

# What the experience of developing a high-level automatic application mapping tool is teaching us about execution models and virtual machines for multi-core processors

**Richard A. Lethin**  
**Reservoir Labs, Inc.**

The research reported in this document/presentation was performed in connection with contract/instrument DARPA F03602-03-C-0033 with the U.S. Air Force Research Laboratory and DARPA. The views and conclusions contained in this document/presentation are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U.S. Air Force Research Laboratory, DARPA, or the U.S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

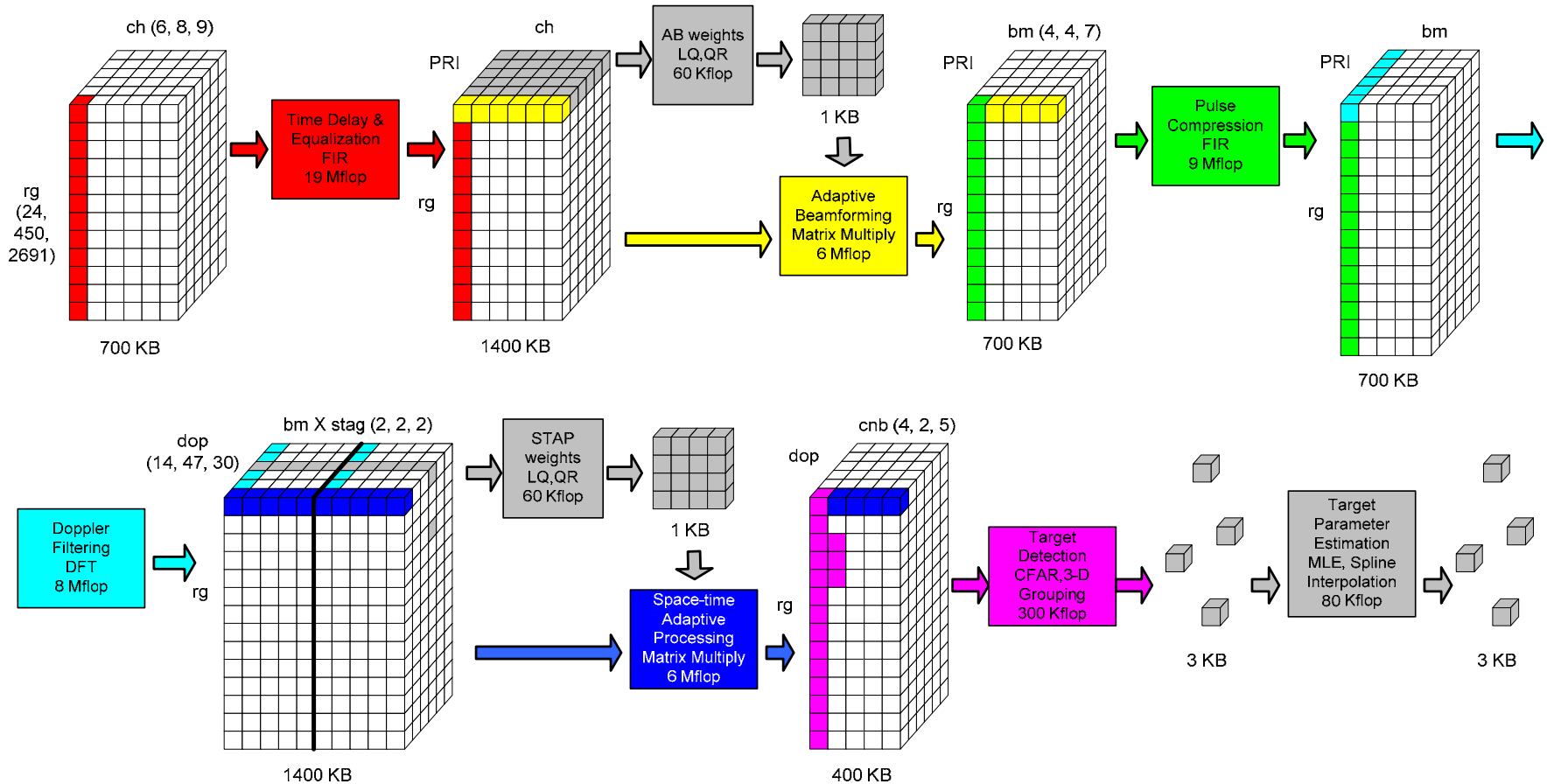
Copyright © 2007 Reservoir Labs, Inc.  
Patent Pending Technologies

# Outline

---

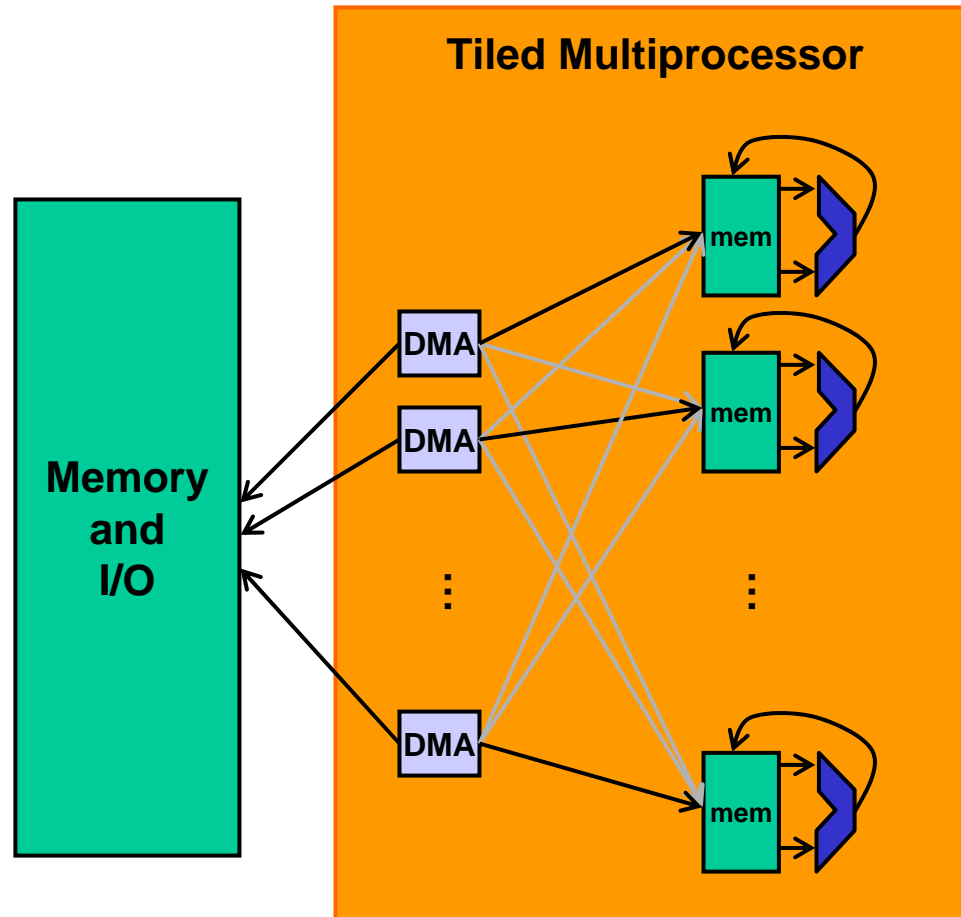
- **Developing a high level mapping and optimization tool**
  - (LOTS)
- **The Streaming Virtual Machine (SVM)**
- **The need for an open standard “VM” or API for multi-cores**

# Problem: Map HPEC Applications ...

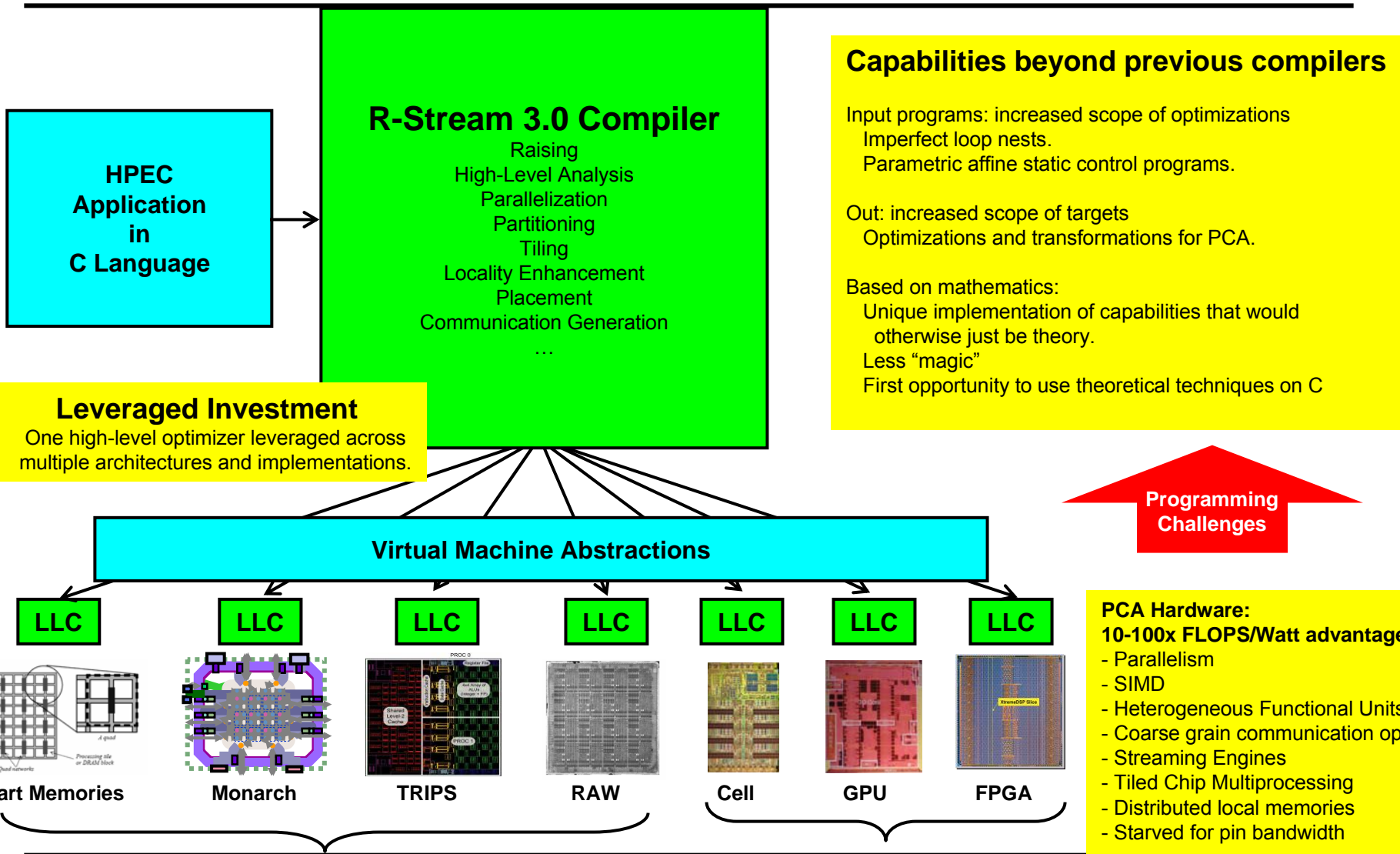


... to new architectures that offer FLOPS/W rivaling ASIC

---



# R-Stream Project



Polymorphous Computer Architectures

COTS Streaming Architectures

# High-Level vs. Low-Level Mapping

---

- **High-Level Mapping**

- Parallelization across streaming elements
- Memory layout and placement optimization
- “Tiling” choosing correct size and organization of tasks
- Improvements to locality of reference – producer consumer
- Management of communication – intra and inter chip
- Management of distributed memory
- Double buffering and high-level software pipelining

*Separation avoids duplication of effort with “vendor”-supplied compilers for particular chips*

- **Low-Level Mapping**

- For individual processor
- Instruction selection
- Register allocation
- Instruction scheduling

*Regardless of whether you agree that the separation should be explicit, it will be present, e.g., an interface in an integrated compiler*

# How R-Stream differs from “Regular Compiler”

---

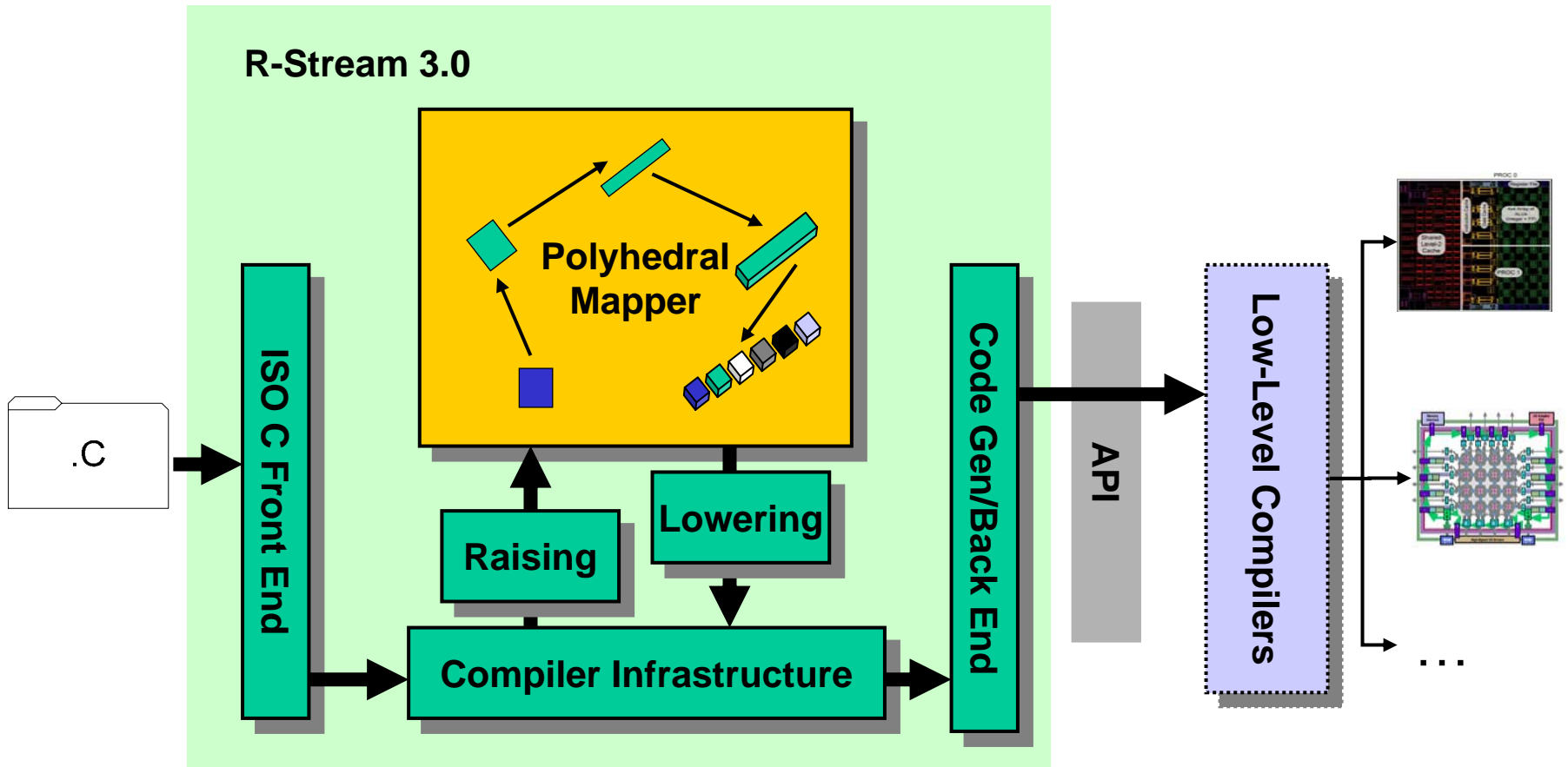
- **Oriented for HPC and HPEC application areas**
- **Find greater parallelism over larger scope of program**
- **Find more locality of reference**
- **Manage distributed (not cc-NUMA) memories**
- **High level scheduling of tasks**
- **Generate bulk communication operations**
- **Produce results in a way that a low level compiler can use**
- **Work from normal C code (OK, that’s in a regular compiler)**
  - *Though we want the program to be as abstracted from the HW as possible*
  - *Sorry – won’t do dusty deck soon – programmer must write in “abstractable” form*

# This is “static optimization,” but...

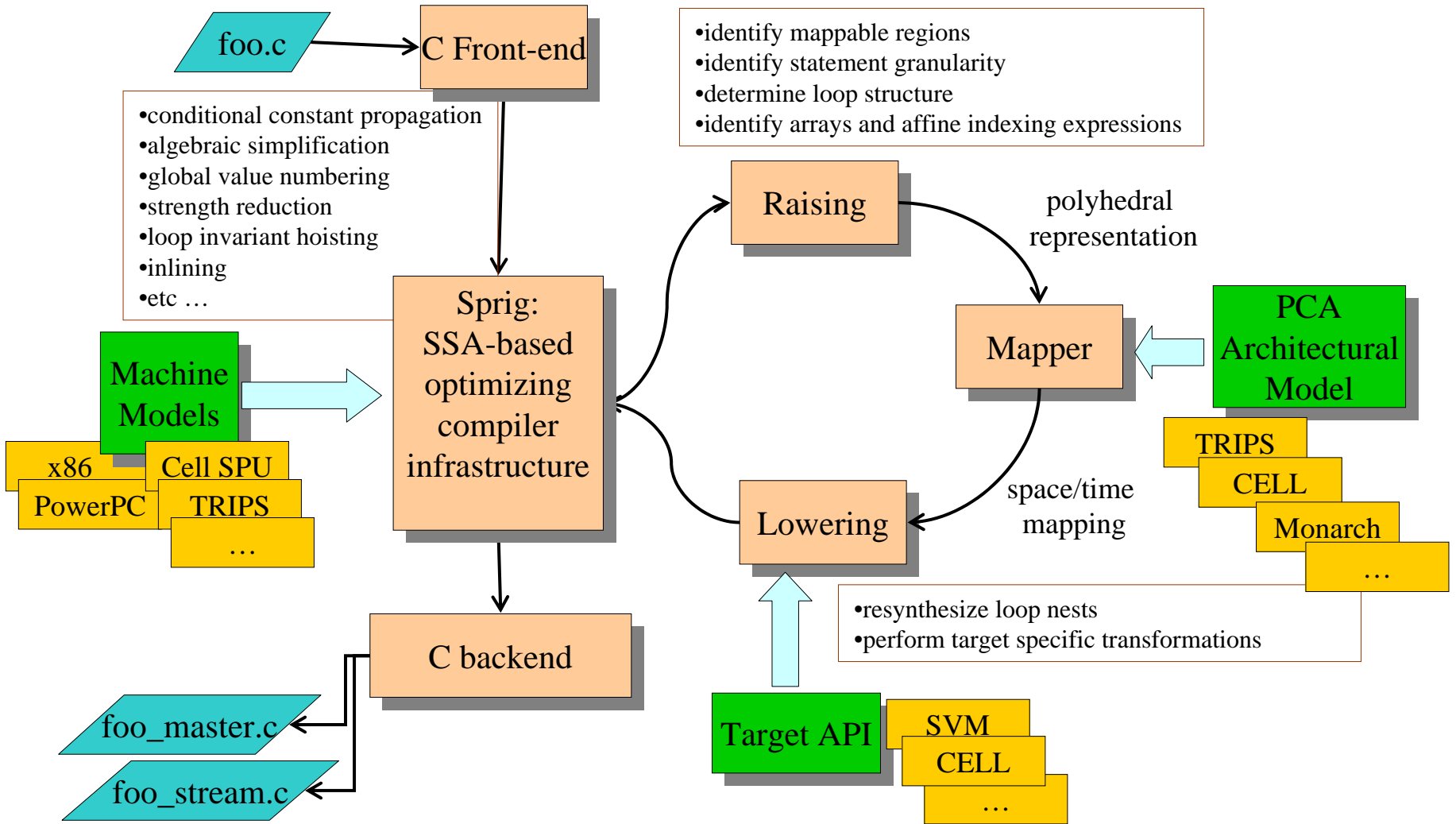
---

- **Static optimization takes advantage of the relatively unlimited resources (computation, time) for doing optimization before running the program.**
  - Allows greater effort to be applied to mapping
  - Allows consideration of greater scope of program
- **These mapping problems when formulated explicitly are *very complex* problems**
  - I don't see how dynamic approaches could work for other than relatively trivial architectures or applications
- **The result of the mapping is not fully static bindings**
  - The formalisms we are using allow for optimizing parametrically specified program forms and producing parametric mappings (not that this is easy)
- **There is plenty of research terrain to explored regarding the interaction between dynamic and static optimization**
  - Consider the analogy of the interaction of global instruction scheduling algorithms helping with out-of-order superscalar microprocessors

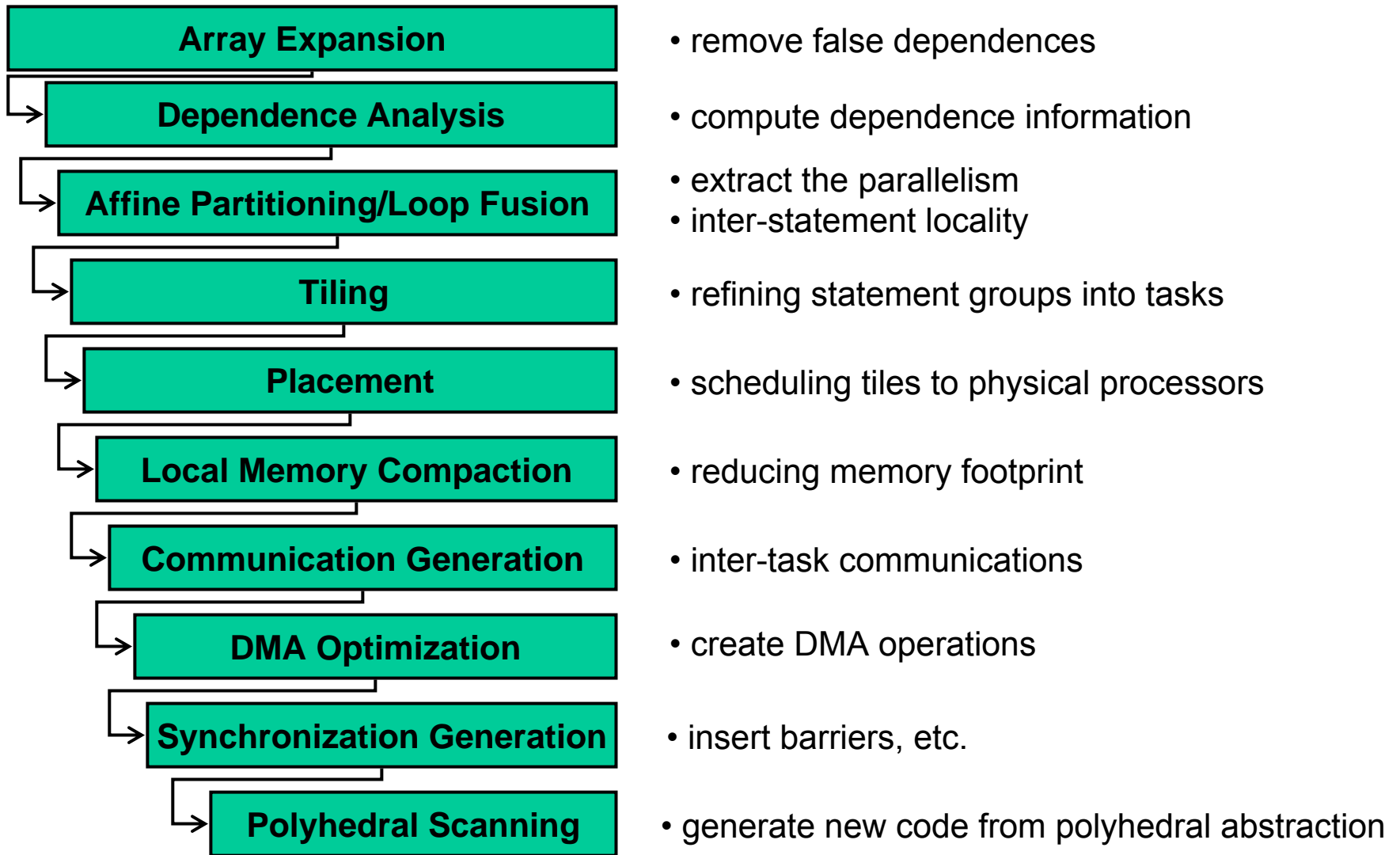
# R-Stream 3.0: High-Level Compiler



# Current R-Stream Infrastructure



# Polyhedral mapper



# Polyhedral representation

```

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];
}

```

$$\{(i, j) \mid \exists k. 2 \leq i \leq M, 0 \leq j \leq N, j = 2k\}$$

Iteration spaces as constraints (polytopes)

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

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

$$\begin{bmatrix} 2 & 0 & 0 & 0 & -4 \\ 8 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

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

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

Dependences can then be extracted from these information.  
Dependences are represented as polyhedra

Array indices as affine functions

# Polyhedral transformations

---

- **Each polyhedral statement contains the following information in constraints form**
  1. an iteration space  $D = \{x \in \mathbf{Z}^n \mid \mathbf{A}x + \mathbf{b} \geq 0\}$
  2. a set of array references  $\mathbf{A}[\mathbf{f}(\mathbf{x})]$
  3. a schedule (space/time mapping)  $\theta$ , which determines when and where a statement instance is executed
- **Combining transformations:**
  - Loop transformations  $\rightarrow$  modifies  $\theta$  (time component)
  - Processor placement  $\rightarrow$  modifies  $\theta$  (space component)
  - Data transformations  $\rightarrow$  modifies  $\mathbf{A}[\mathbf{f}(\mathbf{x})]$
- **We put the mapped regions into the internal constraints form**
- **All mappings transformations are performed in this form**
- **Resynthesize code back from the constraints form when we are done**

# Algorithmic Tools in the Polyhedral Model

---

- **Basic linear algebra**
  - Hermite Normal Form, Smith Normal Form, Gaussian Elimination, Fourier- Motzkin Elimination, ...
- **Minkowski decomposition:  $P = \text{conv}(V) + \text{cone}(R) + \text{lin.space}(L)$**
- **Set operations on polyhedra**
- **Linear programming and extensions**
  - LP, ILP, parametric LP, parametric ILP, ...
- **Farkas Lemma**
- **Computational geometry. e.g., bounding volumes computation**
- **Combinatorial optimizations**
- **Lattice point counting. e.g., Ehrhart Polynomials and related generating functions**

# Polyhedral Model Slogans

---

- **Model everything as polyhedra, linear constraints in N dimensions**
- **Can represent task, pipeline, data parallelism as affine schedules**
- **Can represent affine data transformations**
- **Do both computation and data in same framework**
- **Frameworks subsume classic optimizations fusion, scaling, interchange, reversal, skewing in single phase**
- **Coarse grained parallelization – not JUST vectorization**
  - Can do vectorization too in same framework
- **Tiling: imperfectly nested loop**
- **Memory layout & DMA optimizations**

# CELL Backend in R-Stream

---

- **Targets a minimal API**
  - Asynchronous DMA: `CELL_dma_get`, `CELL_dma_put`, `CELL_dma_wait`
  - Process control: `CELL_mapped_begin`, `CELL_mapped_end`
  - Memory management: C library
  - Synchronization: `CELL_barrier(int id)`
- **DMA restrictions bypassed in runtime layer:**
  - Source and target addresses
    - must have be naturally aligned
    - must have matching lowest 4-bits
  - No strided DMA; simulated via DMA list
- **Backend generates two separate programs, one for SPU and one for PPU**

# Matrix Multiply Example on CELL

---

```
float A[1024][1024];
float B[1024][1024];
float C[1024][1024];
for (int i = 0; i <= 1023; i++) {
    for (int j = 0; j <= 1023; j++) {
        C[i][j] = 0;
        for (int k = 0; k <= 1023; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

# Parallelizing

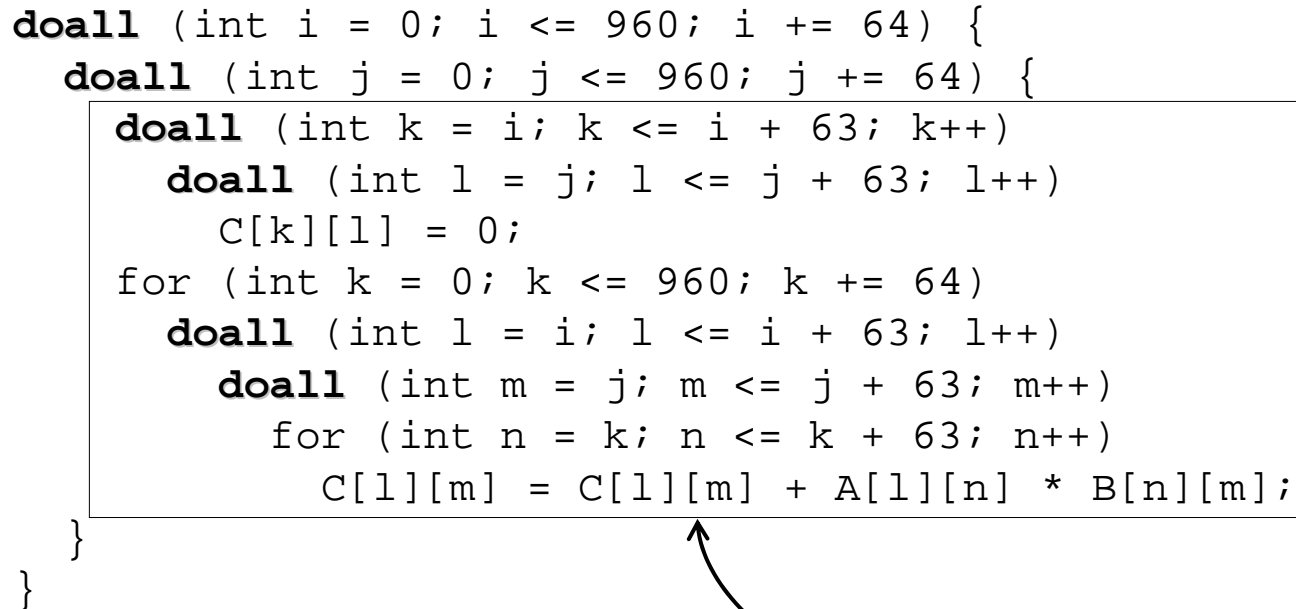
---

```
doall (int i = 0; i <= 1023; i++) {  
    doall (int j = 0; j <= 1023; j++) {  
        C[i][j] = 0;  
        for (int k = 0; k <= 1023; k++) {  
            C[i][j] = C[i][j] + A[i][k] * B[k][j];  
        }  
    }  
}
```

# Tiling

---

```
doall (int i = 0; i <= 960; i += 64) {  
  doall (int j = 0; j <= 960; j += 64) {  
    doall (int k = i; k <= i + 63; k++)  
      doall (int l = j; l <= j + 63; l++)  
        C[k][l] = 0;  
    for (int k = 0; k <= 960; k += 64)  
      doall (int l = i; l <= i + 63; l++)  
        doall (int m = j; m <= j + 63; m++)  
          for (int n = k; n <= k + 63; n++)  
            C[l][m] = C[l][m] + A[l][n] * B[n][m];  
  }  
}
```



**A logical tile which can fit  
into local memory**

- **256K memory on each SPU**
- **tile size = 64x64**
- **tile size takes into account of SIMD restrictions**

# Processor Placement

```
// The processor grid on CELL is one dimensional.
// PROC0 is a parameter which stands for the processor number.
// It ranges from 0 to 7.
doall (i = 0; i <= 1; i++) {
    doall (j = 0; j <= 15; j++) {
        doall (k = 512 * i + 64 * PROC0;
            k <= 512 * i + 64 * PROC0 + 63; k++)
            doall (l = 64 * j; l <= 64 * j + 63; l++)
                C[k][l] = 0;
        for (k = 0; k <= 15; k++)
            doall (l = 512 * i + 64 * PROC0;
                l <= 512 * i + 64 * PROC0 + 63; l++)
                doall (int m = 64 * j; m <= 64 * j + 63; m++)
                    for (int n = 64 * k; n <= 64 * k + 63; n++)
                        C[l][m] = C[l][m] + A[l][n] * B[n][m];
    }
}
```

- Outermost i-loop distributed onto 8 SPUs on the CELL
- Each processor executes the above code

# After Local Memory Compaction/DMA generation

```
float A_l_buf1[64][64], A_l_buf2[64][64];  
float B_l_buf1[64][64], B_l_buf2[64][64];  
float C_l_buf1[64][64], C_l_buf2[64][64];  
float (* A_l_v1)[64] = A_l_buf1;  
float (* A_l_v2)[64] = A_l_buf2;  
float (* B_l_v1)[64] = B_l_buf1;  
float (* B_l_v2)[64] = B_l_buf2;  
float (* C_l_v1)[64] = C_l_buf1;  
float (* C_l_v2)[64] = C_l_buf2;
```

- Local buffers and pointers to these buffers
- Total memory  $4*6*64*64=98304<256K$

```
doall (i = 0; i <= 1; i++) {  
  doall (j = 0; j <= 15; j++) {  
    if (j == 0) {  
      Initialization code  
    }  
    Pipelined 64x64 matrix multiply  
  }  
}
```

Executed sequentially,  
but parallelism available  
to be taken advantage of  
in other LLCs

General structure of  
mapped code

Logical tile

# Initialization Section

---

```
for (k = 0; k <= 16; k++) {
    if (k >= 1) { rotate C_l_v1 and C_l_v2; }
    if (k <= 15) {
        for (l = 0; l <= 63; l++)
            for (m = 0; m <= 63; m++)
                C_l_v1[l][m] = 0.0f;
    }
    if (k >= 1) CELL_dma_wait(1);
    if (k <= 15) {
        for (l = 0; l <= 63; l++)
            CELL_dma_put(
                &C_l_v1[l][0],
                &C[512 * i + l + 64 * PROC0][64 * k],
                64 * 4, 4, 4, 1, 1);
    }
}
```

# Pipelined 64x64 Matrix Multiply

```
for (k = -1; k <= 16; k++) { // 16 stages + 1 prologue and 1 epilogue
  if (k <= 15 && k >= 0) { // Block until the prefetched data is ready
    CELL_dma_wait(0);
    rotate C_l_v1 and C_l_v2, A_l_v1 and A_l_v2, B_l_v1 and B_l_v2;
  }
  if (k <= 14) {
    for (l = 0; l <= 63; l++) // Prefetch next block of A, B and C
      CELL_dma_get(&B[64*j+l][64+64*k], &B_l_v2[l][0], 64*4,4,4,1,0);
    for (l = 0; l <= 63; l++)
      CELL_dma_get(&A[512*i+l+64*PROC0][64*j], &A_l_v2[l][0], 64*4,4,4,1,0);
    for (l = 0; l <= 63; l++)
      CELL_dma_get(&C[512*i+l+64*PROC0][64+64*k], &C_l_v2[l][0], 64*4,4,4,1,0);
  }
  if (k <= 15 && k >= 0) { // 64x64 matrix multiply kernel
    doall (l = 0; l <= 63; l++)
      doall (m = 0; m <= 63; m++)
        for (n = 0; n <= 63; n++)
          C_l_v1[l][m] = C_l_v1[l][m] + B_l_v1[n][m] * A_l_v1[l][n];
  }
  if (k >= 1) CELL_dma_wait(1); // Block until the previous write completes
  if (k <= 15 && k >= 0) { // Initiate write back to C
    for (l = 0; l <= 63; l++)
      CELL_dma_put(&C_l_v1[l][0], &C[512*i+l+64*PROC0][64*k], 64*4,4,4,1,1);
  }
}
```

# Interface between PPU and SPU

```
union __context {
    struct {
        float (*A)[1024];
        float (*B)[1024];
        float (*C)[1024];
    } context;
    double padding[2];
}
```

PPU

SPU

```
union __context context;
extern spe_program_handle
    matmult1024_spu;
struct CELL_mapped_region* region;
context.context.A = A;
context.context.B = B;
context.context.C = C;
region = CELL_mapped_begin(0, 8, 0,
    &matmult1024_spu, &context,
    sizeof(context));
CELL_mapped_end(region);
```

```
int main(uint64_t id, uint64_t argp)
{
    union __context c;
    uint64_t t1;
    CELL_spu_init(id, argp, ...);
    CELL_dma_get((void *)_t1, &c,
        sizeof(c), 0, 0, 1, 0);
    CELL_dma_wait(0);
    __kernel(c.context.A,
        c.context.B,
        c.context.C);
    return 0;
}
```

# Deficiencies of Current Matrix Multiply Mapping

---

- **Array C should be resident in local memory between logical tiles**
- **The initialization section should be fused with the pipelined matrix multiply**
- **Loops around `CELL_dma_get/put` can be folded inside the DMA calls.**
- **`restrict` qualifiers missing from the pointer declarations**
- **For CELL**
  - **`#pragma disjoint` annotations missing from the pointer declarations**
  - **`__attribute__((aligned(...)))` annotations missing**
- **These are being addressed**

# QR Example

```
for (int i = 0; i <= 255; i++) {
    nrm[i] = 0;
    for (int j = i; j <= 255; j++) {
        nrm[i] = hypot(nrm[i], QR[j][i]);
    }
    if (nrm[i] != 0) {
        nrm[i] = sn(QR[i][i], nrm[i]);
        for (int j = i; j <= 255; j++) {
            QR[j][i] = QR[j][i] / nrm[i];
        }
        QR[i][i] = 1 + QR[i][i];
        for (int j = 1 + i; j <= 255; j++) {
            s[i][j] = 0;
            for (int k = i; k <= 255; k++) {
                s[i][j] = s[i][j] + QR[k][i] * QR[k][j];
            }
            s[i][j] = - s[i][j] / QR[i][i];
            for (int k = i; k <= 255; k++) {
                QR[k][j] = QR[k][j] + QR[k][i] * s[i][j];
            }
        }
    }
    Rdiag[i] = - nrm[i];
}
```

•256x256

•s and nrm  
manually  
array  
expanded

predicate elided

```
sn(x,y) =
x < 0 ? -y : y
```

# Parallelizing ...

```
// prologue code omitted
for (int i = 0; i <= 254; i++) {
    for (int j = i; j <= 255; j++)
        nrm[i] = hypot(nrm[i], QR[j][i]);
    nrm[i] = sn(QR[i][i], nrm[i]);
    doall (int j = i; j <= 255; j++)
        QR[j][i] = QR[j][i] / nrm[i];
    QR[i][i] = 1 + QR[i][i];
    doall (int j = i + 1; j <= 255; j++)
        for (int k = i; k <= 255; k++)
            s[i][j] = s[i][j] + QR[k][i] * QR[k][j];
    doall (int j = i + 1; j <= 255; j++)
        s[i][j] = - s[i][j] / QR[i][i];
    doall (int j = i + 1; j <= 255; j++)
        doall (int k = i; k <= 255; k++)
            QR[k][j] = QR[k][j] + QR[k][i] * s[i][j];
}
// epilogue code omitted
```

sequential

should be fused

# After Tiling


```
for (i = 0; i <= 255; i++) {
  for (j = i; j <= 255; j++)
    nrm[i] = hypot(nrm[i], QR[j][i]);
  nrm[i] = sn(QR[i][i], nrm[i]);
  doall (j = lo2 + gap3; j <= 224; j += 32
    doall (k = max(i, j); k <= j + 31; k++)
      QR[k][i] = QR[k][i] / nrm[i];
  QR[i][i] = 1 + QR[i][i];
  doall (j = lo6 + gap7; j <= 240; j += 16)
    for (k = lo4 + gap5; k <= 224; k += 32)
      doall (l = max(i + 1, j); l <= j + 15; l++)
        for (m = max(i, k); m <= k + 31; m++)
          s[i][l] = s[i][l] + QR[m][i] * QR[m][l];
  doall (j = lo8 + gap9; j <= 240; j += 16)
    doall (k = max(i + 1, j); k <= j + 15; k++)
      s[i][k] = - s[i][k] / QR[i][i];
  doall (j = lo12 + gap13; j <= 240; j += 16)
    doall (k = lo10 + gap11; k <= 224; k += 32)
      doall (l = max(i + 1, j); l <= j + 15; l++)
        doall (int m = max(i, k); m <= k + 31; m++)
          QR[m][l] = QR[m][l] + QR[m][i] * s[i][l];
}
```

- tile sizes 32x32, 16x16, 16x32
- prologue and epilogue omitted
- index computation code omitted...

# After Processor Placement

```
// PROC0 ranges from 0 to 7
for (i = 0; i <= 255; i++) {
  if (PROC0 == 0) {
    for (j = 0; j <= 255; j++) {
      if ( - i + j >= 0) nrm[i] = hypot(nrm[i], QR[j][i]);
      if (j == 0) {
        nrm[i] = sn(QR[i][i], nrm[i]);
        QR[i][i] = 1 + QR[i][i];
      }
    }
  }
  if ( - i + 32 * PROC0 >= -31)
    doall (j = max(i, 32 * PROC0); j <= 32 * PROC0 + 31; j++)
      QR[j][i] = QR[j][i] / nrm[i];
  doall (j = max(0, ceilDiv(i - 16 * PROC0 + -14, 128)); j <= 1; j++)
    for (k = max(0, ceilDiv(i + -31, 32)); k <= 7; k++)
      doall (l = max(128*j+16* PROC0, i+1); l <= 128*j+16*PROC0+15; l++)
        for (int m = max(32 * k, i); m <= 32 * k + 31; m++)
          s[i][l] = s[i][l] + QR[m][i] * QR[m][l];
  doall (j = max(0, ceilDiv(i - 16 * PROC0 + -14, 128)); j <= 1; j++)
    doall (k = max(128*j+16*PROC0, i+1); k <= 128*j+16*PROC0+15; k++)
      s[i][k] = - s[i][k] / QR[i][i];
  doall (j = max(0, ceilDiv(i - 16 * PROC0 + -14, 128)); j <= 1; j++)
    doall (k = max(0, ceilDiv(i + -31, 32)); k <= 7; k++) {
      doall (l = max(128*j+16*PROC0, i+1); l <= 128*j+16*PROC0 + 15; l++)
        doall (m = max(32 * k, i); m <= 32 * k + 31; m++)
          QR[m][l] = QR[m][l] + QR[m][i] * s[i][l];
    }
}
```

Sequential code  
mapped onto  
processor 0



# Local Memory Compaction

```
for (i = 0; i <= 255; i++) {  
    ... // other sections omitted  
    doall (j = max(0, ceilDiv(i - 16 * PROC0 + -14, 128));  
          j <= 1; j++)  
        doall (k = max(0, ceilDiv(i + -31, 32)); k <= 7; k++) {  
            doall (l = max(128*j+16*PROC0, i+1);  
                  l <= 128*j+16*PROC0 + 15; l++)  
                doall (m = max(32 * k, i); m <= 32 * k + 31; m++)  
                    QR[m][l] = QR[m][l] + QR[m][i] * s[i][l];  
        }  
}
```

A logical computation tile

- QR[m][l] and QR[m][i] are disjoint references
- QR[m][l] allocated to QR\_l\_8[32][16]
- QR[m][i] allocated to QR\_l\_7[32]
- s[i][l] allocated to s\_l\_3[16]

# Local Memory Compaction/DMA, Synchronization Generation

```
doall (j = max(0, ceilDiv(i + -126, 128)); j <= 1; j++) {
  doall (k = ...) {
    doall (l = ...) {
      if ( - i + 32 * k >= -31 && k <= 7 && k >= 0) {
        CELL_barrier(8);
        if (l - PROC0 == 0) {
          CELL_dma_wait(0);
          rotate pointers to QR_l_7, QR_l_8, s_l_3
        }
      }
    }
    if (l - PROC0 == 0) {
      if (k <= 6) {
        prefetch s to s_l_3, QR to QR_l_8 and QR_l_7
      }
      if ( - i + 32 * k >= -31 && k <= 7 && k >= 0) {
        doall (m = max(128*j+16*PROC0, i + 1); m <= 128*j+ 16 * PROC0 + 15; m++) {
          doall (int n = max(32 * k, i); n <= 32 * k + 31; n++) {
            QR_l_8[-32 * k + n][-128 * j + m -16 * PROC0] +=
              s_l_3[-128 * j + m -16 * PROC0] * QR_l_7[-32 * k + n];
          }
        }
      }
    }
    if ( - i + 32 * k >= 1 && k >= 1) CELL_dma_wait(1);
    if ( - i + 32 * k >= -31 && k <= 7 && k >= 0) {
      initialize write back of QR_l_8 to QR
    }
  }
}
```

# R-Stream is emerging from research stage

---

- **Plenty of caveats**
  - We know the problems are and they are being addressed
- **Soon: Substrate for hardware vendors trying to get libraries up rapidly and porting through time**
- **Follow on projects for NRL/MDA, OSD, ...**
- **But very powerful, doing opts unique and unprecedented**
- **We're interested in users using it**
  - *Summer internships for graduate students 2007*
  - *People logging in remotely*
  - *Shipping binaries*
  - *Mostly limited by our ability to support*
- **Substrate for a real tool for next generation multi-core research**
  - *Lots and lots and lots of interesting experiments enabled by the tool*

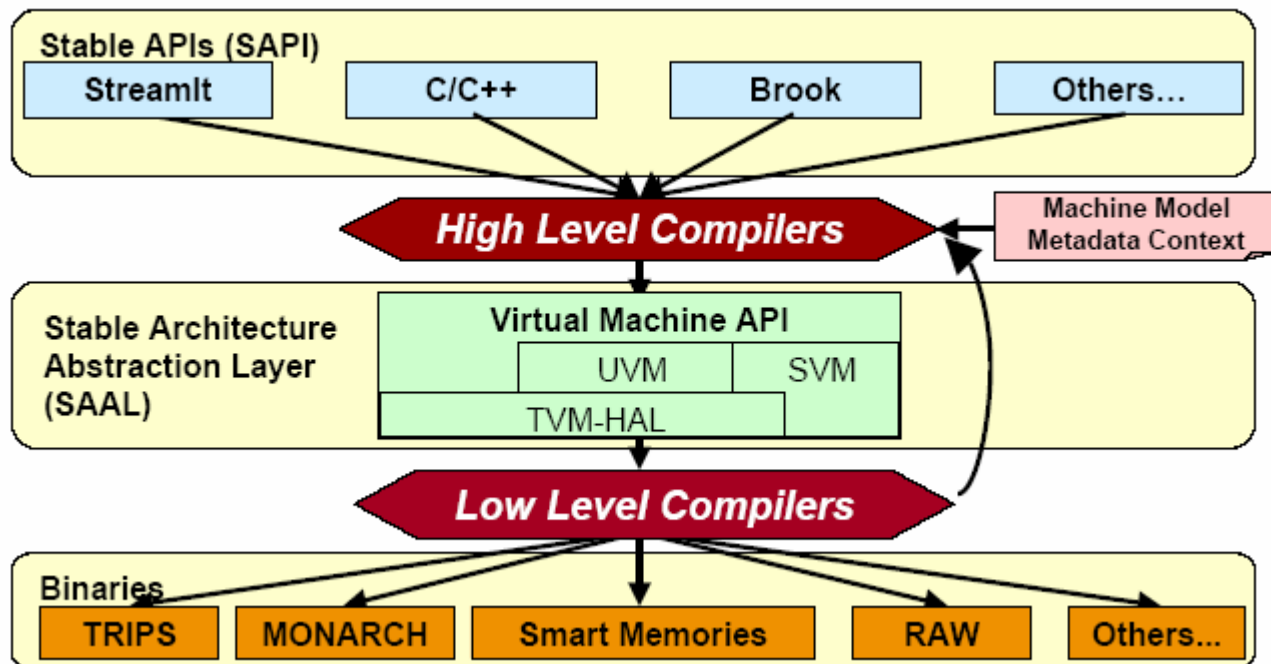
# Some Enabled Forward Research Projects

---

- Mapping, mapping, mapping
- Interaction with algebraic/domain-specific transformations
- Dependence precision vs. performance
- Non-affine transformations
- Fault tolerance (compiler-based checkpointing)
- Verification Projects
- Dynamic mapping capability (e.g., Morphing)
- Scalability of mapper
- Interaction of Phases
- New Source Language
- *Virtual Machines*
- *Machine Model*
- *LLC interface: dependence information and feedback*
- Heterogeneous Compilation
- User interaction: feedback, guidance, debugging
- New computational hardware architectures/features

# That interface: Morphware and SAAL

- As part of the PCA program, we worked on the “Morphware API” See [www.morphware.org](http://www.morphware.org) for more information, draft specifications.



# Comments on SAAL

---

- **An extraordinarily visionary proposition**
  - Credit to Mark Richards and Dan Campbell of GTRI
  - Both in terms of objectives:
    - Address the problems of SW engineering for diverse heterogeneous architectures
    - Attempt to deal with **MORPHING**
  - And, in terms of the structure:
    - A structure based on a centralized controller dispatching work to streaming engines.
    - To heterogeneous CPUs
    - With facilities for a machine model

# What Morphware Built

---

- **Streaming virtual machine specification**
- **Research implementations of SVM and early version of R-Stream targets SVM**
- **Machine model specification, early implementations**
- **Lots of lessons learned:**
  - Need to design mapping tool in hand with the execution mechanisms
  - Practical challenges in getting a LLC to optimize machine generated code
  - Need much more care in defining the execution model abstractly, and separating this from syntactic considerations
  - Much (and I'd venture most) of the specification went unused, and even hindered, high level mapping
- **We're ready to iterate and improve**

# In the meantime

---

- **Emerging standards that are working at what seems to be the same level**
  - E.g., for Cell, SPURS, Mercury MCF, other industrial, and every academic group rolling their own
  - For one microprocessor – no agreement on the standard!
  - Commercially significant “scrums” to determine what these APIs should look like on real machines
  - Commercial folks don’t have to bother with consensus
- **Here’s why Morphware is visionary – working to develop an open standard**
  - Think RISC vs. CISC, Windows vs. Unix, VLIW vs. Superscalar, C vs. FORTRAN
  - Analogy to the old adage that you have to develop the compiler and the architecture in-hand. The design of this API and software mapping methodologies should be done together.
  - Let’s cut to the chase and have an open standard
  - But we can and should look carefully at the commercial versions of this

# What it should have (1/3)

---

- **Clear execution model**
- **Small amount of memory / node – maybe 4K sized blocks**
  - Buy into the notion (Patterson) that caches are obsolete
  - Mechanisms for synthesizing shared memory from message passing
- **Can't have full OS, or even an MPI or OpenFabrics stack per node**
  - Not news to the supercomputing folks
- **Communication**
  - Nearest neighbor as well as dynamically point to point
  - Streams, scatter/gather
  - Other bulk transformations – corner turn, collective operations
  - Polyhedral and affine operations
  - Parallel prefix
  - Implementation questions: flow control, buffering
  - Intra-chip and inter-chip messaging

# What it should have (2/3)

---

- **Synchronization**
  - Producer-consumer (streams)
  - Barriers
  - Other forms of indicating dependence between tasks
  - Docking/undocking
  - Transactions?
- **Memory Management**
  - Explicit, but not too explicit, management of memory
  - For both data and instructions
- **Work initiation**
  - Active messages
  - Code management

# What it should have (3/3)

---

- **Other**
  - Debugging
  - Isolation and protection
  - Fault detection and isolation
  - Real time

# Designing for the Mapper

---

- **Clear semantics, clear execution model absolutely essential**
- **Don't have to make it too low level**
  - If it could take abstractions in the form of constraint-specified polyhedra, we'd be very happy.
  - Don't make us manage everything
  - But we can manage local memories and generate bulk communication
  - Don't have to bind everything
- **Make it concise (RISC philosophy)**
  - That's good for low memory constraints
- **Definition of the API should be similar to what a mapping is and can be**
  - E.g., our  $\theta$  from the polyhedral mapper

# What about the machine model?

---

- **That's a subject for another entire presentation**
  - Needs to be tied deeply to the models used that define “optimal” in the mapper
  - Gets interesting when we want to talk about “Morphing” - we want to be able to effect morphing changes by writing to the model