Programming the Cell Multiprocessor:
A Brief Introduction

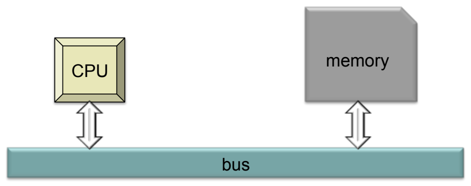
David McCaughan, *HPC Analyst*
SHARCNET, University of Guelph
`dbm@sharcnet.ca`

Overview

- Programming for the Cell is non-trivial
  - many issues to be addressed
    - explicit management of heterogeneous compute cores
    - explicit management of limited memory bandwidth
    - explicit vectorization of code
  - it will be impossible to cover all of this in a meaningful way in 3 hours
- Our goal:
  - ramp up our understanding of the major architectural issues
  - begin the process of getting hands-on experience with the Cell Broadband Engine
  - set the stage for taking our next steps
    - I will attempt to follow-up on this course with a more in-depth treatment over AccessGrid (possibly as a multi-part offering) at some point over the summer

A brief history of multiprocessing

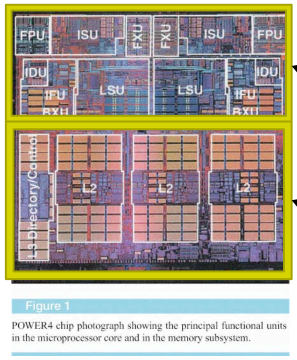
- In the beginning…
  - Von Neumann architecture
    - one processing unit + one memory + bus
    - processor executes one instruction at a time
    - other device elements share the bus for data transfer



Performance and the single processor

- Two bottlenecks:
  - compute
    - increased in agreement with Moore's Law
  - memory bandwidth
    - bus typically slow (fraction of CPU clock)
    - memory access slower due to physics
    - increases more slowly than compute
- Strategies:
  - pipelining
  - caching

Example: IBM Power 4

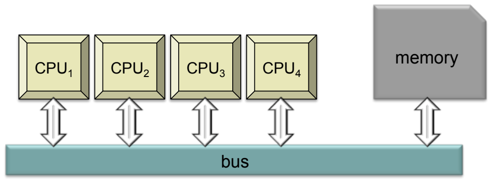

FPU
ISU
FXU
FXU
ISU
FPU
IDU
IFU
BXU
LSU
LSU
IDU
IFU
BXU
L3 Directory/Control
L2
L2
L2
Figure 1
POWER4 chip photograph showing the principal functional units in the microprocessor core and in the memory subsystem.

power core

cache

Multiprocessing: compute

- <u>Step 1</u>: *if it isn't broken, don't fix it*
  - Von Neumann architecture is adaptable
    - add processors sharing existing data bus
    - issues:
      - cache coherence
      - memory bandwidth



Bus architecture

- Benefits:
  - simple design
  - easy to maintain cache coherence
- issues:
  - contention for the bus
  - performance degrades quickly
    - amplifies memory bandwidth issues

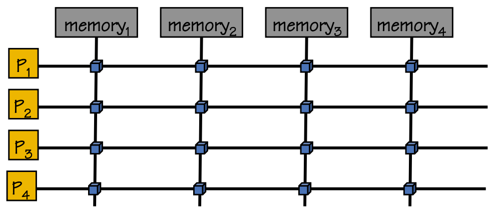
Multi-processing: memory

- <u>Step 2</u>: *try to keep the compute beast fed*
  - limited benefit from having huge compute that cannot be kept busy with data
  - communication with memory is the issue
- Possible solutions:
  - crossbar switch
  - non-uniform memory access (NUMA)

Crossbar switch

- Each piece of hardware has a bus-like communication channel organized so that they interconnect in a grid
  - some path exists between each piece of hardware through the crossbar
  - attempts to mitigate bottleneck a single bus produces
- Switches at intersections route connections as necessary
  - must also resolve contention when more that one device wishes to access the same hardware

Crossbar switch (cont.)



A 4x4 Crossbar Switch

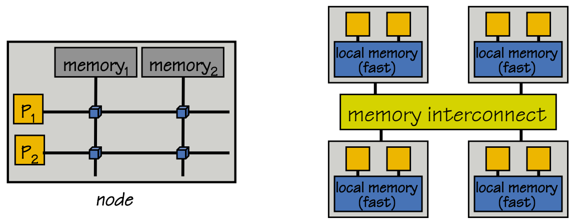
Crossbar switch (cont.)

- Benefits
  - uniform memory access
  - efficient shared memory
  - programmer's view of system is simplified
- Issues
  - cache coherence (requires some means of broadcasting updates)
  - memory access patterns still significant
    - processors hammering the same block of memory will still saturate available bandwidth
  - does not scale well
    - increasing complexity (particularly the switches)
    - increasing cost

Non-uniform memory (NUMA)

- Allow models to scale by abandoning uniform memory access
  - processors organized into nodes containing processors and local (fast) memory
  - nodes connected using an interconnect
  - local memory will be much faster than accessing memory on another node
    - shared memory systems may make global memory appear as a single memory space
    - in the extreme case is message passing in a cluster where there is no direct remote memory access
  - on some level, this is simply a different way to look at the distributed memory paradigm

Non-uniform memory (cont.)

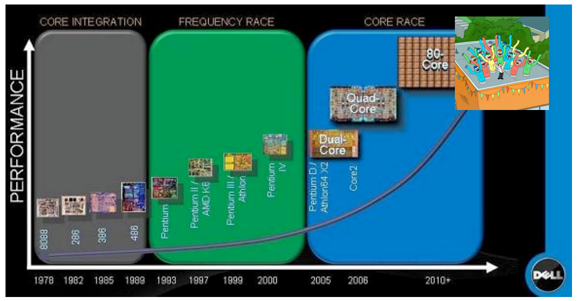


A Non-Uniform Architecture Constructed from 2-Processor Nodes

Multi-processing: the rise of multi-core

- For years, larger SMP machines with increasingly complex crossbar switches were built
  - FLOP/$ so poor that clusters dominated
  - note: large SMP solutions today are all NUMA
    - widely used; a fair trade, but ***not*** true SMP
- Multi-core
  - multiple compute cores within a single chip
  - dual-core was cheap, but nearly as powerful as a true dual-processor box
    - *warning: now is when the marketing types typically take over*
    - Intel/AMD/SUN/Dell touting 64+ cores on a chip
      - is this realistic?

Marketing; or "wacky, waving, inflatable, arm flailing tube man!"



DELL slide from SC'08 (minus cartoon)

Multi-core: the reality sets in

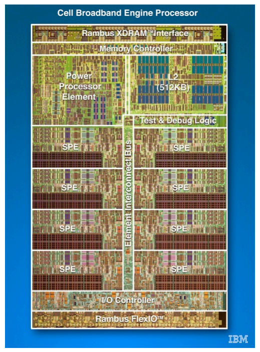
- Have faith that chip designers *can* design/build a chip with as many cores as they'd like
  - is this somehow circumventing the issues that plagued scaling of multi-processor systems?
  - have we solved the crossbar problem?
  - have we actually introduced yet another level of memory access complexity?
    - consider: sharing a cache with 60 cores
- Strategies:
  - build more specialized hardware that works given the known limitations rather than pretending they don't exist
  - shift burden of leveraging hardware to developer/compiler

Multi-core: the next generation

- Old way:
  - multiple general purpose CPU cores on a chip (***homogenous***)
- New way:
  - since we can't practically use a large number of general purpose CPUs on the same chip anyway, specialize the cores to expose what *is* possible (***heterogeneous***)
- Homogenous multi-core:
  - non-uniform threads
  - highly specialized algorithms; limit bandwidth demands across threads
- Heterogeneous multi-core:
  - GP-GPU (extension of classical SIMD)
  - Cell processor

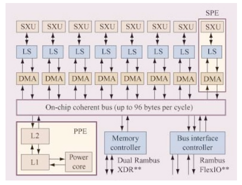
Cell processor

- PPE
  - power processor element
  - general purpose core
  - L1/L2 cache
- SPE
  - synergistic processor elements
  - DMA unit
  - local store memory 256KB (LS)
  - execution units (SXU)
- On-chip coherent bus (EIB)
  - allows single address space to be accessed by PPE/SPEs
  - high bandwidth (200GB/s+)



Cell processor (cont.)

- Programming strategies:
  - fine-grained control over flow of data between main memory & local store
    - DMA hardware gather/scatter data
    - explicit control of data flow
      - hide latency (read/write/compute)
    - avoid hardware cache management
- Allows much more efficient use of available bandwidth
  - the trade-off being it's all you doing it
  - contrast with EPIC architecture
    - *only* compiler-centric instruction scheduling



Feeding the multi-core beast (IBM)

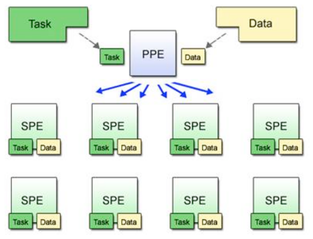
- It's now *all* about the data
  - orchestrate the delivery/removal of data
  - keep as much of the compute resources as possible working at all times
- Constraints:
  - main memory latency
  - size of local store
  - bandwidth of coherent bus
- And none of this is automated for you!
  - you should start to see why I'm on about ease of programming taking steps backwards in order to leverage power

Programming the Cell

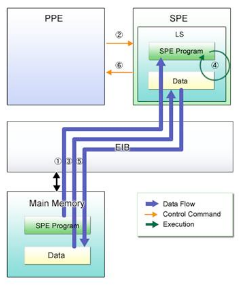
- Basic idea:
  - control functions assigned to PPE
  - calculation functions assigned to SPEs
- Maximizing performance involves:
  - operate SPEs in parallel to maximize instructions per unit time
  - perform SIMD parallelization on each SPE to maximize instructions per cycle

Basic programming model

- PPE used for execution of the main program
  - SPEs execute sub-programs, making use of DMA transfers to get/put data independent of PPE



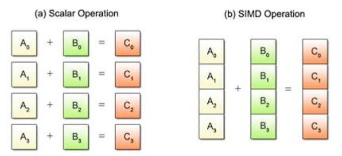
Program control and data flow



- [PPE]:
  1. load SPE program into LS
  2. instruct SPE to execute program
- [SPE]:
  1. transfer data memory -> LS
  2. process data
  3. transfer results LS -> memory
  4. notify PPE processing is done

SIMD programming

- Similar to Altivec/SSE/etc.
- Large registers (128 bit)
  - explicit vector instructions can manipulate register contents as multiple values simultaneously
    - 4 x 32-bit float, 8 x 16-bit integer, 16 x 8-bit integer, etc.



Cell programming fundamentals

- SPE Runtime Management Library (libspe v2)
  - controls SPE program execution from PPE program
  - handles SPEs as virtual objects (SPE contexts)
    - SPE programs are loaded and executed by operating on SPE contexts (note: there are file system issues here)
- Programming basics:

```
#include "libspe2.h"
```

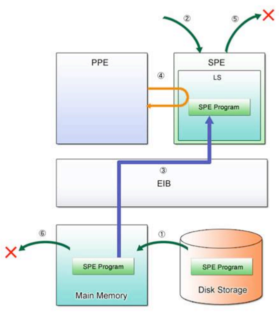
  - compile using cell-specific language support
    - IBM: xlc, gcc (requires gcc with support for vector extensions)
    - IBM provides a decent compilation framework we can (and will) use today --- see exercise

The SHARCNET Cell cluster: prickly.sharcnet.ca

- Heterogenous cluster:
  - dual quad-core Xeon@2.5GHz 8GB RAM x 4 nodes
  - dual PowerXCell 8i@3.2GHz 16GB RAM x 8 nodes
  - no scheduler currently
  - OpenMPI
- NOTE: you must be logged in to one of the Cell nodes in order to access the Cell development environments
- see also:
  - http://www.sharcnet.ca/help/index.php/Cell_Accelerated_Computing
  - this is the "new" training material – although not linked from the main page yet, you must be logged into the web portal to view it

PPE programming

- PPE program execution basics:
  1. open SPE program image
  2. create SPE context
  3. load SPE program into LS
  4. execute program on SPE
  5. destroy SPE context
  6. close SPE program image



SPE program image

- SPE program is compiled and linked as a separate entity from the PPE program
  - there is no OS running on the SPEs
  - PPE must explicitly copy the SPE image from a file to memory, and then explicitly invoke the DMA controller on a SPE to move the SPE image into LS
  - SPE-ELF images are embedded in the PPE binaries
    - declare external reference to force inclusion
    - i.e.:

```
extern spe_program_handle_t spe_program_handle;
```

Creating a SPE context

```
spe_context_ptr_t spe_context_create
(
    unsigned int flags,
    spe_gang_context_ptr_t *gang
);
```

- Create a new SPE context; if none available, will either block or return an error if you configure the library to do so
- `flags`
  - adjust behaviour/policies of requested SPE
- `gang`
  - allows for control of SPE affinity (if not NULL); contexts are assoiciated with physical SPEs randomly by libspe by default

Load SPE program into context

```
int spe_program_load
(
    spe_context_ptr_t spe,
    spe_program_handle_t *program
);
```

- Load an SPE main program into a provided SPE context (loads into SPE LS); returns 0 on success, non-zero (-1) otherwise
- `spe`
  - SPE context on which to load the program
- `program`
  - address of SPE program (recall extern reference to SPE program image)

Execute program on SPE

```
int spe_context_run
(
    spe_context_ptr_t spe, unsigned int *entry,
    unsigned int runflags, void *argp, void *envp,
    spe_stop_info_t *stopinfo
);
```

- Execute program loaded in provided SPE context (NOTE: blocking call!!)
- `entry`
  - entry point for program (SPE_DEFAULT_ENTRY = main); updated during call
- `runflags`
  - permit control of execution behaviour (0 = defaults)
- `argp, envp`
  - arguments, environment data --- passed to SPE main
- `stopinfo`
  - information about termination condition of SPE program

Destroy SPE context

```
int spe_context_destroy
(
    spe_context_ptr_t spe
);
```

- Destroy provided SPE context and free associated resources
- `spe`
  - SPE context to be destroyed

SPE program essentials

- SPE programs are particular entities that require specific compiler support
  - we use a different compiler for SPE compilation
  - recall: no OS runs on the SPE
- Special format for main()

```
int main( unsigned long long spe,
          unsigned long long argp,
          unsigned long long envp )
{

}
```

Example: "Hello, world!"

- SPE view: pretty much as expected
  - in order to provide more information, we make use of the speid provided to the main() routine in our output

```
#include <stdio.h>

int main( unsigned long long speid,
          unsigned long long argp,
          unsigned long long envp )
{
    printf("Hello, world, from SPE %lld!\n", speid);

    return(0);
}
```

Example (cont.)

- PPE controller for "Hello, world!"
  - note: this is just for 1 SPE running the program

```
#include <stdlib.h>
#include <stdio.h>
#include <libspe2.h>

extern spe_program_handle_t chello_spu;

/*
 * NOTE: these need not be global
 */
spe_context_ptr_t speid;
unsigned int entry;
spe_stop_info_t stop_info;
unsigned int flags = 0;
…
```

SPE programming (cont.)

```
    …

int main(int argc, char *argv[])
{
    entry = SPE_DEFAULT_ENTRY;

    if (!(speid = spe_context_create(flags, NULL)))
        /* error creating context */
    if (spe_program_load(speid, &chello_spu))
        /* error loading program into context */
    if ((spe_context_run(speid, &entry,
            0, NULL, NULL, &stop_info)) < 0)
        /* error running program in context */

    spe_context_destroy(speid);

    return(0);
}
```

Exercise

1. log into prickly and shell to one of the Cell nodes (pri5-pri12)
2. in `~dbm/pub/exercises/cell/chello` you'll find a working implementation of the "Hello, world!" program we just considered; copy this directory to your own workspace
3. examine the directory structure, source code and Makefile
4. build this program and execute it on the command line
5. modify the program to execute the SPE program a second time

Food for thought
- *did the second execution occur in parallel?*
- *did the second execution occur on the same SPE?*
- *think about how you would make it execute on the same and different SPEs*

Running SPEs in parallel

- Recall that `spe_context_run()` is a blocking call
  - the PPE program blocks until the SPE program exits
  - obviously we want things running on the SPEs in parallel --- how are we going to accomplish this?
- Answer: pthreads
  - the Cell BE expects the programmer to use POSIX threads to allow for multiple SPEs to execute concurrently
  - i.e. execute `spe_context_run()` in its own thread
    - each thread can block until the SPE program exits
    - allows for concurrent execution in SPEs

Pthreads review

- Include Pthread header file
  - `#include "pthread.h"`
- Compile with Pthreads support/library
  - `cc -pthread …`
    - compiler vendors may differ in their usage to support pthreads (link a library, etc.)
    - GNU and xlc use the `-pthread` argument so this will suffice for our purposes
    - when in doubt, consult the man page for the compiler in question

Pthreads Programming Basics (cont.)

- Note that all processes have an implicit "main thread of control"
- We need a means of creating a new thread
  - `pthread_create()`
- We need a way to terminating a thread
  - threads are terminated implicitly when the function that was the entry point of the thread returns, or can be explicitly destroyed using `pthread_exit()`
- We may need to distinguish one thread from another at run-time
  - `pthread_self(), pthread_equal()`

pthread_create

```
int pthread_create
(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*f_start)(void *),
    void *arg
);
```

- Create a new thread with a function as its entry point

- `thread`
  - handle (ID) for the created thread
- `attr`
  - attributes for the thread (if not NULL)
- `f_start`
  - pointer to the function that is to be called first in the new thread
- `arg`
  - the argument provided to f_start when called
  - consider: how would you provide multiple arguments to a thread?

pthread_join

```
int pthread_join
(
    pthread_t *thread,
    void **status
);
```

- Suspends execution of the current thread until the specified thread is complete

- `thread`
  - handle (ID) of the thread we are waiting on to finish
- `status`
  - value returned by `f_start`, or provided to `pthread_exit()` (if not NULL)

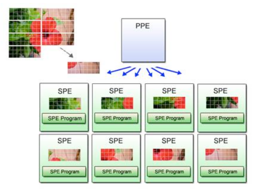
Exercise

1. refer again to the "Hello, world!" code in `~dbm/pub/exercises/cell/chello` you may wish to make another copy of the directory (or use the one you modified previously)
2. modify the program so that it loads the chello_spu program into all 8 SPEs and runs them concurrently
   - Note: you will need to modify the data structures to allow for the record-keeping associated with multiple SPE contexts, move the code responsible for executing the SPE program into a function that will be the entry point of the threads, and introduce loops to spawn threads to execute the SPE programs and wait for them to finish

<u>Food for thought</u>
- *how are you able to verify that the SPE programs are executing on different cores?*
- *if you did not use threads, how would the SPE programs execute?*
- *what can happen if you don't wait for the SPE running threads to exit (i.e. don't use pthread_join)?*

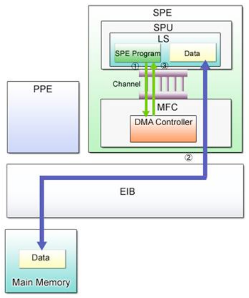
Toward more advanced programs

- PPE typically responsible for partitioning data
  - consider constraints
  - for multi-SPE execution:
    1. open SPE program image
    2. create all SPE contexts
    3. load SPE program into all LS memories
    4. execute SPE program on each thread
    5. wait for threads to terminate
    6. destroy all SPE contexts
    7. close SPE program image



Advancing the SPE: Data Management

- Going bigger:
  - must make use of DMA engine
  - data transfer between memory and LS
  - controlled by SPE program
    1. issue DMA transfer command
    2. execute DMA transfer
    3. wait for completion of transfer
  - note: fetch/execute/store typical for most tasks



Advancing the SPE: SIMD Programming

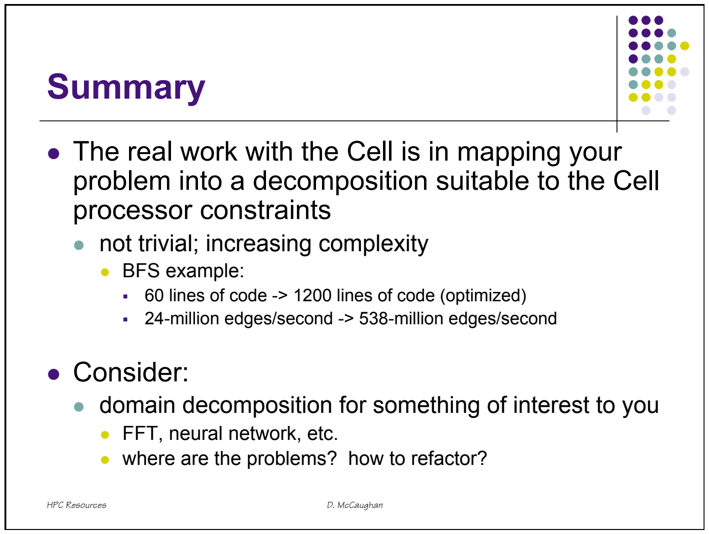
- To this point we have only discussed running programs in parallel on the SPE cores
- Each SPE core is a vector unit, and further speed-up is possible by using language extensions and vector operations
  - vector types corresponding to scalar types of merit
  - vector operations that operate on vector types
  - note: vector types map directly on to contiguous scalar types
    - can leave all data in arrays and step through them with pointers to the vector type
- The real power of the Cell is leveraging vectorized code running in parallel on SPEs

SIMD programming example

```
#include <stdio.h>
#include <altivec.h>  /* necessary to access VMX instructions */

int a[4] __attribute__((aligned(16))) = {1, 2, 3, 4};
int b[4] __attribute__((aligned(16))) = {5, 6, 7, 8};
int c[4] __attribute__((aligned(16)));

int main(int argc, char *argv[])
{
    __vector signed int *va = (__vector signed int *) a;
    __vector signed int *vb = (__vector signed int *) b;
    __vector signed int *vc = (__vector signed int *) c;

    *vc = vec_add(*va, *vb);
    printf("c = <%d, %d, %d, %d>\n",c[0],c[1],c[2],c[3]);

    return(0);
}
```

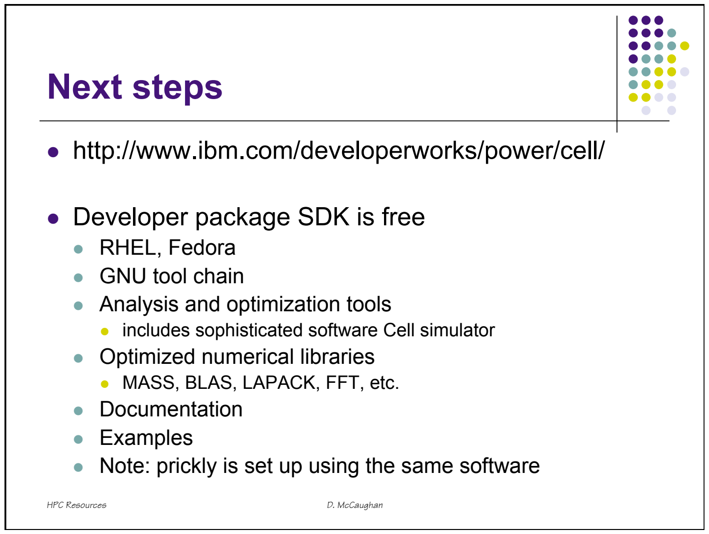
Other considerations

- We are only scratching the surface in this brief introduction
  - channels
    - communication and coordination facilities (e.g. mailboxes)
    - PPE <-> SPE and SPE <->SPE
  - gangs
    - affinity of SPEs can be important (consider channels)
    - gang contexts allow us to control mapping of contexts to physical SPEs
  - dealing with Cell constraints
    - memory bandwidth issues (consider main memory vs. EIB)
    - 256KB local store (potential for library issues!)
    - physical organization of EIB rings
    - memory alignment issues



Summary

- The real work with the Cell is in mapping your problem into a decomposition suitable to the Cell processor constraints
  - not trivial; increasing complexity
    - BFS example:
      - 60 lines of code -> 1200 lines of code (optimized)
      - 24-million edges/second -> 538-million edges/second
- Consider:
  - domain decomposition for something of interest to you
    - FFT, neural network, etc.
    - where are the problems? how to refactor?



Next steps

- http://www.ibm.com/developerworks/power/cell/
- Developer package SDK is free
  - RHEL, Fedora
  - GNU tool chain
  - Analysis and optimization tools
    - includes sophisticated software Cell simulator
  - Optimized numerical libraries
    - MASS, BLAS, LAPACK, FFT, etc.
  - Documentation
  - Examples
  - Note: prickly is set up using the same software