Programming the Cell Broadband Engine Processor

Robert Cooper

With thanks to:
Brian Bouzas, Luke Cico, Jon Greene, Maike Geng, Jamie Kenny, Michael Pepe, Myra Prelle
Outline

- Cell architecture and performance
- Mercury approach to programming Cell
- MultiCore Framework
- Final thoughts
Cell BE Processor Architecture

- Resembles distributed memory multiprocessor with explicit DMA over a fabric
Mercury Multi-DSP Board (1996)

Refer to the Mercury Computer Systems data sheet on RACEway Interlink modules for more information.

**One dual compute node daughtercard can be replaced with an I/O daughtercard; some compute node daughtercards are available with one compute node.**
Programming Cell: What’s Good and What’s Hard

**Good**

- No second guessing about cache replacement algorithm
- Very deterministic pipeline
- 128 registers mask pipeline latency very well

**SPE**

- Burden on software to get code and data into local store
- Local store is small compared to ring latency
- Branch prediction is manual and very restricted

**Ring and XDR**

- DMA has negligible impact on SPE local store bandwidth
- Generous ring bandwidth means topology is seldom an issue

**PPE**

- Standard Power® core
- 128 byte alignment necessary for best performance
- XDR bandwidth is a bottleneck
- Cell chips linked in coherent mode increases latency

**Hard**

- Performance is modest
How Much Faster Is Cell?

Relative performance of Cell and leading general purpose processors

• Performance relative to 1GHz Freescale 744x (i.e. Freescale = 1)

• In all cases, we are comparing Mercury optimized Cell algorithm implementations with the best available (Mercury or 3rd party) implementations on other processors

• Did not compare with dual core x86 processors
Goals for Programming Cell

• **Achieve high performance:**
  ▪ The only reason for choosing Cell

• **Ease of programming:**
  ▪ An important aspect of this is *programmer* portability

• **Code Portability**
  ▪ Important for large legacy code bases written in C/C++, Fortran
  ▪ And new code developed for Cell should be portable to current and anticipated multiprocessor architectures
Mercury Approach to Programming Cell

- **Very pragmatic**
  - Can’t wait for tools to mature
  - Develop our own tools when it makes sense

- **Emphasis on explicitly programming the architecture rather than trying to hide it**
  - When the tools are immature, this allows us to get maximum performance

- **Achieve ease-of-use and portability through function offload model**
  - Run legacy code on PPE
  - Offload compute intensive workload to SPEs
Mercury Elements of Cell Ecosystem

- **Maximum performance tools**
  - SPE local algorithm libraries
    - Mercury Scientific Algorithm Library (SAL) ported to SPE
  - Explicit multicore middleware
    - MultiCore Framework
  - Assembly language generation and preprocessing tools

- **Portability tools**
  - Portable algorithm libraries callable from PPE
    - PPE SAL runs on PPE vector unit
    - Multicore SAL executes in parallel on SPEs

- **Performance debugging tool**
  - Mercury Trace Analysis Tool (TATL) extended to work on SPEs

- **Complement other tools in ecosystem**
  - Compilers, languages, libraries, debuggers, simulators
MultiCore Framework

• An API for programming heterogeneous multicores that contain explicit non-cached memory hierarchies
• Provides an abstract view of the hardware oriented toward computation of multidimensional data sets
• First implementation is for the Cell BE processor
**MCF Abstractions**

- **Function offload model**
  - Worker Teams: Allocate tasks to SPEs
  - Plug-ins: Dynamically load and unload functions from within worker programs

- **Data movement**
  - Distribution Objects: Defining how n-dimensional data is organized in memory
  - Tile Channels: Move data between SPEs and main memory
  - Re-org Channels: Move data among SPEs
  - Multibuffering: Overlap data movement and computation

- **Miscellaneous**
  - Barrier and semaphore synchronization
  - DMA-friendly memory allocator
  - DMA convenience functions
  - Performance profiling
MCF Abstractions

• **Function offload model**
  - Worker Teams: Allocate tasks to SPEs
  - Plug-ins: Dynamically load and unload functions from within worker programs

• **Data movement**
  - Distribution Objects: Defining how n-dimensional data is organized in memory
  - Tile Channels: Move data between SPE and main memory
  - Re-org Channels: Move data among SPEs
  - Multibuffering: Overlap data movement and computation

• **Miscellaneous**
  - Barrier and semaphore synchronization
  - DMA-friendly memory allocator
  - DMA convenience functions
  - Performance profiling
Frame

One complete data set in main memory

- **Distribution Object parameters:**
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, including partition overlap
MCF Distribution Objects

One complete data set in main memory

- **Distribution Object parameters:**
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, including partition overlap
MCF Partition Assignment

- **Distribution Object parameters:**
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, including partition overlap
MCF Tile Channels

Partitions

Tile Channel

- SPE 0
- SPE 1
- SPE 2

- Distribution Object parameters:
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, including partition overlap
MCF Manager Program

```c
main(int argc, char **argv) {
    mcf_m_net_create();
    mcf_m_net_initialize();

    mcf_m_net_add_task();
    mcf_m_team_run_task();

    mcf_m_tile_distribution_create_3d("in");
    mcf_m_tile_distribution_set_partition_overlap("in");
    mcf_m_tile_distribution_create_3d("out");

    mcf_m_tile_channel_create("in");
    mcf_m_tile_channel_create("out");
    mcf_m_tile_channel_connect("in");
    mcf_m_tile_channel_connect("out");

    mcf_m_tile_channel_get_buffer("in");

    // fill input data here

    mcf_m_tile_channel_put_buffer("in");
    mcf_m_tile_channel_get_buffer("out");

    // process output data here
}
```

1. Add worker tasks
2. Specify data organization
3. Create and connect to tile channels
4. Get empty source buffer
5. Fill it with data
6. Send it to workers
7. Wait for results from workers
mcf_w_main (int n_bytes, void * p_arg_ls) {
    mcf_w_tile_channel_create("in");
    mcf_w_tile_channel_create("out");
    mcf_w_tile_channel_connect("in");
    mcf_w_tile_channel_connect("out");

    while (! mcf_w_tile_channel_is_end_of_channel("in")) {
        mcf_w_tile_channel_get_buffer("in");
        mcf_w_tile_channel_get_buffer("out");
        // Do math here
        mcf_w_tile_channel_put_buffer("in");
        mcf_w_tile_channel_put_buffer("out");
    }
}
MCF Implementation

• **Consists of**
  - PPE library
  - SPE library and tiny executive (12 KB)

• **Utilizes Cell Linux “libspe” support**
  - But amortizes expensive system calls
  - Reduces overhead from milliseconds to microseconds
  - Provides faster and smaller footprint memory allocation library

• **Based on Data Reorg standard**
  - [http://www.data-re.org](http://www.data-re.org)

• **Derived from existing Mercury technologies**
  - Other Mercury RDMA-based middleware
  - DSP product experience with small footprint, non-cached architectures
MCF Experience

• **Works very well for regular N-dimensional matrix data and image data**
  - Used by customers and for almost all internal application development

• **Next steps**
  - *Data re-org channels* for SPE-SPE data movement
  - Support for irregular, variable length data
  - Better integration with cluster-level middleware
  - Overlays / code movement
  - SPE emulation on GPP to ease debugging
  - Extend to GPP multicores

• **What we need from the community for our “performance first” approach:**
  - OS to get out of the way of SPE and DMA performance
  - Compilers that treat carefully written SPU intrinsics code with respect
Summing Up

• The Cell BE processor can achieve one to two orders of magnitude performance improvement over current general purpose processors
  ▪ Lean SPE core saves space and power
  ▪ And makes it easier for software to approach peak performance

• Cell is a distributed memory multiprocessor on a chip
  ▪ Prior experience on these architectures translates easily to Cell

• But for most programmers, Cell is a new architecture
  ▪ Successful adoption by programmers is Cell’s biggest challenge
  ▪ And the history of other new processor architectures is not encouraging

• We need a range of tools that span the continuum from ease-of-use to high performance
Final Thoughts:
Criteria for Evaluating Cell Software Tools