Automatic High Level Compilation to the Cell/B.E. with the Polyhedral Approach

Allen Leung, Benoît Meister, Nicolas Vasilache, David Wohlford, Richard Lethin

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

- Programming the CELL Processor
- The R-Stream Compiler
- The Programming Model
- CELL Runtime Layer
- The Polyhedral Model
- Mapping Tasks and Algorithms
- Current Weaknesses and Future Improvements
- Conclusion
The Challenge of Optimizing to Cell B.E.

High Level Mapping
- Partitioning between PPE/SPE
- Parallelization
- Scheduling
- Memory distribution & management
- Communication generation
- Synchronization
- Locality of reference

Low Level Mapping
- Instruction selection
- Instruction scheduling
- Register allocation
- SIMDization
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 Polymorphous Computing Architectures (PCA)
- Based on the polyhedral model:
  - Less “magic”

Target Machine Abstractions

Programming Challenges

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

Polymorphous Computer Architectures
- Smart Memories
- Monarch
- TRIPS
- RAW
- Cell
- GPU
- FPGA

COTS Streaming Architectures
- reservoir labs®
Current R-Stream Infrastructure

- C Front-end
  - conditional constant propagation
  - algebraic simplification
  - global value numbering
  - strength reduction
  - loop invariant hoisting
  - unrolling
  - global reassociation
  - inlining
  - pointer analysis
  - etc …

- Sprig: SSA-based optimizing compiler infrastructure
  - identify mappable regions
  - identify statement granularity
  - determine loop structure
  - identify arrays and affine indexing expressions

- Mapper
  - polyhedral representation

- Machine Models
  - x86
  - PowerPC
  - TRIPS
  - Cell SPU
  - AMD64

- Lowering
  - space/time mapping

- Target API
  - resynthesize loop nests
  - perform target specific transformations

- Raising
  - global reassociation

- Mapper Machine Model
  - TRIPS
  - CELL
  - FPGA

- foo.c
  - C Front-end

- foo_ppu.c
  - OpenMP backend
  - FPGA backend

- foo_spu.c
  - CELL backend
  - OpenMP backend
  - FPGA backend

- foo.c

- Target API
  - GPU
  - CELL
  - OpenMP
  - FPGA
The Programming Model

- Sequential C
  - Auto-parallelization and mapping by the compiler
  - Output in mapped C form (programs in PPE and SPE form)
- Caveats:
  - The mapper can only handle static affine control programs: affine indexing functions, affine loop bounds, parameters allowed
  - But, data-dependent predicates are allowed
  - Currently, the mapper requires the user to specify which functions she wants to map via a pragma

```c
#pragma rstream map
void matmult(float A[N][N], float B[N][N], float C[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i++) {
    C[i][j] = 0;
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        C[i][j] += A[i][k] * B[k][j];
  }
}
```
CELL Runtime

- A thin layer on top of libspe2.h
- SPE side:
  - Async DMA: `CELL_dma_get`, `CELL_dma_put`, `CELL_dma_wait`
    - Alignment and size restrictions removed in most generate version
    - Specialized variants when special size and alignment invariants are satisfied
    - Translate strided DMA → DMA list
    - Implemented in terms of `spu_mfcdma64`, `spu_writech`, `spu_mfcstat`, etc.
  - Synchronization barrier: `CELL_barrier`
    - Implemented in terms of `atomic_ea_t` and `completion_ea_t`
- PPE side:
  - Process control, SPE program loading: `CELL_mapped_begin`, `CELL_mapped_end`
    - Implemented in terms of `spe_program_load()` and `pthreads`
- Execution model preferences:
  - Data parallel
  - Bulk communication (take advantage of DMA)
  - Coarse-grained synchronization
  - SPEs can coordinate parallel execution
/* Mapped Matrix Multiply on SPEs */

```c
// outer loops omitted
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);
        swap C_l_v1 and C_l_v2, A_l_v1 and A_l_v2, B_l_v1 and B_l_v2;
    }
    if (k <= 14) {
        // Prefetch next block of A, B and C
        CELL_dma_get(&B[64*j][64+64*k], &B_l_v2[0][0], 64*4,1024*4,64*4,64,0);
        CELL_dma_get(&A[512*i+64*PROC0][64*j], &A_l_v2[0][0],
                     64*4,1024*4,64*4,64,0);
        CELL_dma_get(&C[512*i+64*PROC0][64+64*k],&C_l_v2[0][0],64*4,
                     1024*4,64*4,64,0);
    }
    if (k <= 15 && k >= 0) { // 64x64 matrix multiply kernel
        for(l = 0; l <= 63; l++)
            for(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
        CELL_dma_put(&C_l_v1[0][0],&C[512*i+64*PROC0][64*k],
                     64*4,64*4,1024*4,64,1);
    }
}
// outer loops omitted
```
Interface between PPE and SPEs

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

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;
}
What’s Mapping?

• Scalar optimizations
• **Raising**: SSA $\rightarrow$ Polyhedral Form
• **Array expansion** and **dependence analysis**
• **Affine Scheduling**: extract coarse-grained and fine-grained synchronization-free and pipelined parallelism + locality optimization
• **Tiling**: group computation into coarse-grained “tasks” with bulk communication
• **Processor assignment**: assign tasks to processors in grid (1-D in CELL)
• **Local memory compaction**: layout memory for each tasks and generate communication; double-buffering to overlap communication and computation
• **DMA Generation**: communication $\rightarrow$ target specific DMA operators
• Optionally, **loop unrolling, unroll and jam, array contraction**, etc.
• **Polyhedral Scanning** and **Lowering**: polyhedral Form $\rightarrow$ AST $\rightarrow$ SSA
• Scalar optimizations
Scalar Optimizations

- Apply classical scalar optimizations in Static Single Assignment (SSA) Form
  - Clean up input code
  - Make other analyses more accurate
- Conditional Constant Propagation
- Global Value Numbering
- Global Code Motion
- Induction Variable Elimination/Strength Reduction
- Global Reassociation
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];
}

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 & -4 \\
8 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2
\end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \]

\[
\begin{bmatrix}
i & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \]

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

\[
\{(i, j) | \exists k. 2 \leq i \leq M, 0 \leq j \leq N, j = 2k\}
\]

\[
\{(i, j) | 2 \leq i \leq M, i \leq j \leq N\}
\]
Raising: C Code $\rightarrow$ Polyhedral Form

- Pointer analysis
- Loop detection
- Recurrence detection
  - Affine, modulo and exponential recurrence can be detected
  - Normalizes C’s pointer arithmetic idioms into array indexing
- If-conversion for programs with if/then/else
- Regroup fine-grained operations into “statements”
Affine Scheduling

**Affine scheduling**: given statements $S_1, \ldots, S_n$ and
dependence relations $R_{ij}$,
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 if $\Theta_{S_i}(x) \geq \Theta_{S_j}(y)$, $(x, y) \in R_{ij}$ 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 algorithm tries to maximize the amount of coarse-grained (synchronization-free and pipelined) parallelism while trying to maximize locality and minimize communication.
Tiling

- Affine scheduling determines the “tile shape”
- Tiling phase determines the tile size(s)
- Statement-wise tiling
- Tile size determines
  - Communication volume
  - Memory footprint
- Limited by
  - SPE’s local memory
  - SIMD/DMA alignment restrictions
- Algorithm:
  - Ehrhart polynomials are used compute functions from tile sizes to cost metric
  - Similar to cache-miss equations
  - Genetic algorithm to guide the search for good tile sizes
Placement

- Mapped inter-tile coordinates into processors
- Local communications can be optimized
Local Memory Compaction

\[
\min \prod_{i=1}^{d} S_i
\]

\[\forall y. L(y) \leq U(x) < L(y) + S, \quad x \in D(y)\]

\(U : \mathbb{Z}^d \rightarrow \mathbb{Z}^d\) is a unimodular reindexing function

\(L : \mathbb{Z}^p \rightarrow \mathbb{Z}^d\) is an integral affine function

which represents the lower bound of the indices

\(S : \mathbb{Z}^d\) represents the dimensions of the local array

\(D(y)\) is a parametric data footprint set within a tile
DMA Generation

- Starts from a data transfer set: a set of data accessed by a convex set of iterations

- Scan data along dimensions for which array elements are contiguous
- Collapse contiguous array elements into DMA operations

remote memory \[\rightarrow\] affine relation \[\rightarrow\] local memory
for (l = 512 * i + 64 * PROC0; l <= 512 * i + 64 * PROC0 + 63; l++)
  for (m = 64 * k; m <= 64 * k + 63; m++)
    for (n = 64 * j; n <= 64 * j + 63; n++)
      C_l[-512 * i + l - 64 * PROC0][-64 * k + m] +=
      B_l[-64 * j + n][-64 * k + m] *
      A_l[-512 * i + l - 64 * PROC0][-64 * j + n];

• Both loops are equally “easy” for our mapper
• But gcc and IBM’s XLC cannot SIMDize the above loop
• Find “equivalent schedules” which are simpler

for (l = 0; l <= 63; l++)
  for (m = 0; m <= 63; m++)
    for (n = 0; n <= 63; n++)
      C_l[l][m] += B_l[n][m] * A_l[l][n];
Polyhedral Scanning

- Polyhedral representation → loop nests
- Some improvements over state-of-the-art, CLooG
- Constraints tightening in every step to remove redundant loop bounds
- Better detection of loop strides
- Hoisting loop invariant predicates
- Mechanisms for controlling code explosion
  - Specialize only in innermost loop nests
  - Generate predicates in outer loops for control execution
- Reverse if-conversion
High Level Syntax Reconstruction

- CFG/SSA → CFG+quads
- Quads → expressions
- CFG → while, do-while, for-loops, if/else, switch, break, continue, goto, etc.
- Pointer expressions → array expressions
- Matching common C idioms:
  - \( i=i+1 \) → ++
  - \( *(A+2*i) \) → \( A[2*i] \)
- Insert target specific pragmas and type annotations (e.g.,
  \_
\_
declspec(...), restrict, \_
\_
attribute((...)), etc.)
- Generally speaking, each compiler requires different customization
to its output
  - E.g., XLC doesn’t SIMDize while loops with breaks, and some pointer
    arithmetic idioms
Future Extensions

- Automatically determine mapping region
- Larger scope of input loop nests
- Better support for CELL/XLC features
Non-Affine Extensions

- Geometric recurrences --- appears in some FFT implementations
- Non-linear indexing functions --- encoding dynamic multidimensional arrays as 1-D arrays in C89
- \( A[p[x]] \) – where \( p \) is an updatable permutation
- \( A[f(x) \% m] \) where \( f(x) \) is affine
- While-loops
- Pragmas support for
  - Associativity and commutativity
  - Data independence
SIMDization

- Currently XLC cannot SIMDize loops with complex bounds or complex array indexing functions
- Performance of compiler SIMDization still lags behind hand SIMDized code
- Performance gaps should narrow in future XLC and gcc releases

- Alternative: perform SIMDization in the polyhedral mapper
- Generate SPU intrinsics for the computationally intensive loops
- Advantage: the mapper has more accurate information about dependencies and alignment
Conclusion

- Automatic and effective mapping of high performance numerical code is feasible with state-of-the-art techniques

- The polyhedral model:
  - Powerful techniques of formulating and solving mapping problems for numerical kernels/applications
  - Organization principles for structuring a parallelizing compiler
Thanks

- To the commercial interest and support by the members of STI
- Kathryn O’Brien, Yuan Shao, Alexandre Eichenberger and others at IBM/Watson for providing valuable help for XLC
- DARPA/AFRL for providing funding for this work (F03602-03-C-0033, W31P4Q-07-C-0147, W9113M-07-C-0072, W9113M-08-C-0146 and W31P4Q-08-C-0319.)
- Other government agencies and components for their interest and support