Parallelism in Cell
- PPE has SIMD VMX instructions and is 2-way SMT
- 8 SPEs
  - Each SPE operates exclusively on 4-vectors
  - Odd/even instruction pipes
  - Odd for branching and memory / Even for compute
- Asynchronous DMAs
  - SWP of communication and computation
  - Memory-level parallelism

Locality in Cell
- PPE has L1 and L2 caches
  - L2 cache is 512KB
- SPEs have lots of registers and local store
  - 128 registers per SPU
  - 256KB 16 registers per SPU

Communication and Synchronization in Cell
- Element Interconnect Bus (EIB)
  - Carries all data between memory system and PPE/SPEs
  - Carries all data between SPE-SPE, SPE-PPE, and PPE-SPE
- PPE L2 cache is coherent with memory
- SPEs are not coherent (LS is not a cache)
- SPE DMA coherent with L2
- SPEs can DMA to/from one another
  - Memory map LS space into global namespace
  - Each SPE has local virtual memory translation to do this
  - No automatic coherency - explicit synchronization
- Synchronization through memory or mailboxes

Outline
- Cell mechanism review
- Cell programming challenges
- Sequoia
  - (up to mapping in this lecture)
- Other Cell programming tools
  - Next lecture
- Sequoia part courtesy Kayvon Fatahalian, Stanford
- All Cell related images and figures © Sony and IBM
- Cell Broadband Engine ™ Sony Corp.

Minimalistic HW structures improve performance and efficiency
Outline

• Cell mechanism review
• Cell programming challenges
• Sequoia
• Other Cell programming tools

Sequoia part courtesy Kayvon Fatahalian, Stanford

All Cell related images and figures © Sony and IBM
Cell Broadband Engine ™ Sony Corp.

Cell Software Challenges

• Separate code for PPE and SPEs
  - Explicit synchronization
• SPEs can only access memory through DMAs
  - DMA is synchronous, but prep instructions are part of SPE code
  - SW responsible for consistency and coherency
  - SW responsible for alignment, granularity, and bank conflicts
• SPEs must be programmed with SIMD
  - Alignment is up to SW
  - Lots of pipeline challenges left up to programmer/compiler
  - Deep pipeline with no branch predictor
  - 2-wide scalar pipeline needs static scheduling
  - LS shared by DMA, instruction fetch, and SIMD LDST
  - No memory protection on LS (stack can “eat” data or code)

Memory System Challenges

• No arbitrary global LD/ST from SPE
  - Everything must go through DMA
• Strict DMA alignment and granularity restrictions
  - All DMAs must be aligned to 16-bytes and request at least 16 bytes
  - All DMAs should be aligned to 128-bytes and request at least 128 bytes of contiguous memory
  - Failing to maintain alignment will cause a bus error!
• Banked DRAM structure
  - Consecutive DMA requests should span 2KB (8 banks)
• DMA commands issued from within SPE
  - Limited data access
  - Chain of state commands to prepare SPE and store in LS
  - Need to explicitly synchronize with DMA completion

SPE Programming Challenges

• Vector only
  - Scalars must be placed in LSW of vector and some instructions will take this into account
  - All casts and operations are aligned to 16 bytes (4 words)
  - Poor integer to float casting
• LS is only SRAM in SPE
  - Careful with growing stack
  - Sometimes need to force instruction prefetch to avoid stalls
  - Prioritized DMA first, then LS, and only then fetch
• No branch prediction (predict not-taken?)
  - Can have branch hint instructions inserted explicitly
• Some help from XLC compiler (and an occasional bug)

Separate Code for PPE and SPEs

• PPE should be doing top-level control and I/O only
  - Trying to compute on the PPE is problematic
  - In order 2-way SMT with deep pipeline
  - Busy enough just trying to coordinate between the SPEs
• SPEs are the parallel compute engines
  - PowerPC architecture, but really different ISA and requirements
  - Vector only, 2-way superscalar, ...
• Two processor types really have two different software systems
  - Synchronization can be tricky to get efficient
  - PPE should use mailbox
  - SPE should use L2 cache location
  - Why?
• Spawning threads on SPEs painfully slow

Overall

• Lots to deal with
  - Explicit communication and synchronization
  - Explicit locality
  - Explicit parallelism
  - Explicit pipeline scheduling
• Need some help from programming tools
Outline
- Cell mechanism review
- Cell programming challenges
- **Sequoia**
- Other Cell programming tools

Sequoia

Programming the Memory Hierarchy
Kayvon Fatahalian, Daniel Reiter Horn, Alex Asken, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Pat Hanrahan, Mattan Erez, Manman Ren
Stanford University

Avoiding latency stalls
- Exploit locality to minimize number of stalls
  - Example: Blocking / tiling

Avoiding latency stalls
1. Prefetch batch of data
2. Compute on data (avoiding stalls)
3. Initiate write of results

... Then compute on next batch (which should be loaded)
**Exploit locality**

- Compute > bandwidth, else execution stalls

```
| time | compute 1 | compute 2 | stall |
|      | Write output 0 | Read input 2 | stall |
|      | Read input 1 | Write output 1 | Read input 3 |
```

**Streaming**

Streaming involves structuring algorithms as collections of independent [locality cognizant] computations with well-defined working sets.

This structuring may be done at any scale.

- Keep temporaries in registers
- Cache/scratchpad blocking
- Message passing on a cluster
- Out-of-core algorithms

Efficient programs exhibit this structure at many scales.

**Locality in programming languages**

- Local (private) vs. global (remote) addresses
  - UPC, Titanium
- Domain distributions (map array elements to location)
  - HPF, UPC, ZPL
  - Adopted by DARPA HPCS: X10, Fortress, Chapel

Focus on communication between nodes
Ignore hierarchy within a node

**Locality in programming languages**

- Streams and kernels
  - Stream data off chip, Kernel data on chip.
  - StreamC/KernelC, Brook
  - GPU shading (Cg, HLSL)
  - CUDA

Architecture specific
Only represent two levels

**Hierarchy-aware models**

- Cache obliviousness (recursion)
- Space-limited procedures (Alpern et al.)

Programming methodologies, not programming environments
Sequoia’s goals

- Facilitate development of hierarchy-aware stream programs … … that remain portable across machines
- Provide constructs that can be implemented efficiently without requiring advanced compiler technology
  - Place computation and data in machine
  - Explicit parallelism and communication
  - Large bulk transfers

Hierarchical memory

- Abstract machines as trees of memories

Hierarchical memory in Sequoia

Similar to:
Parallel Memory Hierarchy Model (Alpern et al.)
Hierarchical memory

Dual Cell blade
Main memory

(No memory affinity modeled)

Hierarchical memory

Cluster of dual Cell blades
Aggregate cluster memory (virtual level)

Hierarchical memory

System with a GPU
Main memory
GPU memory

Blocked matrix multiplication

```c
void matmul_L1(int M, int N, int T,
    float* A, float* B, float* C)
{
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            for (int k=0; k<T; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Blocked matrix multiplication

```c
void matmul_L2(int M, int N, int T,
    float* A, float* B, float* C)
{
    Perform series of L1 matrix multiplications.
}
```

Blocked matrix multiplication

```c
void matmul(int M, int N, int T,
    float* A, float* B, float* C)
{
    Perform series of L2 matrix multiplications.
}
```
Sequoia tasks

- Special functions called tasks are the building blocks of Sequoia programs

```c
task matmul::leaf( in float A[M][T],
in float B[T][N],
inout float C[M][N] )
{
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            for (int k=0; k<T; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Read-only parameters M, N, T give sizes of multidimensional arrays when task is called.

Sequoia tasks

- Task arguments and temporaries define a working set
- Task working set resident at single location in abstract machine tree

```c
task matmul::inner( in float A[M][T],
in float B[T][N],
inout float C[M][N] )
{
    tunable int P, Q, R;
    mappar( int i=0 to M/P,
            int j=0 to N/R ) {
        mapseq( int k=0 to T/Q ) {
            matmul( A[P*i:P*(i+1);P]
                    B[Q*k:Q*(k+1);Q],
                    C[P*i:P*(i+1);P] );
        }
    }
}
```

Task hierarchies

```
task matmul::inner( in float A[M][T],
in float B[T][N],
inout float C[M][N] )
{
    tunable int P, Q, R;
    mappar( int i=0 to M/P,
            int j=0 to N/R ) {
        mapseq( int k=0 to T/Q ) {
            matmul( A[P*i:P*(i+1);P]
                    B[Q*k:Q*(k+1);Q],
                    C[P*i:P*(i+1);P] );
        }
    }
}
```

Variant call graph

Calling task: matmul::inner
Located at level X

Callee task: matmul::leaf
Located at level Y
Task hierarchies

```
Task matmul::inner(in float A[M][T],
in float N[R][N],
inout float C[M][N])
{
    tunable int P, Q, R;
    mappar(int i=0 to M/P, int j=0 to N/R) {
        mapseq(int k=0 to T/Q) {
            matmul(A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
                    B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
                    C[P*i:P*(i+1);P][R*j:R*(j+1);R]);
        }
    }
}
```

- Tasks express multiple levels of parallelism

Leaf variants

```
Task matmul::leaf(in float A[M][T],
in float B[T][N],
inout float C[M][N])
{
    for(int i=0; i<M; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<T; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

- Be practical: Can use platform-specific kernels

```
Task matmul::leaf_cblas(in float A[M][T],
in float B[T][N],
inout float C[M][N])
{
    cblas_sgemm(A, M, T, B, T, N, C, M, N);
}
```

Summary: Sequoia tasks

- Single abstraction for
  - Isolation / parallelism
  - Explicit communication / working sets
  - Expressing locality

- Sequoia programs describe hierarchies of tasks
  - Mapped onto memory hierarchy
  - Parameterized for portability