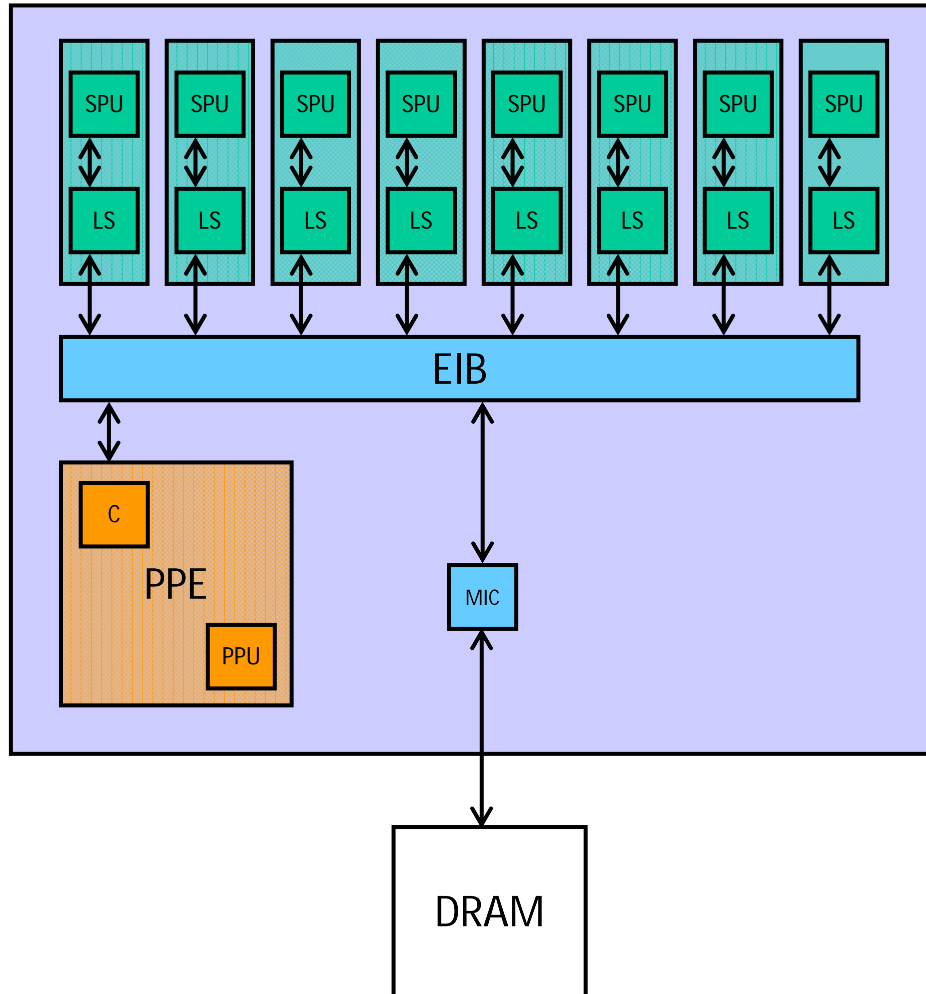
R-Stream™: A Framework for Automatic High-Level Optimization of Applications to the Cell/B.E. Microprocessor: Status and Challenges

Richard A. Lethin

Reservoir Labs, Inc.
New York, NY Portland, OR Columbia, MD
(212) 780-0527

The Challenge of Optimizing to Cell B.E.



High Level Mapping

- Partitioning between PPU/SPU
- Parallelization
- Scheduling
- Memory distribution & management
- Communication generation
- Synchronization
- Locality of reference

Low Level Mapping

- Instruction selection
- Instruction scheduling
- Register allocation
- SIMDization

Libraries are good, still…

The plusses:

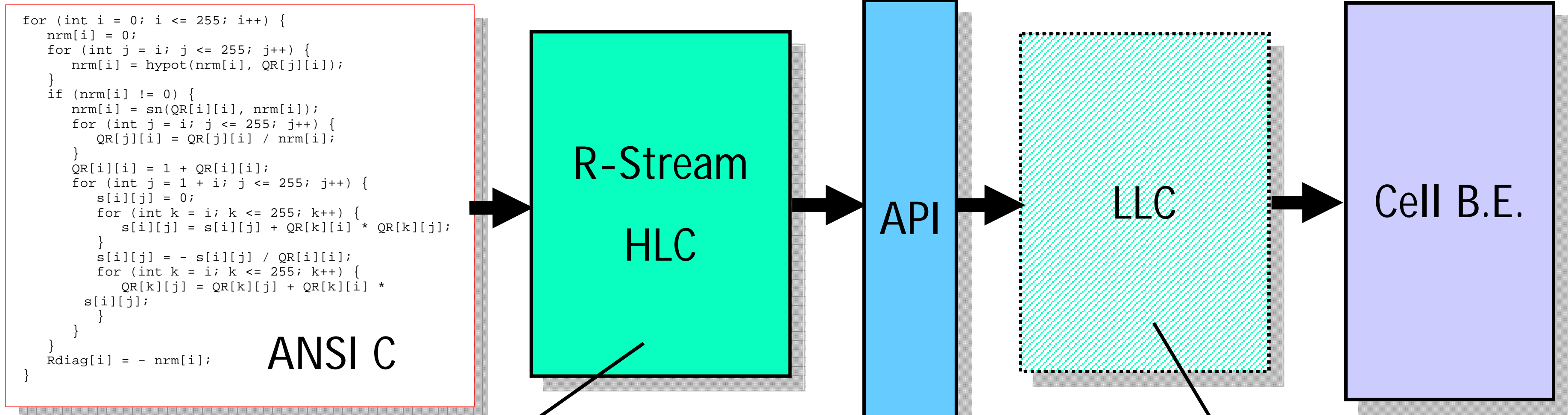
- Someone else does the work
- Just have to write to their abstraction
- People getting good results now
- Existing libraries for physics, image processing, linear algebra…

Still,

- What if the computation you want is not in the library?
- Who writes the library, and how?
- Non-standard or proprietary APIs
- Licensing costs – per core, per chip?
- Does the best performance occur when you break the abstraction and optimize across library calls?

Is hope justified for a tool that can automatically map code to Cell B.E.?

Compilation Flow



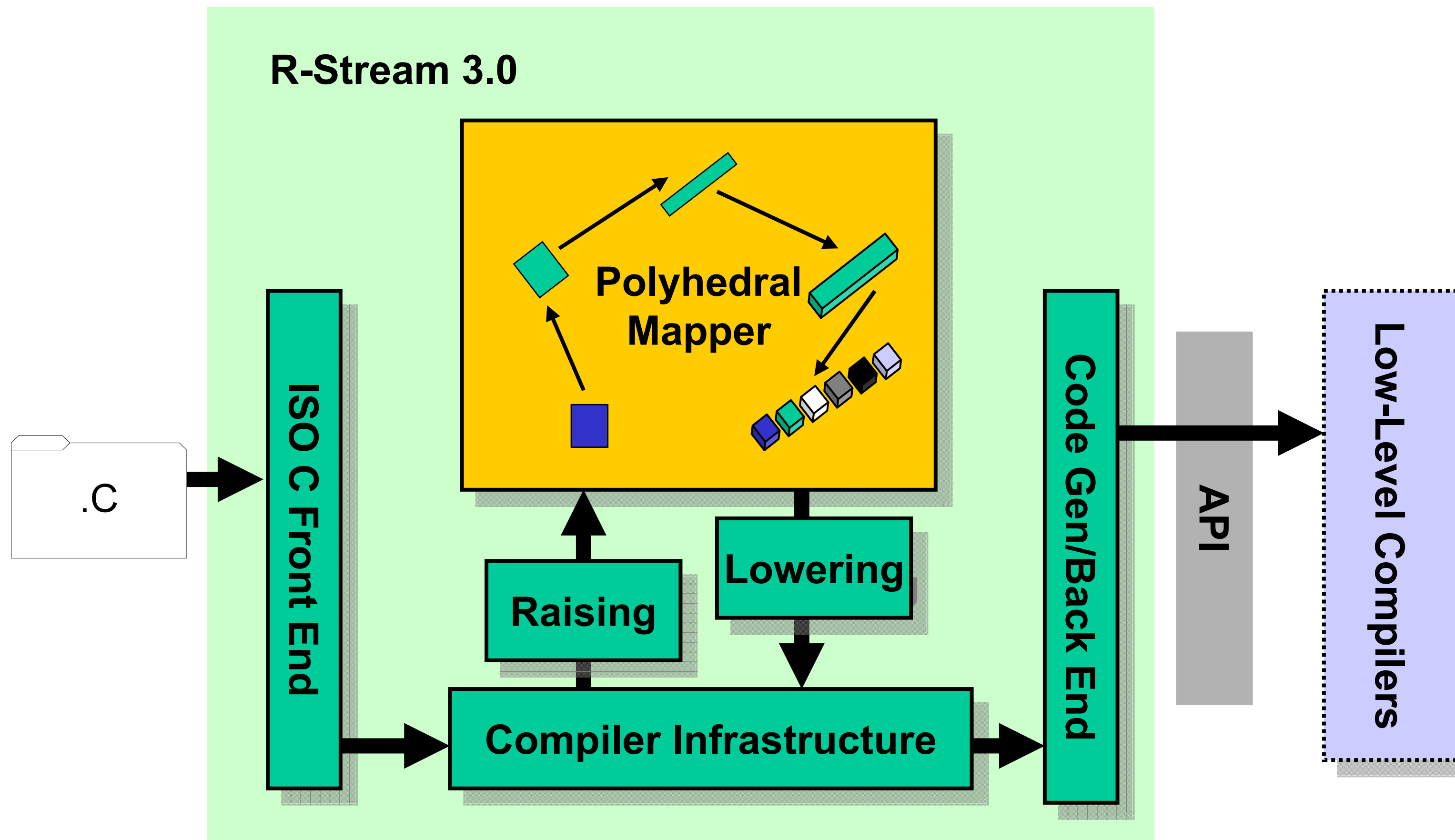
Partitioning between PPU and SPU
Parallelization across streaming elements
Memory layout and placement optimization
"Tiling" choosing correct size and organization of tasks
Improvements to locality of reference – producer consumer
Management of communication – intra and inter chip
Management of distributed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

Instruction selection
Register allocation
SIMDization
Instruction scheduling

Minimal API between HLC and LLC

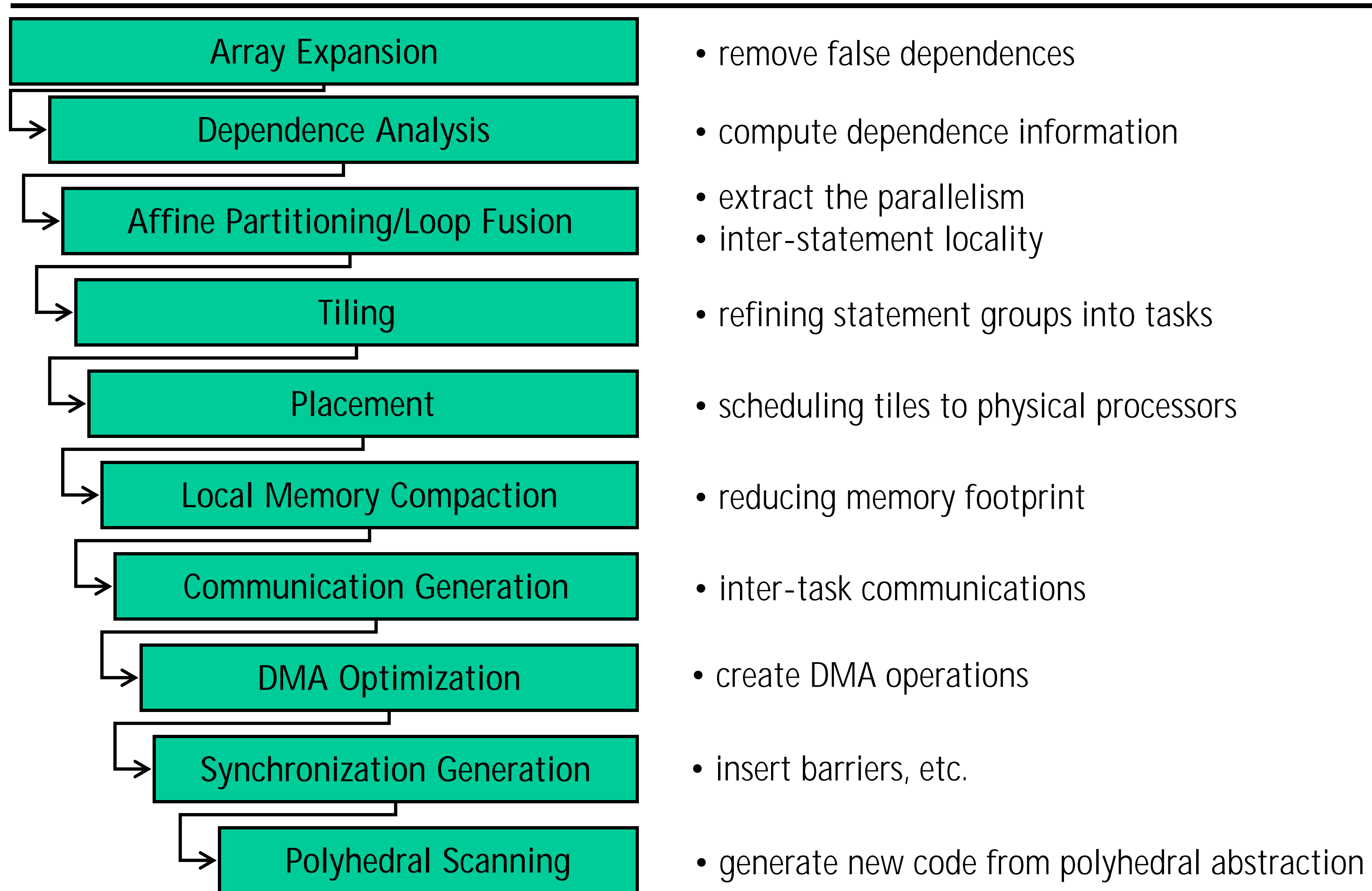
- "C language API"
- HLC Backend generates two separate programs, one for SPU and one for PPU
- Execution model: Most of application on PPU, SPU's operating "long" running tasks, initiating their own DMA and synchronization.
- Asynchronous DMA: `CELL_dma_get`, `CELL_dma_put`, `CELL_dma_wait`
- Process control: `CELL_mapped_begin`, `CELL_mapped_end`
- Memory management: C library
- Synchronization: `CELL_barrier(int id)`

Under the Hood



Details are the subject of many hours of presentations, but I'm happy to talk about it offline today… or please visit us in New York

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];
}
```

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

Iteration spaces as constraints (polytopes)

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

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

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

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

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

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 $D = \{x \in \mathbf{Z}^n \mid \mathbf{A}x + \mathbf{b} \geq 0\}$
  1. an iteration space
  2. a set of array references $A[f(x)]$
  3. a schedule (space/time mapping) **θ**, which determines when and where a statement instance is executed
- Combining transformations:
  - Loop transformations → modifies **θ** (time component)
  - Processor placement → modifies **θ** (space component)
  - Data transformations → modifies $A[f(x)]$
- 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 = conv(V)+cone(R)+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 Properties

- Model everything as polyhedra, linear constraints in N dimensions
- Can represent task, pipeline, data parallelism as affine schedules
- Can represent affine data layout transformations
- Do both computation mapping and data mapping 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
- Communication generation and DMA optimizations

Matrix Multiply Example

```
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
- Each SPU 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

```
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);
```

SPU

```
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;
}
```

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

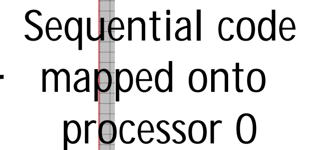
After Tiling

```
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
       }
     }
  }
```

Local memory compaction (Example)

Tiled inner loop from LU

```
float A[256][256];
doall (l = 128 * j + 16 * P; l <= min(-i+254, 128*j + 16*P + 15); l++)
  doall (m = 16 * k; m <= min(-i+254, 16*k + 15); m++)
    A[l + i + m][l + i + l] -= A[l + i + m][i] * A[i][l + i + l];
```

- Disjoint references can be allocated to different local arrays
- Reindexing and allocation can be done automatically

```
float A_2[16][16]; // a triangular subregion of A
float A_3[16];     // a column of A
float A_4[16];     // a row of A

// DMA code omitted
doall (l = 0; l <= min(15,-i-128 *j -16 *P+ 254); l++)
  doall (m = 0; m <= min(-i-16 * k + 254, 15); m++)
    A_2[m][l] -= A_3[m] * A_4[l];
// DMA code omitted
```

Scope of Application

- Generalities: Limitations of:
  - C language
  - Polyhedral model
  - Scalability of optimization algorithms and underlying mathematical solvers
  - Our implementation
  - Raising/lowering modules in R-Stream
  - …limit us to kernels within the immediate scope of model.
  - …taking off-the-shelf code, and raising, optimizing it, is still a challenge
- Currently working through performance issues associated with handing code to an LLC to optimize
- Nevertheless, we have a very powerful substrate, underlying model, and our goal is to provide this as a commercial tool for programming Cell B.E.
  - *It's a question of when, not if…*

How (and when) can I get it?

- Now: inviting select users to log in to try it out on our systems…
- *Will be able to distribute "Alpha" versions in ~ 2 months*
- Goal: to have this usable as a flow alongside an IBM 3.0 SDK release (Q3/07)

- (As an aside, while our plan is to provide this as a commercial tool, it ALSO opens up enormous number of research opportunities in automatic optimization
  - We *will* be interested in working with academic, government, and industrial research groups on optimization on forward research projects)

Thanks

- To the commercial interest and support by the members of STI
- The extremely dedicated and talented members of the Reservoir compiler team
- The members of the Morphware Forum
- DARPA/AFRL for providing funding for this work (F03602-03-C-0033)
- Other government agencies and components for their interest and support