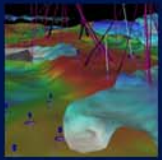
Computer Systems, Inc.
MERCURY
Challenges Drive Innovation







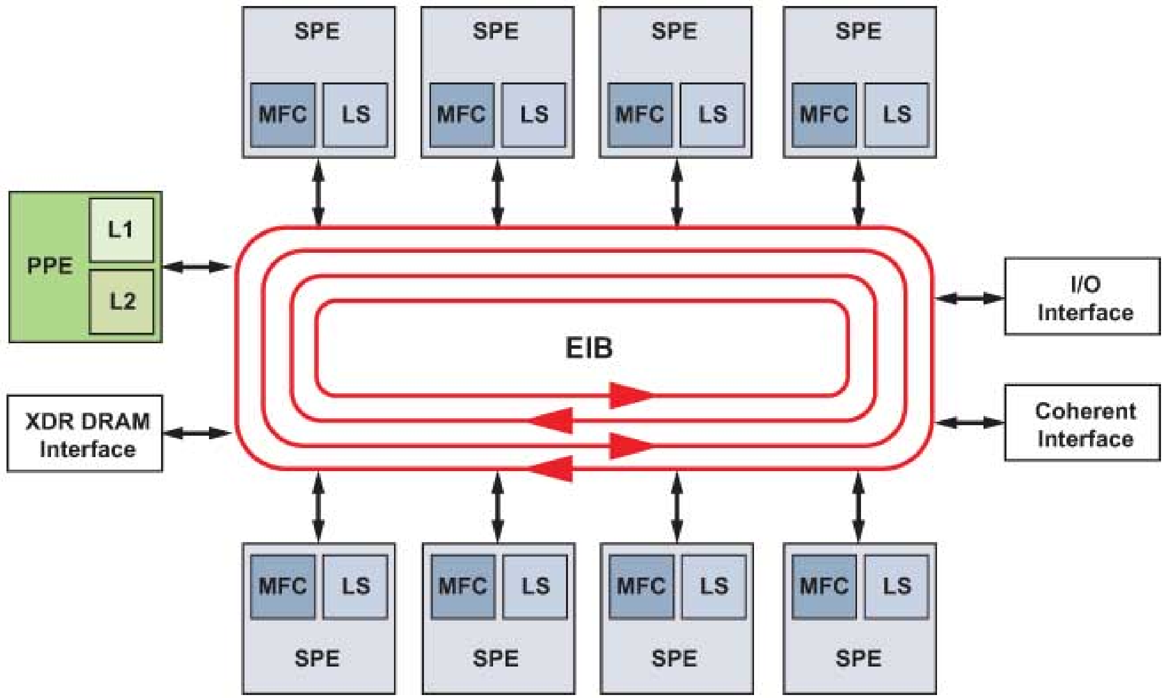

Performance and Programmability of the Cell Broadband Engine Processor

Robert Cooper, Brian Bouzas, Luke Cico, Jon Greene, Maike Geng, Frank Lauginiger, Michael Pepe, Myra Prelle, George Schmid, Matt Sexton

Agenda

- **Programmability**
  - Cell architecture and performance considerations
  - MultiCore Framework
- **Performance**
  - Chip level performance
  - SPE performance
- **Summary**

Cell BE Processor Architecture



- **Cell BE processor boasts nine processors on a single die**
  - 1 Power® processor
  - 8 vector processors
- **Computational Performance**
  - 205 GFLOPS @ 3.2 GHz
  - 410 GOPS @ 3.2 GHZ
- **A high-speed data ring connects everything**
  - 205 GB/s maximum sustained bandwidth
- **High performance chip interfaces**
  - 25.6 GB/s XDR main memory bandwidth

Programming the Cell Processor

- **Easiest aspects of programming Cell**
  - Very deterministic SPE performance
  - Generous ring bandwidth
  - Standards compliant Power® core.
- **Biggest challenges for software**
  - SPE can directly access only 256KB of local store
    - Can be viewed as a large (256KB) L1 cache
    - But getting code and data into and out of it is the job of software
    - Code and data from main memory must be DMA'ed using the Memory Flow Controller (MFC)
    - SPE instruction set includes instructions for DMA initiation and synchronization
  - SPE context switch is expensive
    - Must save registers, local store contents, and outstanding DMAs (if any)

Keys to Performance

Mercury Computer Systems, Inc. — Challenges Drive Innovation

- **PPE performance**
  - Use the PPE for control code
- **SPE performance**
  - Decompose algorithm into chunks that can utilize 256K local store
  - Use run-to-completion model
  - Overlap computation with DMA using double or triple buffering
  - Vectorize inner loop SPE code (4-way SIMD for 32-bit float operations)
- **EIB / XDR performance**
  - Pay careful attention to XDR bandwidth utilization
  - Use 128-byte alignment of data and multiples of 128-byte transfers for maximum DMA performance
  - Exploit SPE-to-SPE ring bandwidth if possible
  - Generally don't need to worry about aggregate EIB bandwidth
- **Dual Cell blade considerations**
  - Use PPE, XDR and SPEs on same Cell BE chip
  - Use Linux support for processor affinity, memory affinity (NUMA) and SPE affinity

Keys to Performance

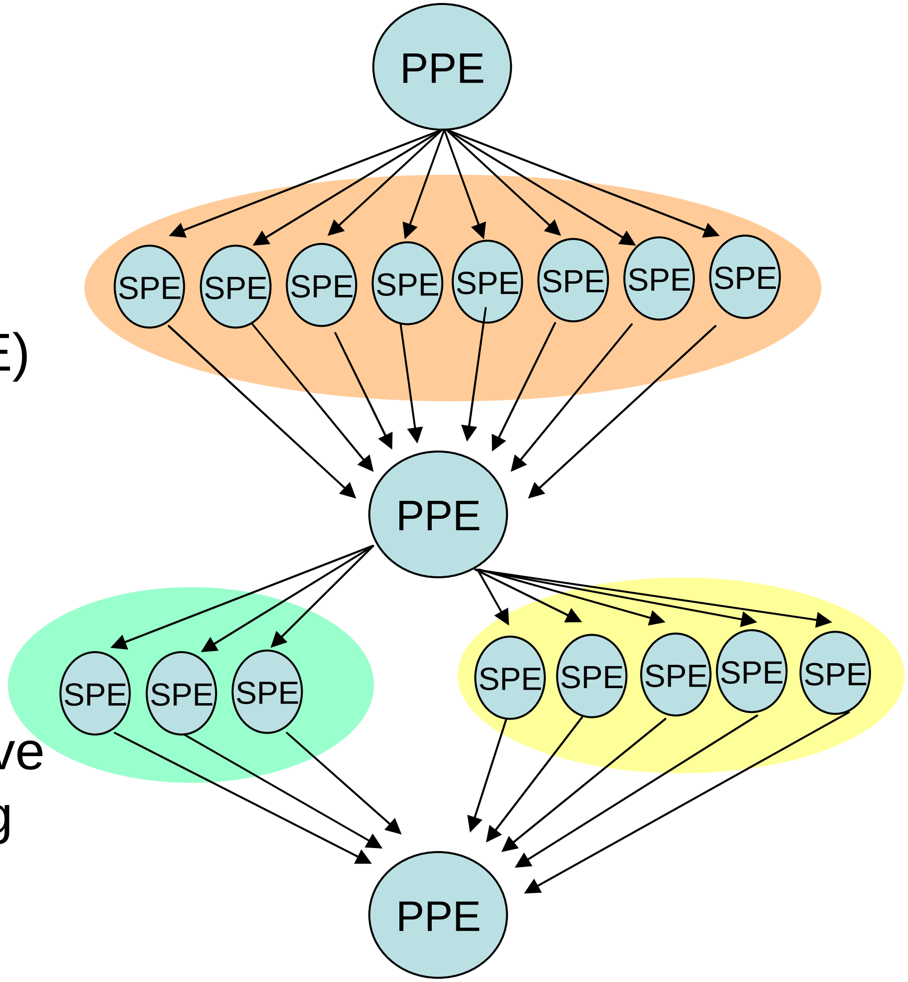
- **PPE performance**
  - Use the PPE for control code
- **SPE performance**
  - Decompose algorithm into chunks that can utilize 256K local store
  - Use run-to-completion model
  - Overlap computation with DMA using double or triple buffering
  - Vectorize inner loop SPE code (4-way SIMD for 32-bit float operations)
- **EIB / XDR performance**
  - Pay careful attention to XDR bandwidth utilization
  - Use 128-byte alignment of data and multiples of 128-byte transfers for maximum DMA performance
  - Exploit SPE-to-SPE ring bandwidth if possible
  - Generally don’t need to worry about aggregate EIB bandwidth
- **Dual Cell blade considerations**
  - Use PPE, XDR and SPEs on same Cell BE chip
  - Use Linux support for processor affinity, memory affinity (NUMA) and SPE affinity

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***
- Goals
  - ***High performance***
  - ***Ease of use***
- First implementation is for the ***Cell BE processor***

MCF Programming Model

- **Function offload engines**
  - Use SPEs as math processors
- **Write code for both processing elements.**
  - Control code for manager (PPE)
  - Algorithms for workers (SPEs)
- **View PPE & XDR memory as traditional multicomputer node.**
  - Use favorite middleware to move data and coordinate processing among nodes



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

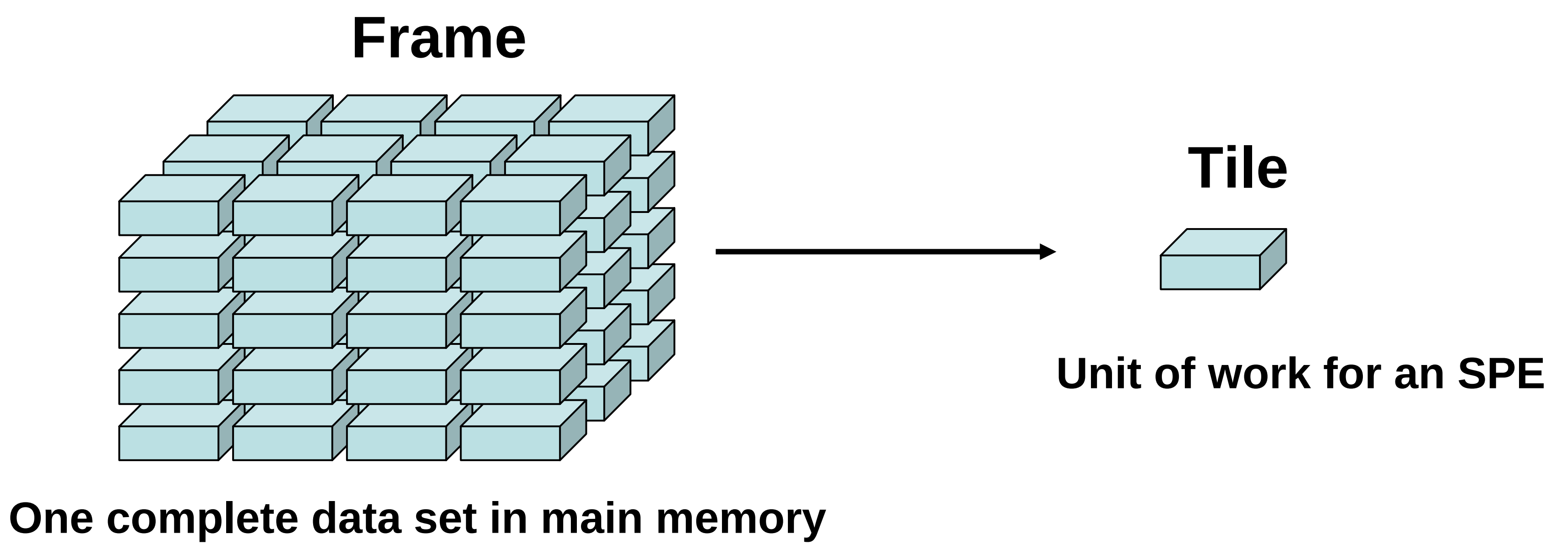
MCF Distribution Objects

Frame

One complete data set in main memory

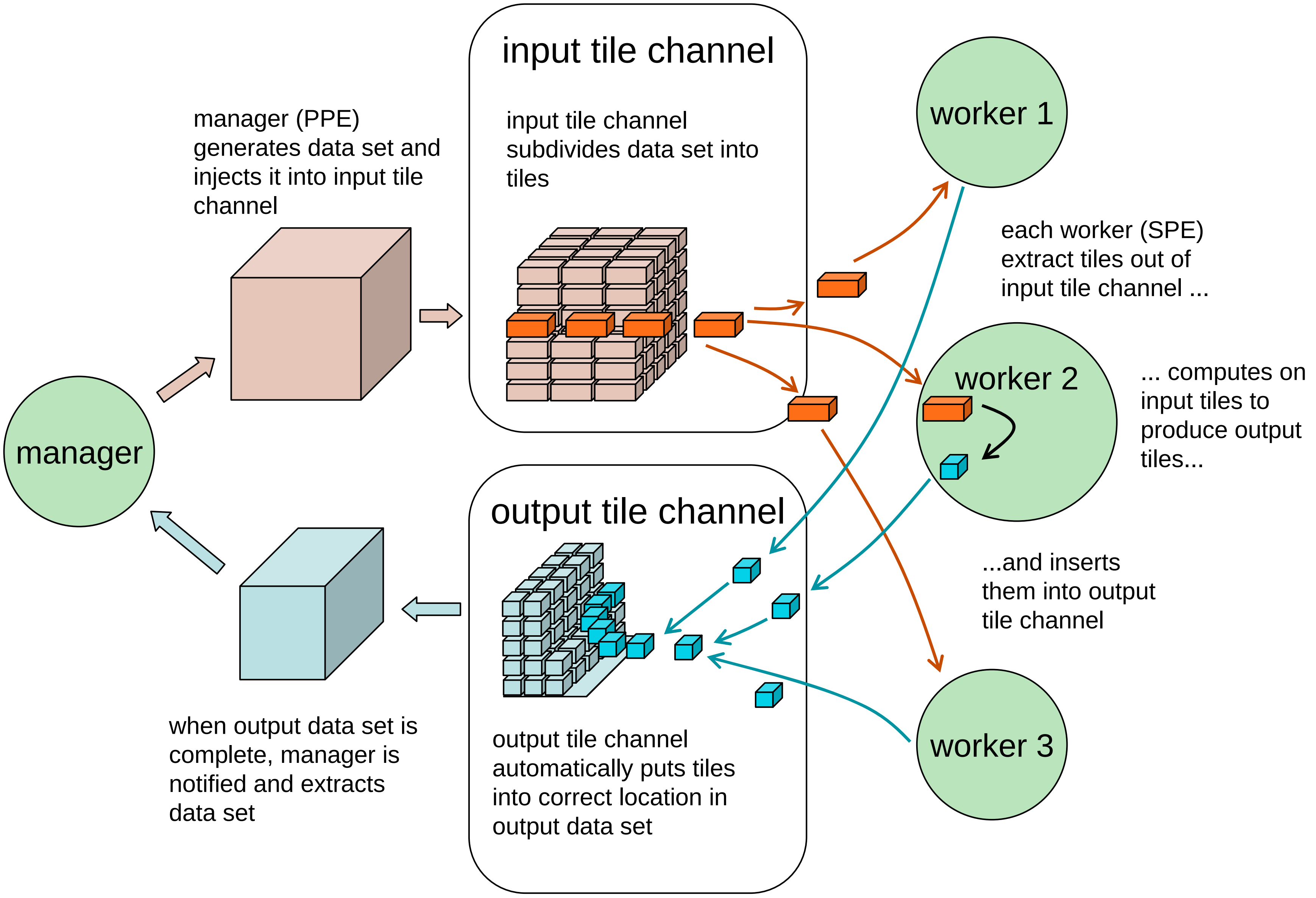
- **Distribution Object parameters:**
  - Number of dimensions
  - Frame size
  - Tile size and tile overlap
  - Array indexing order
  - Compound data type organization (e.g. split / interleaved)
  - Partitioning policy across workers, including partition overlap

MCF Distribution Objects



- **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



MCF Manager Program

```
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
}
```

Add worker tasks

Specify data organization

Create and connect to tile channels

Get empty source buffer

Fill it with data

Send it to workers

Wait for results from workers

MCF Worker Program

```
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”);
   }
}
```

Create and connect to tile channels

Get full source buffer

Get empty destination buffer

Do math and fill destination buffer

Put back empty source buffer

Put back full destination buffer

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
- **Derived from existing Mercury technologies**
  - PAS data partitioning
  - DSP product experience with small footprint, non-cached architectures

Agenda

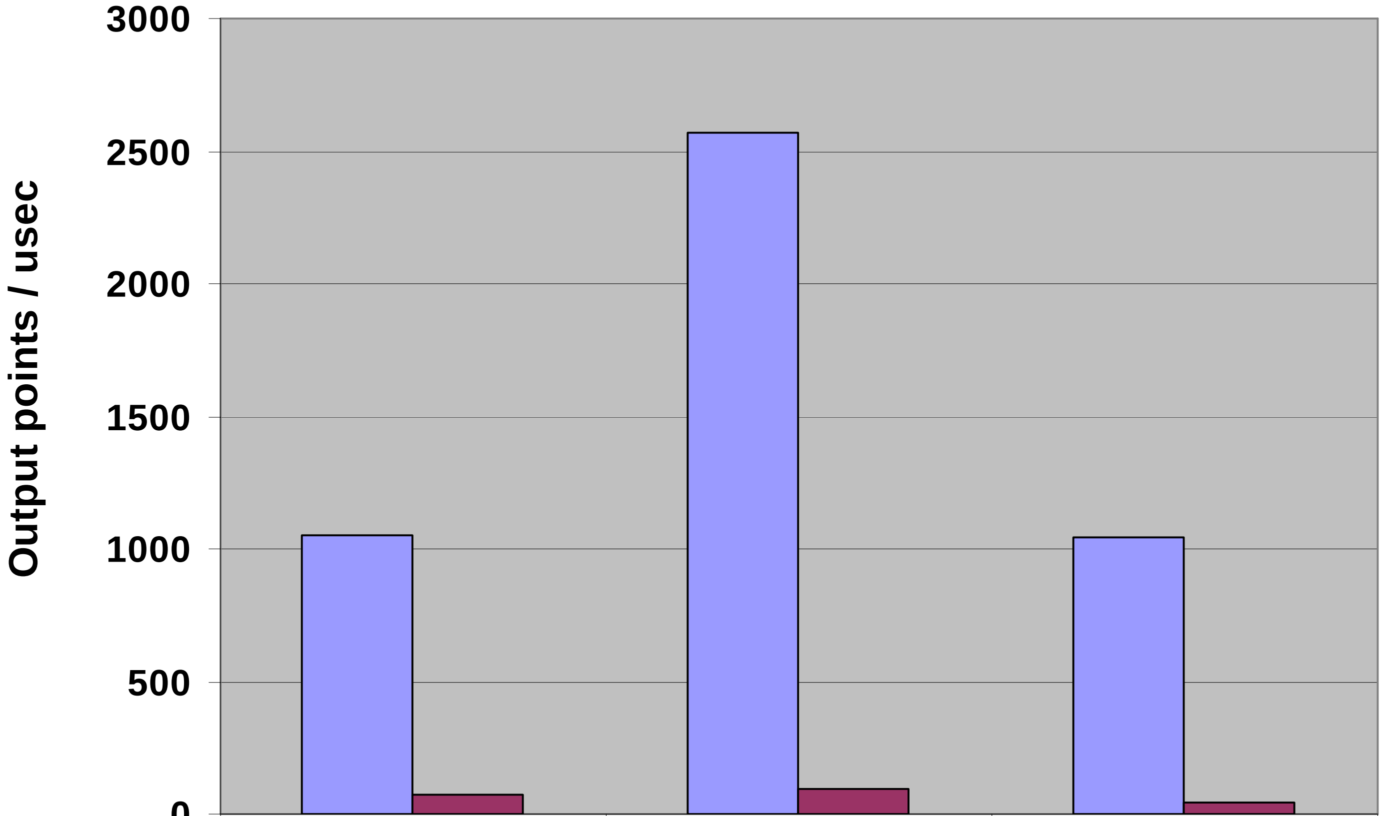
- **Programmability**
  - Cell architecture and performance considerations
  - MultiCore Framework
- **Performance**
  - Chip level performance
    - Large image filters
    - Parallel FFT
  - SPE performance
    - Small FFTs
- **Summary**

Large Image Filters

- **15x15   8-bit symmetric filter**
- **7x7   16-bit separable filter**
- **15x15   16-bit symmetric filter**

- **Images are 2048 x 1024 8 bit or 16 bit pixels**
- **Function offload from PPE**
  - Execution time is latency of blocking PPE call
  - Data starts and ends in XDR

Image Filter Performance: Latency

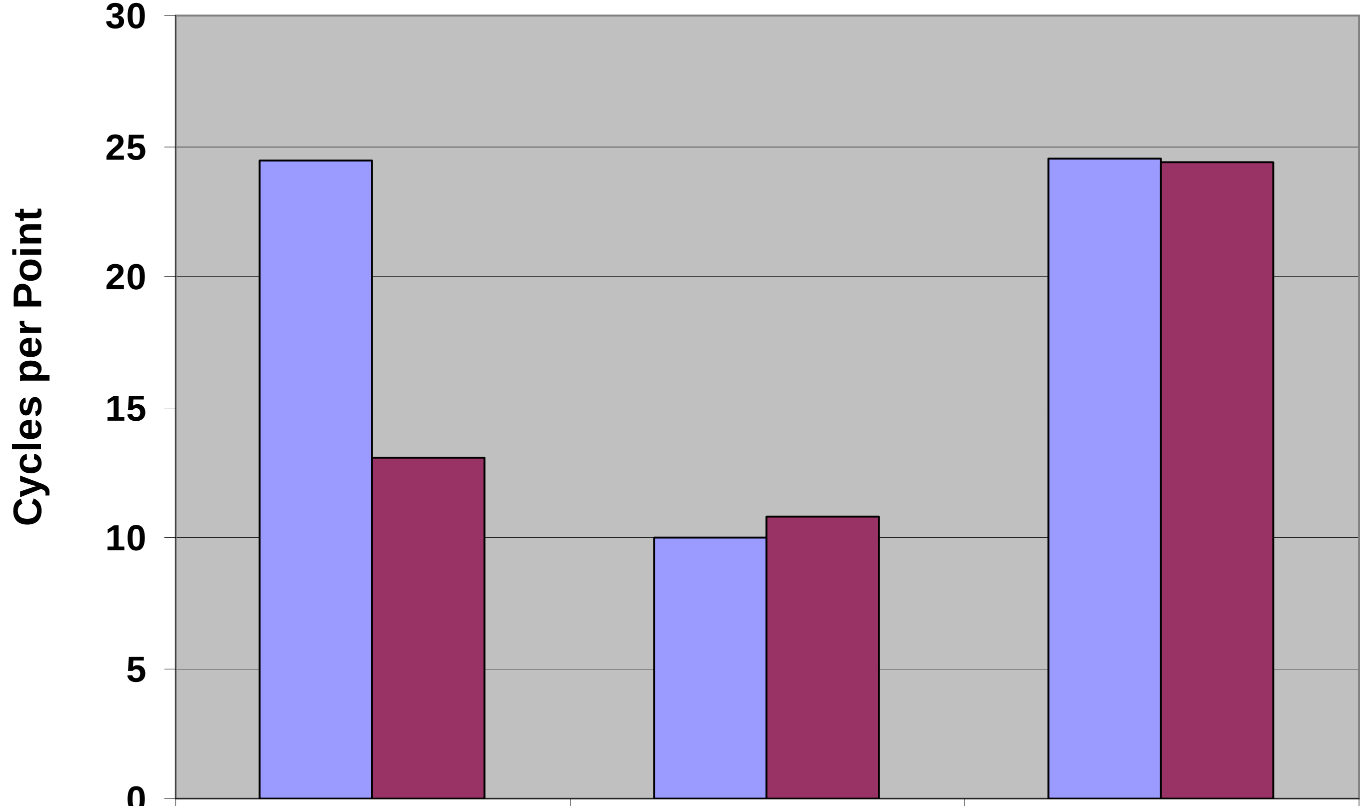


	15x15 8 bit symmetric filter	7x7 16 bit separable filter	15x15 16 bit symmetric filter
Cell BE (8 SPEs) @ 3.2GHz	1048	2566	1043
Freescale 7445 @ 1GHz	76	93	41

Image Filter Algorithms

- **Measured on 3.2 GHz Dual Cell Based Blade**
- **Cell performance is remarkable since the SPE only provides 4-way MACs (multiply-accumulates)**
  - Altivec/VMX provides 8-way 16-bit and 16-way 8-bit MACs
- **Conventional processors are penalized by**
  - Cache unpredictability
  - Cache complexity (area and power consumption)
  - Register starvation
- **It is *much* easier to achieve near to theoretical peak operations per clock on the SPE**

Image Filters: Instruction Set Efficiency

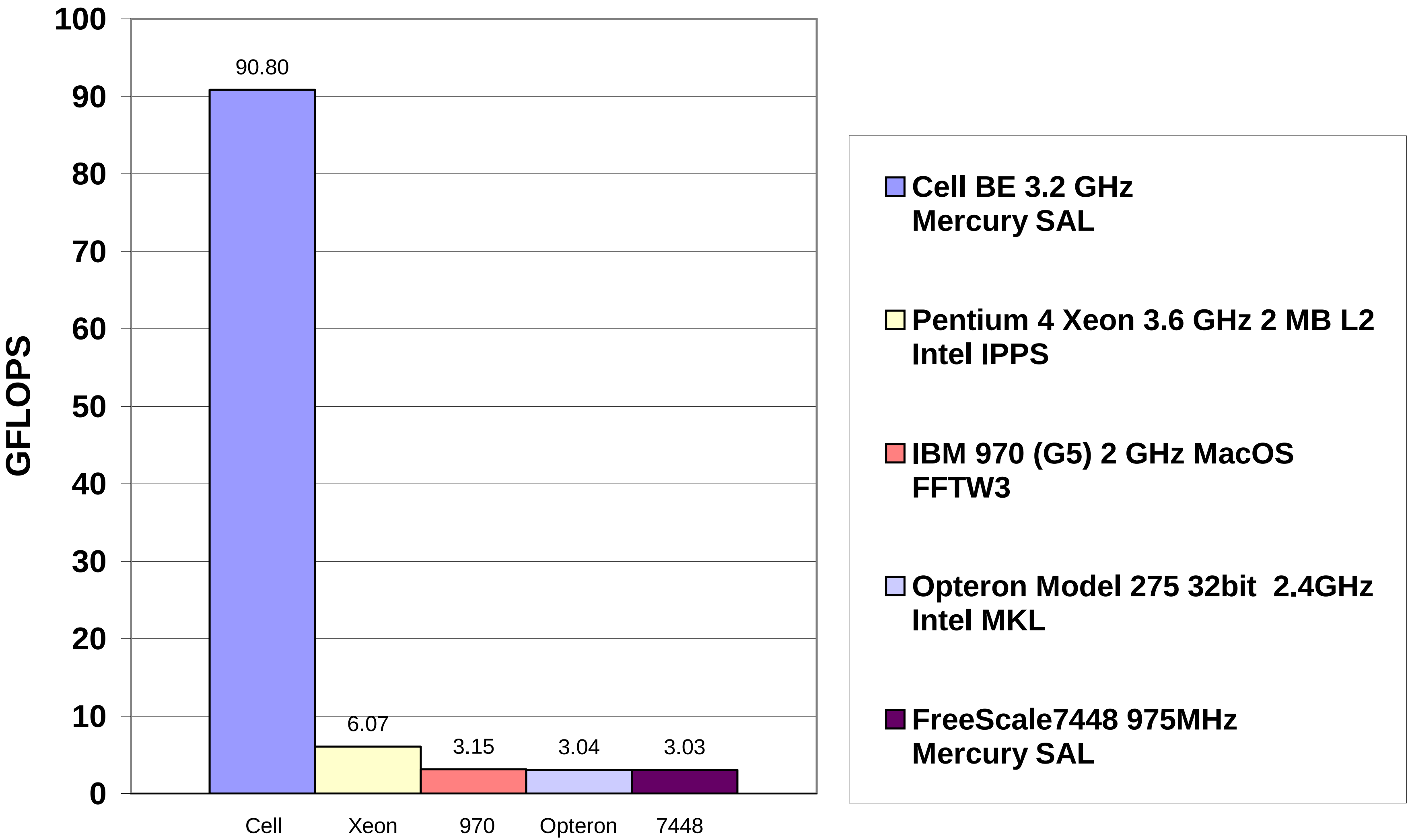


	15x15 8 bit symmetric filter	7x7 16 bit separable filter	15x15 16 bit symmetric filter
Single SPE @ 3.2GHz	24	10	25
Freescale 7445 @ 1GHz	13	11	24

Parallel 64K FFT

- **Parallel implementation of a sequence of 64K point single precision complex FFTs**
- **SPE-to-SPE communication is essential to achieve optimal performance**
  - Data does not fit in a single SPE’s local store
  - But does fit in the sum of all 8 local stores

64K FFT Performance



64K Single Precision Complex FFT

64K FFT– Algorithm Overview

- **Utilizes performance of entire Cell chip**
  - Utilize 8 SPEs, EIB ring bandwidth, XDR bandwidth
- **All data begins and ends in XDR**
- **During each FFT computation, SPEs exchange data in one all-to-all transfer**
- **Triple buffering in local store**
  - Allows overlapping of SPE computation with transfers to/from XDR and SPE-to-SPE transfers
  - While one FFT computation is underway:
    - Results from previous FFT are being DMA'ed back to XDR and
    - Data for next FFT is being DMA'ed from XDR

64K FFT – Footnotes

- **Measured on 3.0 GHz Cell Accelerator Board and scaled up to 3.2 GHz**
  - We report GFLOPS of *throughput*
    - I.e. the time between completions of successive FFTs when performing a stream of multiple FFTs
  - Details in paper presented at GSPx 2005
- **We use "normalized" GFLOPS computed as 5 N log (N)**
  - This is what FFTBench uses
  - Actual executed GFLOPS is lower per FFT due to algorithm optimizations
- **Comparison with**
  - Freescale 7448 with optimized Mercury SAL
  - Intel P4, IBM 970 and AMD Opteron with the fastest algorithm reported on public BenchFFT site http://www.fftw.org/benchfft/

64K FFT – Footnotes: Local Store Usage

- **Our 64K FFT algorithm requires approximately 253 Kbytes (out of the available 256 Kbytes) of local store in each SPE:**
  - Stack size: 8K
  - Code: 31K
  - DMA lists (2): 8K
  - Data buffers (3): 192K
  - Twiddles: 12K
- **Total: 253K**

Why is Cell so Fast for this Example?

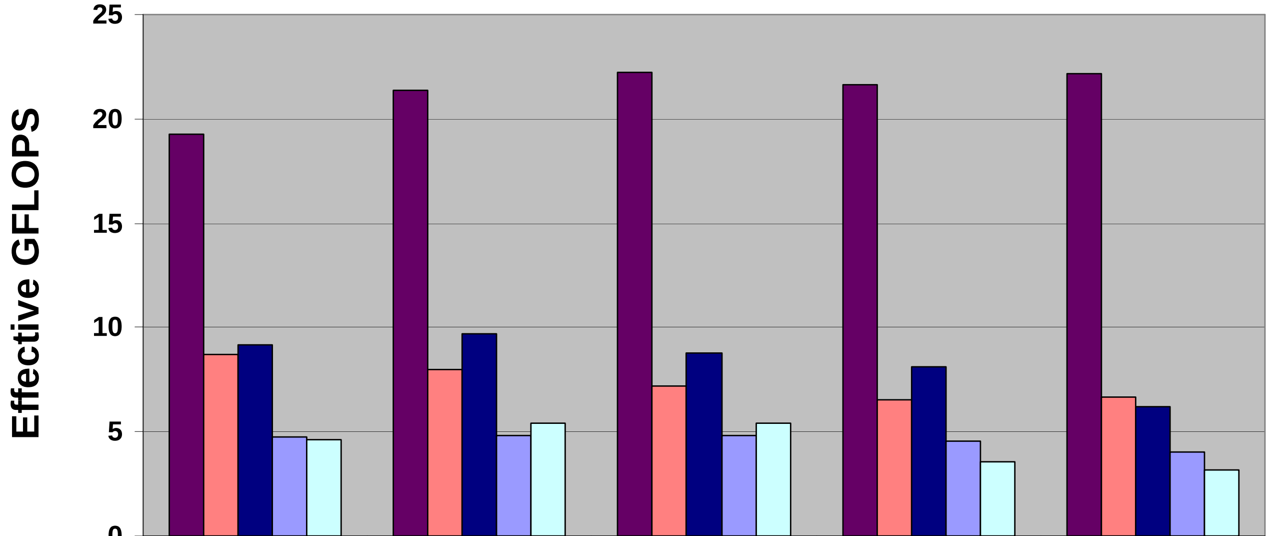
- **Between 15 and 30 times faster than comparable GPPs for this algorithm**
- **Huge inter-SPE bandwidth**
  - 205 GB/s sustained throughput
- **Fast main memory**
  - 25.6 GB/s XDR bandwidth
- **Predictable DMA latency and throughput**
  - DMA traffic has negligible impact on SPE local store bandwidth
  - Easy to overlap data movement with computation
- **High performance, low power SPE cores**

Small FFT Performance

- **256 to 8192 point single precision complex FFTs**
  - SPE local store resident
  - Mostly L1/L2 resident on GPPs
- **Measured on real HW**
  - Theoretical peak is 25.6 GFLOPS per SPE
- **Comparison with**
  - Freescale 7448 with optimized Mercury SAL
  - Intel P4, IBM 970 and AMD Opteron with the fastest algorithm reported on public BenchFFT site http://www.fftw.org/benchfft/

Small Single Precision Complex FFTs

Single SPE performance comparison



	256	1024	2048	4096	8192
Cell Single SPE 3.2GHz Mercury SAL	19.29	21.34	22.21	21.61	22.15
Xeon em64T 3.6GHz Intel IPPS / FFTW3	8.68	8.01	7.18	6.53	6.66
PPC 970 G5 MacOS 2GHz FFTW3 / VDSP	9.17	9.69	8.75	8.09	6.23
Opteron 275 32bit mode 2.4GHz FFTW3	4.77	4.82	4.82	4.55	4.05
Freescale 7448 1.075GHz Mercury SAL	4.63	5.41	5.44	3.58	3.19

FFT Size

Why is the SPE So Fast?

Computer Systems, Inc. MERCURY Challenges Drive Innovation

- **A single SPE core outperforms general purpose cores by up to a factor of 7**
  - Outperforms the highest clocking Pentium single core (3.6 GHz) by a factor of up to 3
- **Reasons**
  - 256KB local store vs. 32KB L1 caches
  - Local store access time is deterministic and local store occupancy is under programmer control
    - No reverse engineering or second guessing about the cache replacement policy
  - 128 registers, each 128 bits long
    - No register starvation when unrolling loops to mask the latency of the pipelines
- **And remember, there are *eight SPEs* in Cell**

Summary

- **The Cell BE processor can achieve one to two orders of magnitude performance improvement over current general purpose processors**
  - Lean SPE core and explicit memory hierarchy saves space and power
  - And makes it easier for software to approach theoretical peak performance
- **The Cell BE architecture is a distributed memory multiprocessor on a chip**
  - Prior experience on these architectures translates easily to Cell