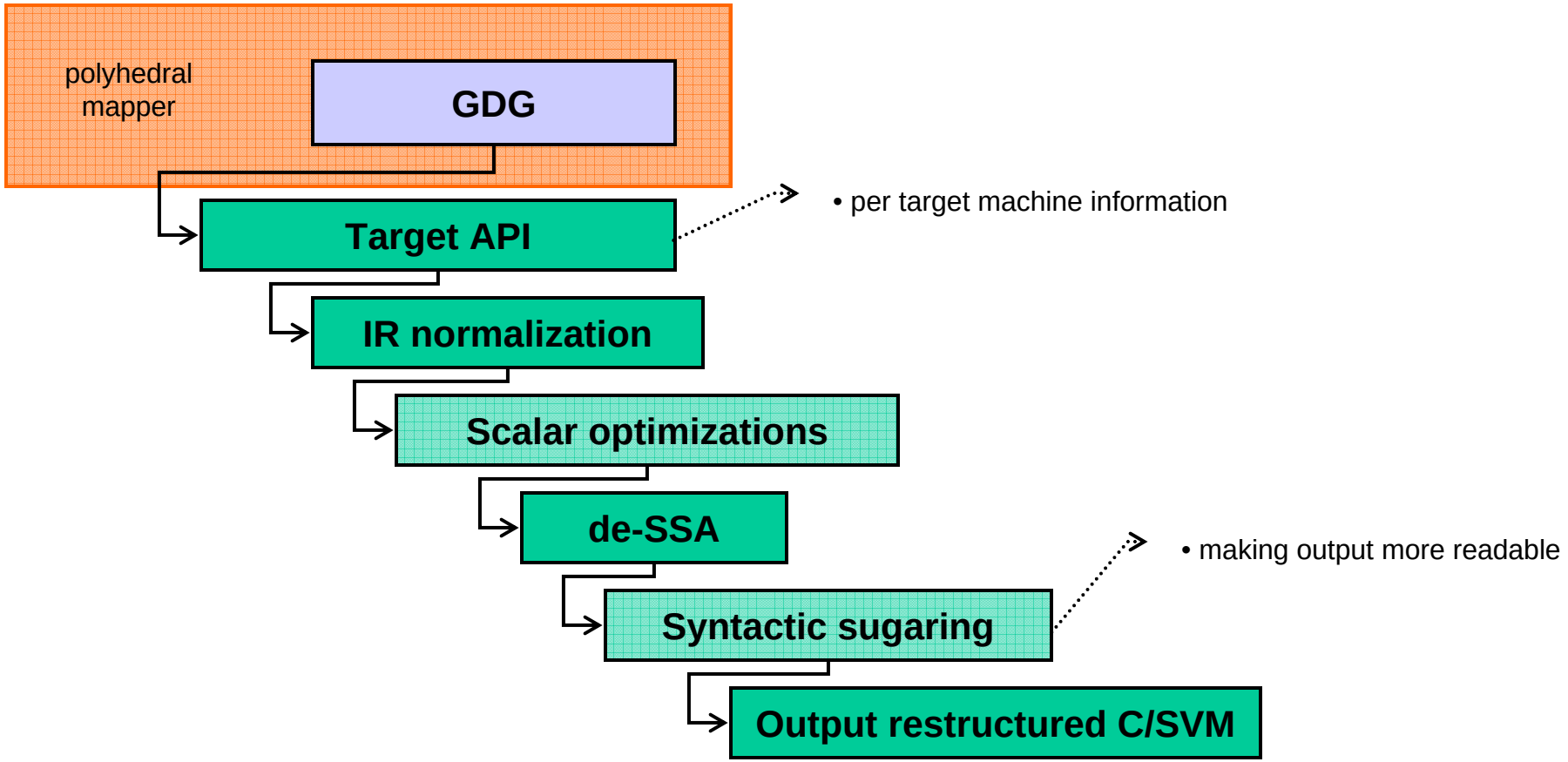


Polyhedral Mapper



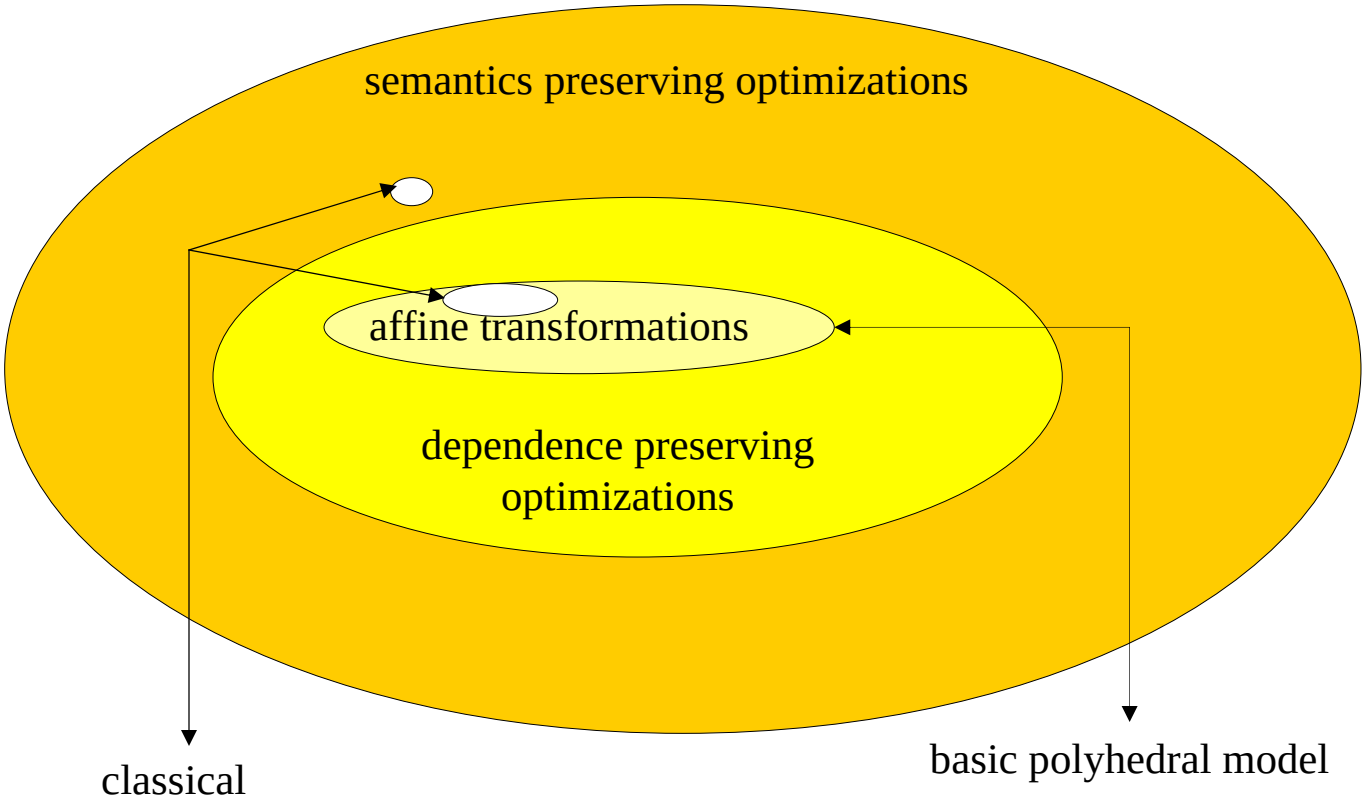
Compiler Infrastructure: Lowering

- Again, similar to traditional compiler in structure
- Goal: lower the mapped code for output to the LLC



The Polyhedral Model

Polyhedral Model



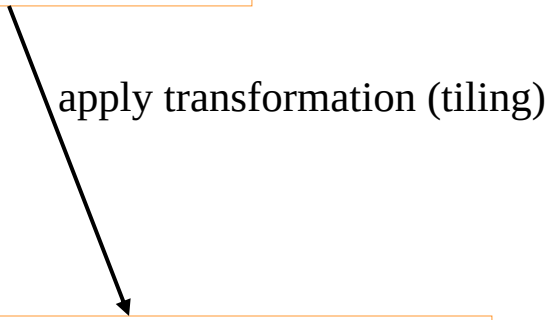
Why the Polyhedral Model?

- **A natural representation for static control programs**
- **All our optimizations can live in the same mathematical space.**
  - Make it easier to combine optimizations
- **More powerful modeling and computation techniques**
- **Parametric analysis**
- **Clean mathematical model to reason about loops**
- **Extensible**

Classical Loop Transformations

Abstract syntax tree

```
for(j=0; j<N; j++)
  for(i=0; i<N; i++)
    S(j,i);
```




```
for(i=0;i<N/M;i++)
  for(j=0;j<N/M;j++)
    for(ii=0;ii<M;ii++)
      for(jj=0;jj<M;jj++)
        S(j+jj,i+ii);
```

Often limited to:

- Coarse dependence summary: e.g., direction and distance vectors
- Single statement transformations
- Perfectly nested loops
- Unimodular transformations
- Specific ordering of phases

Program representation tied to syntax

Often divided into phases:

1. modeling
2. analysis
3. code generation

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

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

Variables and access functions

B

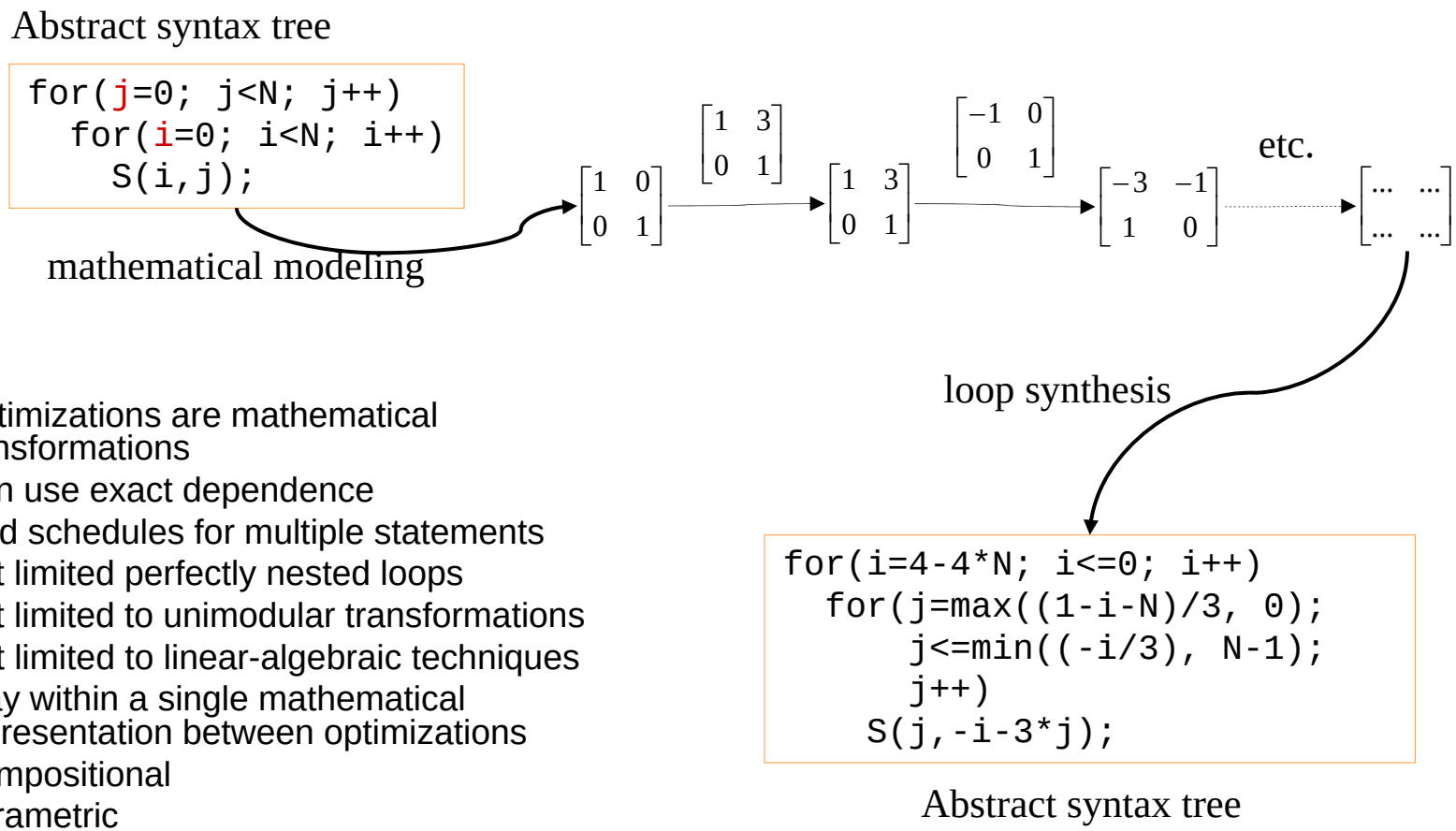
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

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 tie these components together

Transformations in the Polyhedral Model



- Optimizations are mathematical transformations
- Can use exact dependence
- Find schedules for multiple statements
- Not limited perfectly nested loops
- Not limited to unimodular transformations
- Not limited to linear-algebraic techniques
- Stay within a single mathematical representation between optimizations
- Compositional
- Parametric
- Loop synthesis at the very end to generate code

Subsumes Classic Loop Transformations

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    S(i,j);
```

unimodular

permutation

```
for(j=0; j<N; j++)
  for(i=0; i<N; i++)
    S(i,j);
```

$$\theta(i,j) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

reversal

```
for(i=N-1; i>=0; i--)
  for(j=0; j<N; j++)
    S(j,i);
```

$$\theta(i,j) = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

skewing

```
for(i=0; i<N; i++)
  for(j=α*i; j<N+α*i; j++)
    S(i,j-α*i);
```

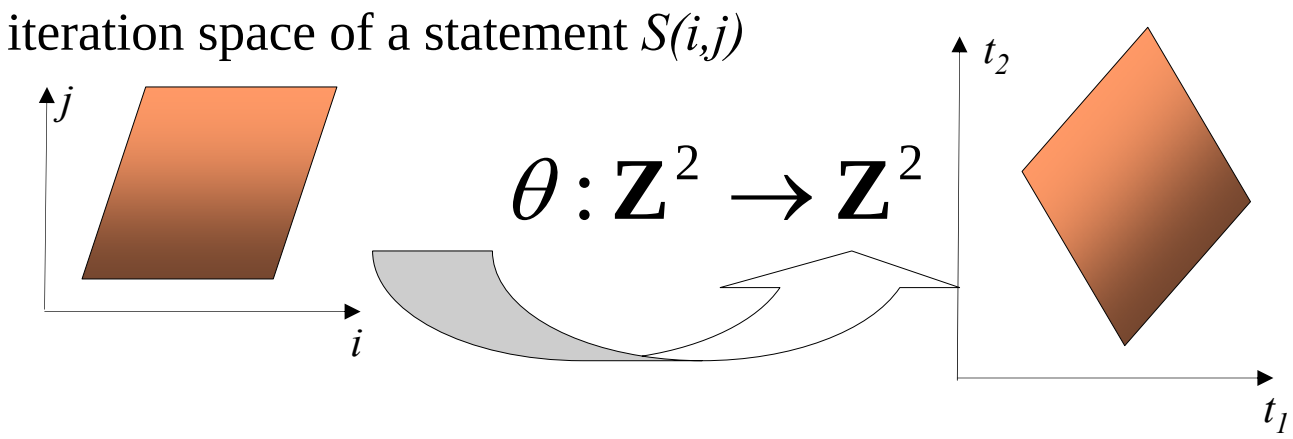
$$\theta(i,j) = \begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

scaling

```
for(i=0; i<α*N; i+=α)
  for(j=0; j<N; j++)
    S(i/α,j);
```

$$\theta(i,j) = \begin{bmatrix} \alpha & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

Loop Transformations as Scheduling



Schedule θ maps iterations to **multi-dimensional** time

A feasible schedule *must* preserve dependencies

Loop transformations/synthesis mean generating code to execution iterations of a loop in the **lexicographical** order of time

Polyhedral Challenges

- **Algorithms for solving the problems**
  - may be very expensive
  - may not scale (can be super-exponential in time and space)
  - may not exist (specific problem not studied in the literature)

- **Power is a double-edged sword**
  - can find “all” parallelism
  - can find infinite families of legal schedules
  - must know what parallelism is best suited for target architecture

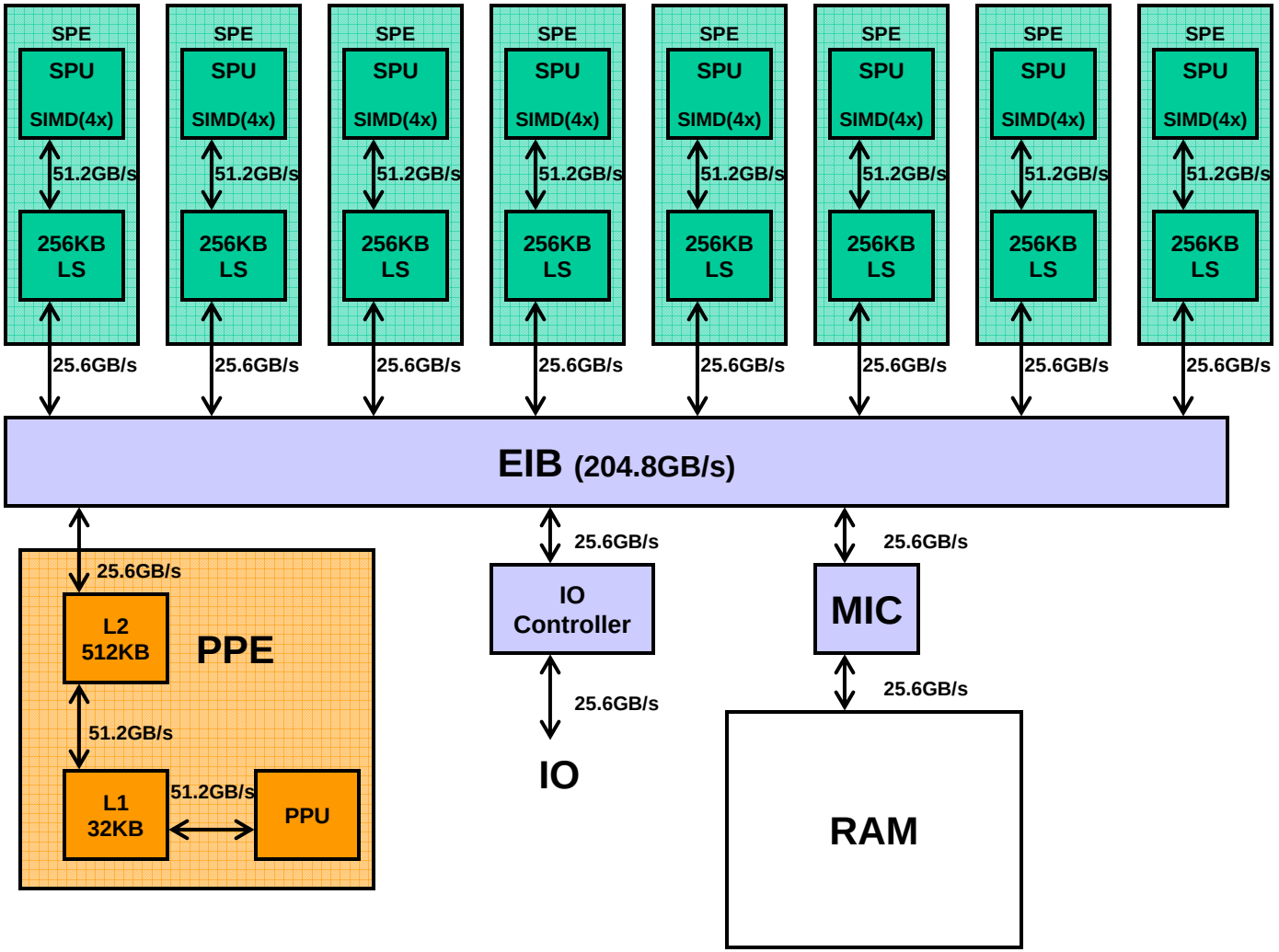
- **Application of transformations is easy**
  - knowing when, where and why (or why not) to apply them can be much more difficult to decide
  - must decide based on the target architecture’s features and limitations

Specific Reservoir Innovations

- **Parallelism extraction**
  - extensions to work of Lim and Lam, and others
  - extensions deal with locality and communication minimization
- **Local memory compaction**
  - rearranges data layout in local memory to improve memory usage
  - extensions to the algorithms of Schreiber and Cronquist
- **DMA optimization**
  - compute min-cost sets of DMA transfers
  - extension of algorithms for generating efficient message passing by Paek, et.al.
- **Parametric tiling**
  - generality: operates on collections of imperfect loop nests
  - find a best tiling, defined in terms of polynomial objective functions and constraints, over a space of possible tile sizes
  - considers re-use, memory footprint, etc.

R-Stream on Cell

Cell Architecture



Kernel Codes

```
for (int i = 0; i < N; i++) {
  C[i] = A[i] * A[i] + B[i] * B[i];
}
```

N=8*1024*1024

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    C[i][j] = 0;
    for (int k = 0; j < N; j++) {
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
  }
}
```

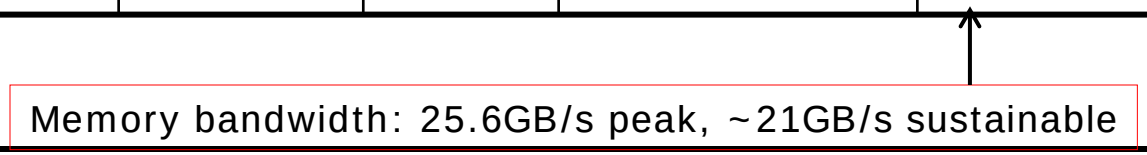
N=1024

Single precision floating point only

Vector Sum of Squares Results

```
float (* restrict A_local) __attribute__((aligned(128))) = ...;
float (* restrict B_local) __attribute__((aligned(128))) = ...;
float (* restrict C_local) __attribute__((aligned(128))) = ...;
for (int i = 0; i < N; i++) {
  C_local[i] = A_local[i] * A_local[i] +
               B_local[i] * B_local[i];
}
```

Trials	SIMDized	Pipelined	N	Time	GFlops/SPU/s	Bandwith (GB/s)
1024	no	no	2048	9.25s	0.32	10.38
1024	no	yes	2048	5.62s	0.53	17.08
1024	yes	no	2048	5.29s	0.57	18.15
1024	yes	yes	2048	4.83s	0.62	19.88
4096	yes	yes	2048	18.21s	0.66	21.09



Matrix Multiply Results

- **Not bandwidth bound; plenty of parallelism**
- ***Key component to performance:***
  - excellent SIMDization on the SPUs
- **Working on closing the gap between the HLC and LLC**
  - working with IBM; new version of their compiler “in the mail”
  - anticipate results will achieve closer to 50% of peak

Trials	SIMDimized	Pipelined	Time	GFlops/SPU/s
64	no	no	79.4	0.202
64	no	yes	78.5	0.204
64	yes	no	13.36	1.20
64	yes	yes	12.55	1.27

25.6 GFlops/SPU/s peak

Research Challenges

- **Some architectures just now realizing silicon**
  - Much more experimentation needed
  - What optimizations are needed? Which work best?
- **Scalability: how big a problem can we map?**
- **What are the implications for parallel programming languages?**
- **How can we guide the programmer constructively?**
- **More work on optimization for the memory hierarchy**
- **How much performance do we sacrifice with layered compilation?**
  - Need strategies to ameliorate this potential gap

Conclusion

- **Investigation and experimentation with phased compilation**
  - PCA, Morphware Forum, SVM
- **Compiler that can automatically**
  - find parallelism
  - manage small scratchpad memories
  - generate communications
- **Advancement of polyhedral model**
  - improving published algorithms
  - new algorithms and optimizations
  - mapper uses polyhedral algorithms upon polyhedral representation
- **Some initial results of applying polyhedral mapper on Cell**
  - working on closing the gaps between HLC and LLC
- **Still more work to be done…**