Concurrent Programming with the Cell Processor

Dietmar Kühl
Bloomberg L.P.
dietmar.kuehl@gmail.com

Copyright Notice

Overview

- what's special about the Cell?
- Cell SDK and how to build Cell programs
- creating threads using the full power
- communication within the Cell
- using SIMD for great performance

What's That All About

- processor clock cycles limits are reached
- many CPUs + shared memory: problems
  - contention on accessing memory
  - synchronization causes more complex CPUs
  - few know how to program this correctly

Cell's Approach

- one chip with many but heterogenous CPUs:
  - one general purpose CPU for controlling
  - many fast CPUs for computing
- memory is not directly shared between CPUs
  - communication is via fast channels

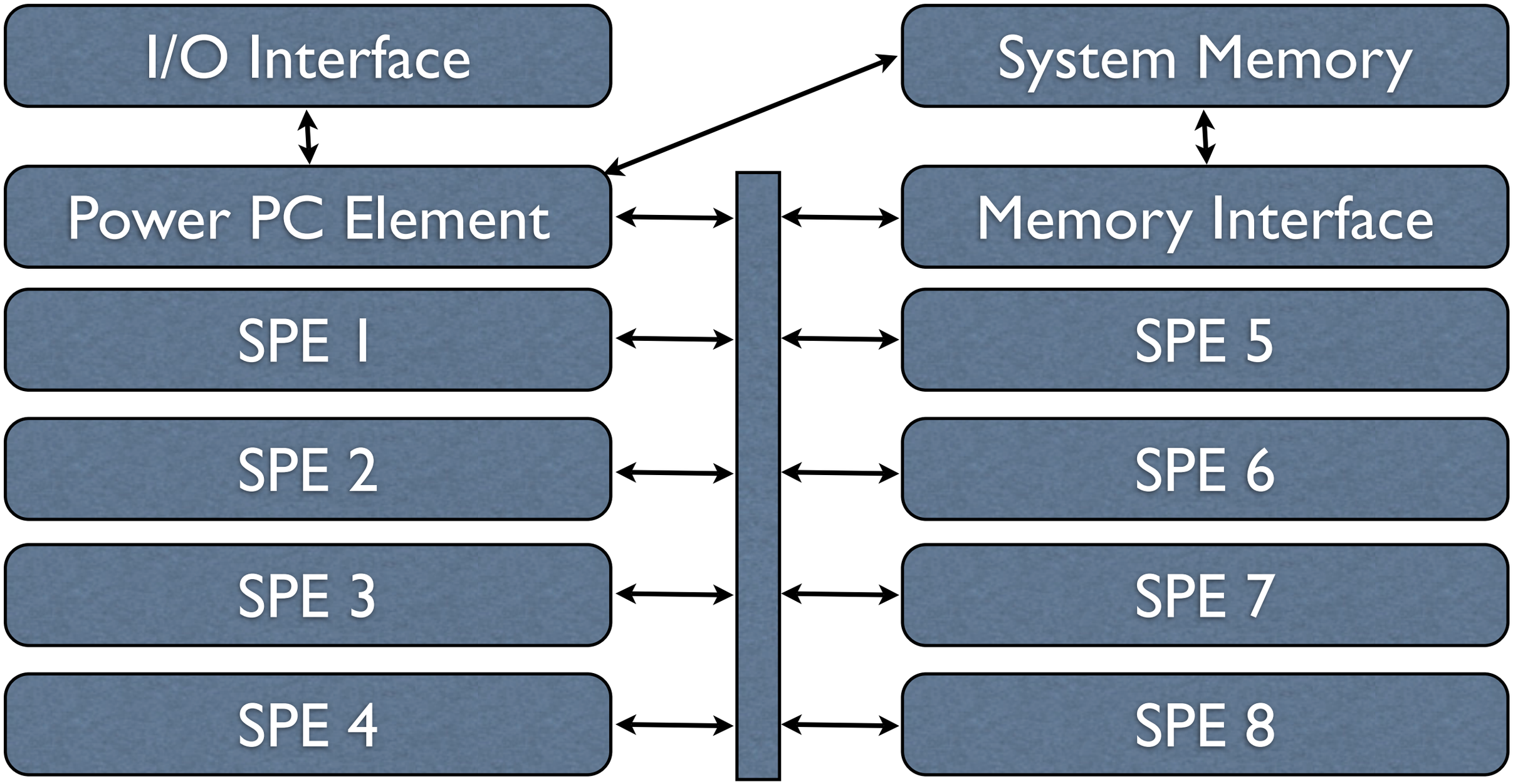
Cell Architecture

- two different CPUs: PPU and SPU
- 1 × PPU: Power PC for control, I/O, etc.
- 8 × SPU: Synergetic Processing Unit
  - simple SIMD CPU for number crunching
  - direct memory: no cache, no virtual memory

Cell Components

- MIC: memory interface controller
- EIB: element interconnect bus
- I/O interfaces
- PPE: PPU and its connection to MIC, EIB, I/O
- SPE: SPU and its connection to MIC and EIB

Cell Diagram



SPE: Processing Units

- own, limited memory: e.g. 256KB
- no memory protection, e.g. for stack overflow
- no cache, no virtual memory, no I/O
- uses asynchronous DMA for communication
- 128 bit SIMD processing for everything

Accessible Supercomputer

- employed e.g. in the PlayStation 3
- this has support to run Linux
- caveat: GPU and 2 SPUs are inaccessible
- other OS is intended: no need for a hack
- IBM provides a free SDK

Developing for the Cell

- IBM provides a free SDK for SPU and PPU
- GCC cross-compiler and cross-binutils
- libraries, in particular for SPU access: libspe
- debugger for PPU and SPU
- a Cell system simulator

Linux vs. PlayStation3

- non-destructive: you can still play games!
- use the “Other OS” option to boot Linux
- many PPC distributions work on the PS3
- Yellow Dog Linux is specifically for the Cell
- IBM provides add-ons for the SPU access

Building SPU Programs

- compile objects normally for respective CPU:
  - PPU: ppu-g++, ppu-gcc, ppu-as, ...
  - SPU: spu-g++, spu-gcc, spu-as, ...
- link SPU program: spu-gcc ...
- embed into PPU object: embedspu ...

Build Example

```
ppu-g++ -W -Wall -g -O2 -c ppe-hello.cpp
spu-g++ -W -Wall -g -O2 -c spe-hello.cpp
spu-gcc -W -Wall -g -O2 -c spe-main.c
spu-gcc -o spe-hello spe-hello.o spe-main.o

embedspu hello spe-hello embed.o

ppu-g++ -o hello ppe-hello.o embed.o -lspe
```

Running the Program

- cross-compile and run with the simulator
- cross-compile, transfer to Cell, run natively
- compile and run natively on the Cell
- there is a loader for stand-alone SPU programs

Creating SPU Threads

- functions from <libspe.h>
- spe_create_thread(grp, addr, arg, env, -1, 0)
- spe_wait(speid, &status, 0)
- SPUs are assigned by the library
- #threads can exceed #SPUs

SPU “hello world”

```
#include <stdio.h>
int main(unsigned long long id,
         unsigned long long arg,
         unsigned long long env)
{
  printf("id=%llx arg=%llx env=%llx\n",
         id, arg, env);
  return 0;
}
```

PPU "Hello World"

```
#include <libspe.h>
extern spe_program_handle_t hello;
int main()
{
  speid_t id = spe_create_thread(0, &hello,
                                 0, 0, -1, 0);
  if (id)
    spe_wait(id, 0, 0);
}
```

C++ PPU “hello world”

```
try {
  cool::spe_thread thread(hello);
}
catch (std::exception const& ex) {
  std::cerr << "ERR:” << ex.what() << "\n";
}
```

SPU Communication

- two ring channels in either direction
- each channel transfers 16 byte at once
- maximum transfer 16kB
- primary transfer facility: DMA
- secondary communication: mailboxes and signals
- names always from SPU perspective

Mailbox: Simple Messages

- communicate 32 bit values only
- per SPE one inbox and one outbox
  - inbox: 4 elements
  - outbox: 1 element
- on SPE exceeding operations will block
- on PPE exceeding operations will not block

SPU Mailbox Operations

- declared in <spu_mfcio.h>
- blocking: spu_read_in_mbox()
- blocking: spu_write_out_mbox(value)
- non-blocking: spu_stat_*_mbox()

PPU Mailbox Operations

- declared in <libspe.h>
- all operations on the PPU are non-blocking
- spe_write_in_mbox(speid, value)
- spe_read_out_mbox(speid)
- spe_stat_*_mbox(speid)

SPU Mailbox Example

```
uint32_t value = ::spu_read_in_mbox();
printf("mbox value=%d\n", value);
for (int i = 2; i <= 4; ++i)
{
  ::spu_write_out_mbox(i * value);
  printf("wrote value=%d\n", i * value);
}
```

PPU Mailbox Example

```
spe_write_in_mbox(sid, value);
spe_write_in_mbox(sid, value);
cout << "stat="
     << spe_stat_in_mbox(sid) << "\n";
while (spe_stat_out_mbox(sid) < 1) { }
for (int i = 0; i < 5; ++i)
  cout << "out="
       << spe_read_out_mbox(sid) << "\n";
```

DMA Operations

- addresses 16 bytes aligned
  - local address within the SPU
  - effective address for the outside address
- size of the transfer
  - multiple of 16 bytes
  - best size is multiple of 128 bytes
- tag to identify and group transfers

DMA Transfer to SPU

```
enum { gtag = 1, ptag };

uint32_t src __attribute__((aligned(16)));

mfc_get(&src, argp, sizeof(src), gtag, 0, 0);
mfc_write_tag_mask(1 << gtag);
mfc_read_tag_status_all();
```

DMA Transfer from SPU

```
uint32_t to[8] __attribute__((aligned(16)));
for (int i = 0; i < 8; ++i)
  to[i] = src + i;

mfc_put(&to, envp, 8*sizeof(to), ptag, 0, 0);
mfc_write_tag_mask(1 << ptag);
mfc_read_tag_status_all();
```

DMA Fences & Barriers

- DMA is unordered in general
- DMA within a tag group can be ordered
  - fence: following existing requests
  - barrier: follow existing and preceed coming request
- read/write operations with fence/barrier

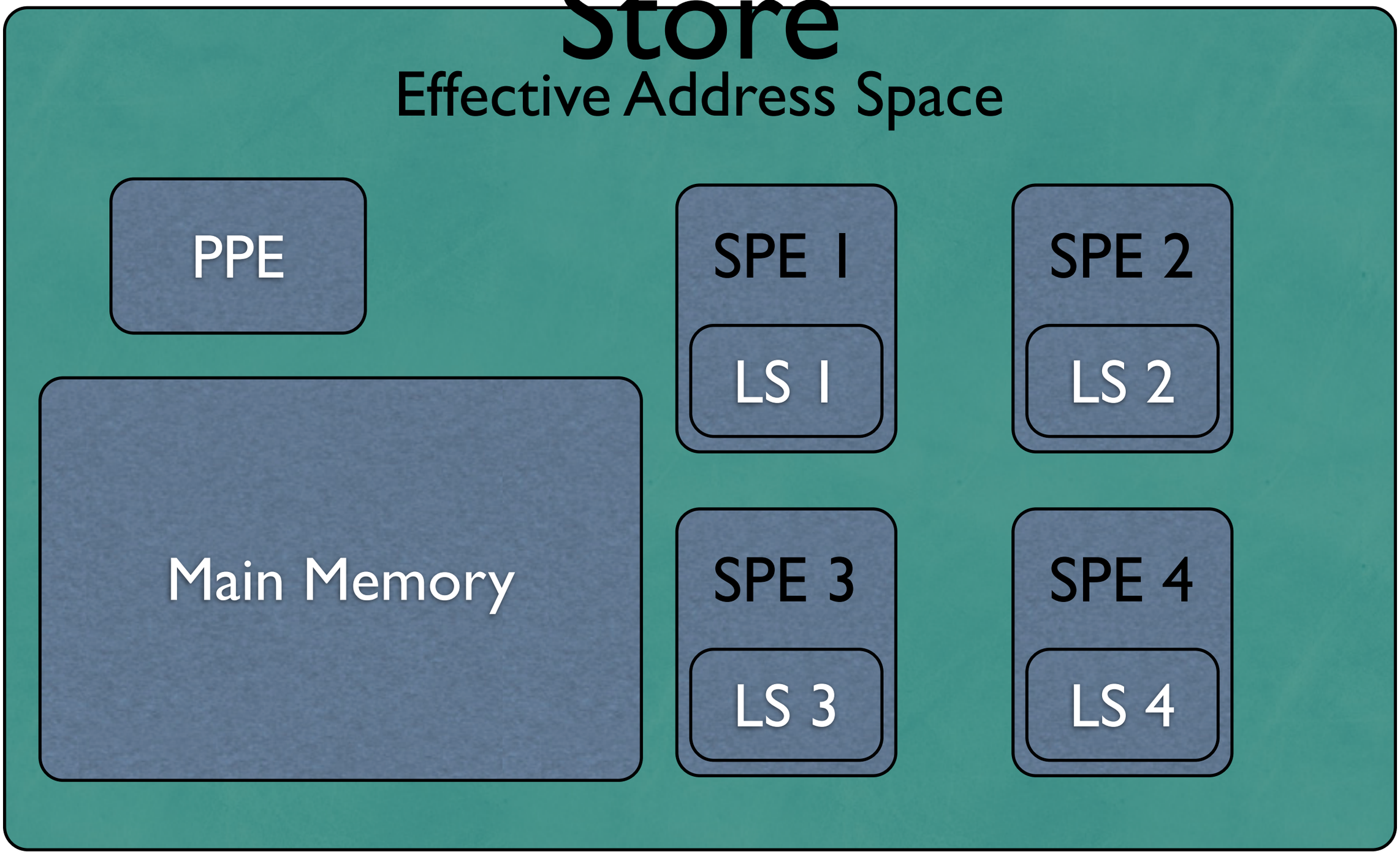
DMA Double-Buffering

```
get_data(data[0], tag);
for (int index(0); !done; ++index, tag ^= 1) {
  get_data(data[(index + 1) % 2], tag ^= 1);
  wait_for_tag(tag);
  process(data[index]);
  put_data(data[index], tag);
}
wait_for_tag(tag);
```

DMA for Scattered Data

- DMAed data on the local store is contigues
- DMAed data at the other end need not be:
  - a list can provide location of the pieces
  - each piece is still a multiple of 16 bytes
- used to get more complex data structures into the local store

DMA: EA vs. Local Store



More Parallelism: SIMD

- single (same) instruction, multiple data
- more computations with same logic
- on the SPU all operations are on vectors
- SIMD is available on most CPUs

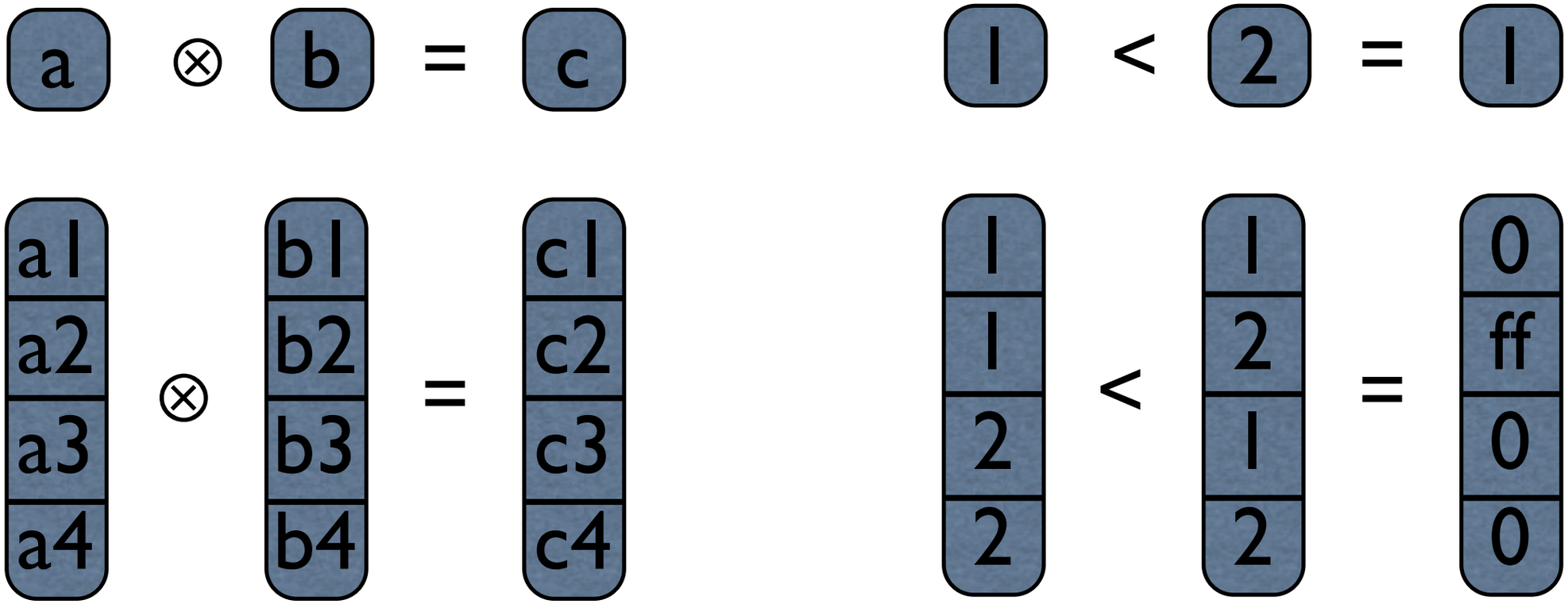
SIMD: Overview

- operations on built-in fixed sized vectors
- vector size on the Cell: 128 bit
  - 2 double
  - 4 int, float, ...
  - 8 short
  - 16 char

SIMD Operations

- load vector with scalars
- arithmetic: element-wise + - * / % ...
- comparison: element-wise < > == != ...
- shuffle vector elements
- operations to help with conditions

SIMD Operations



SIMD Types and Values

```
typedef float fv4 __attribute__((vector(16)));
typedef float fs4[4];
union ufv4 {
  ufv4(fv4 v): vec(v) {}
  ufv4(float f1, float f2, float f3, float f4):
    vec((fv4){f1, f2, f3, f4}) {}
  fv4 vec __attribute__((aligned(16)));
  fs4 val __attribute__((aligned(16)));
};
```

Scalar Example

```
int compute(int max, complex<float> c)
{
  int n(0);
  for (complex<float> z(c); n < max
     && z.magnitude() < 4; ++n)
    z = z * z + c;
  return n;
}
```

Vector Example

```
fv4 const zero(0), one(1), four(4);
uv4 rc = uv4(), cond(1);
for (complex<fv4> z(c); cond && max--; )
{
  z = z * z + c;
  cond = z.magnitude() < four;
  rc += select(cond, zero, one);
}
```

SIMD vs. branches

- comparison creates a vector of results
- use “select” (vec_sel() or spu_sel())
  - similar to ternary operator:
    result = condition? value1 : value2
    result = select(condition, value1, value2)
  - evaluates both expressions
- select() removes branches from the loop

Data Dependency

- instructions are processed in a pipeline
- steps might need to wait for earlier results
- no branches ⇒ interleave multiple

  iterations
  that is loop-unrolling is easy
- branch prediction is even less of a problem

Nicer Programming

- obviously: read-made libraries
  - there are already libraries using the SPUs
- generic libraries: somewhat problematic
  - data structures not directly shared
  - PPU and SPU run different code

Idea for Generics: llvm

- do not encode parallelism into the compiler!
- compile to llvm and operate on that
- detect specific entry points
  - algorithms marshall between CPUs
  - create different translation units for CPUs

Conclusion

- two different CPUs: 1 PPU and 8 SPUs
- multi-processing without shared memory
- communication via a fast internal bus
- DMA to transfer lots of memory
- mailboxes for control messages