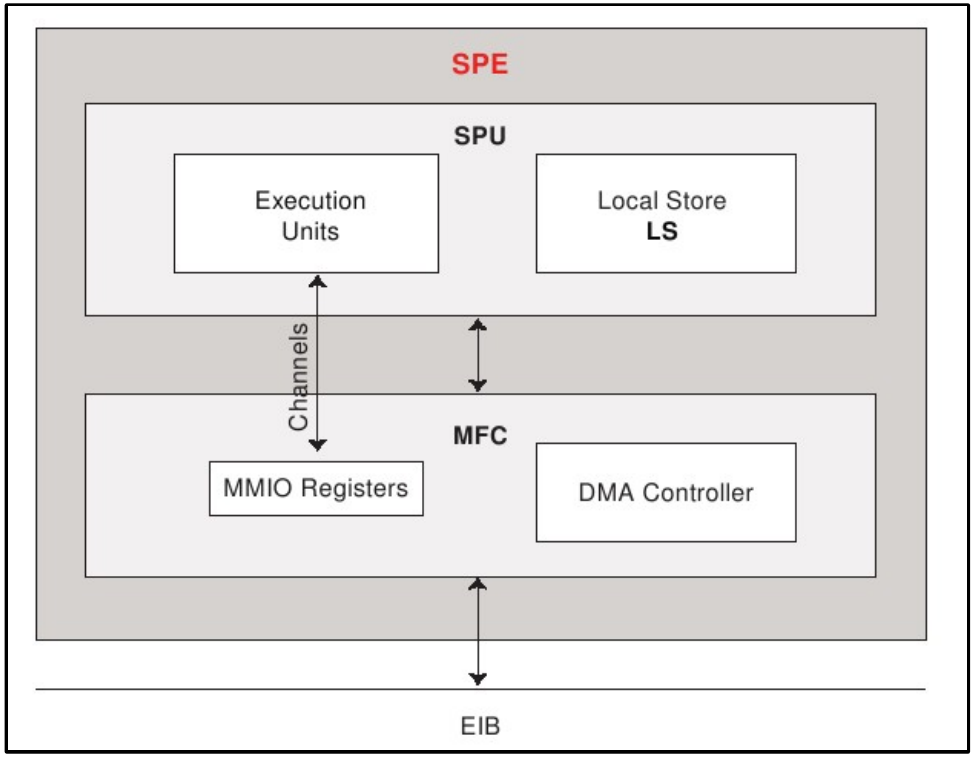
Programming the SPEs

quirin.n.meyer@stud.informatik.uni-erlangen.de

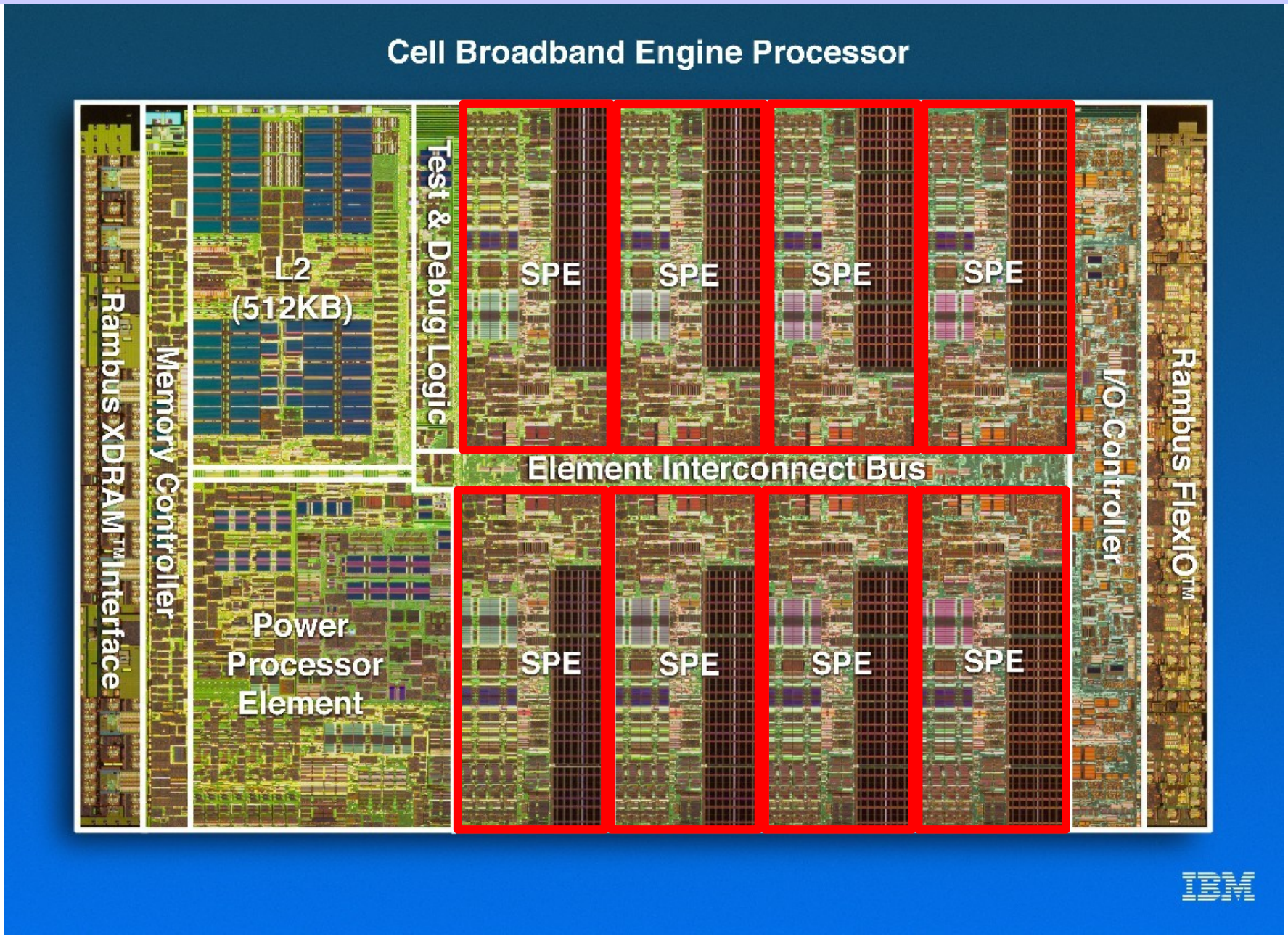
Outline

- Overview
- SPU
- MFC
- Coding Methods
- Miscellaneous



Taken from [0]

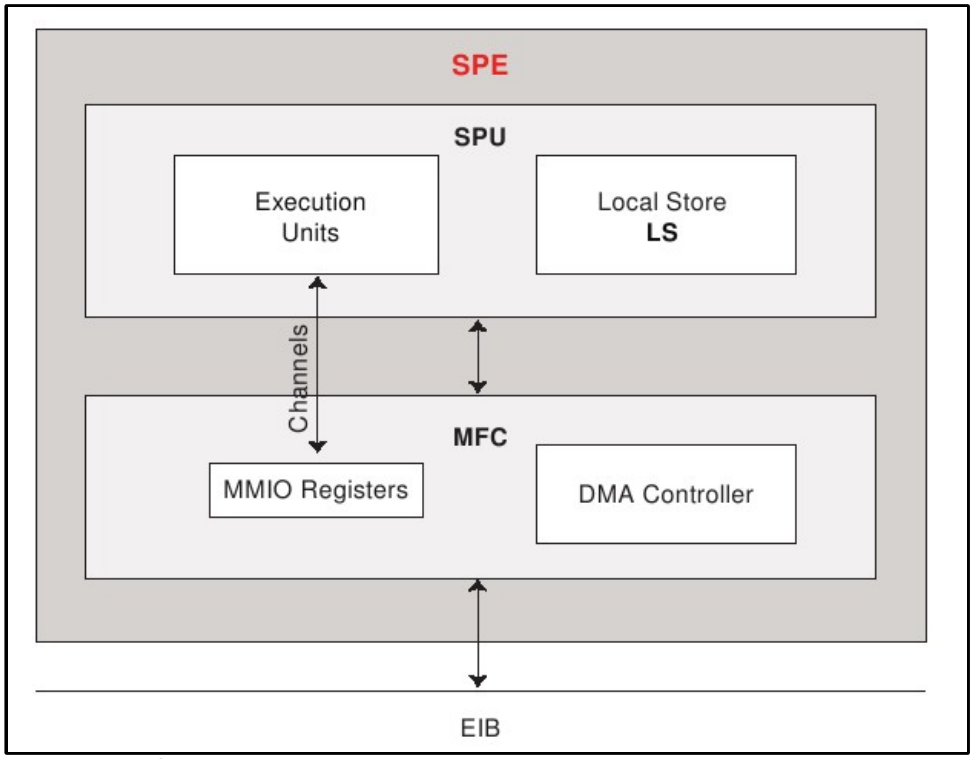
Location



Source: http://www.research.ibm.com/cell/cell_chip.html

SPE – Hardware Overview

- 8 SPEs “Synergistic Processing Element”
  - Synergistic Processing Unit (SPU)
    - Local Store (LS): 256kB
    - Register File: 128 x 128 Bit
  - Memory Flow Controller (MFC)
    - Memory Mapped IO (MMIO) Registers
    - DMA
  - Element Interconnect Bus
- Main control at PPU
- Number crunching tasks for SPUs



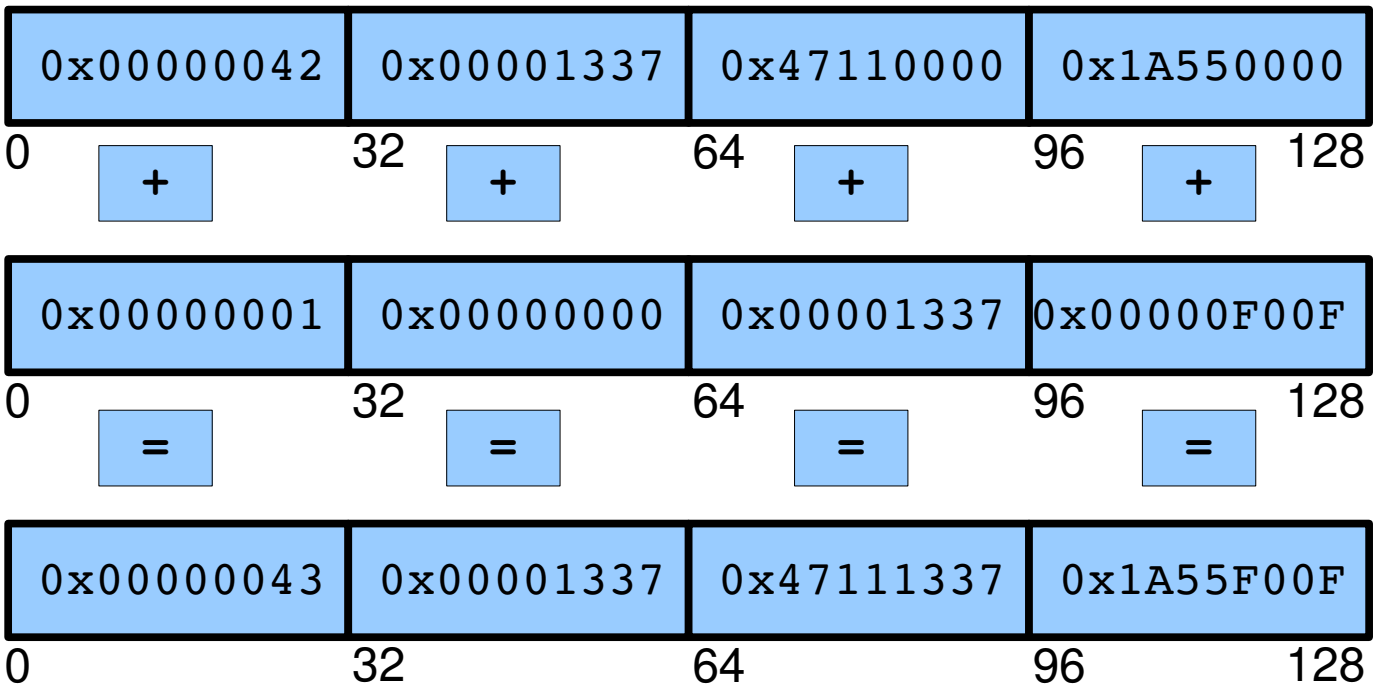
Taken from [0]

SPU

- Registers:
  - 128 Single Instruction Multiple Data GPRs
    - Float, Double (similar to IEEE 754)
      - SP way faster than DP (21.03 GFLOPS compared to 230.4 GFLOPS at 3.2Ghz) [1]
  - Floating-Point Status and Control Register
    - Compare result information, FP exceptions
  - No caches

SPU

- SIMD: single instruction multiple data
  - MD: Independent variables in one register
  - SI: One instruction performed on all variables



SPU

- SIMD
  - Register subdivision:
    - 8,16,32,64 and 128 Bits
  - Scalar operations
  - C/C++ data types:
    - `vector [[un]signed] {char | short | int | long | long long}`
    - `vector {float|double}`
    - `qword`
    - Fixed maximum number components in registers

SPU Intrinsics

- C/C++ built-in extensions for SPU Instructions
  - e.g.: `c = si_fa(a,b); /* FA d, a, b */`
- Mapped to one/more SPU assembly instruction
- Compiler (sometimes) performs
  - Register allocation
  - Instruction scheduling
  - Data loads and stores
  - Loop optimization
  - Correct up-stream placement of branch hints

SPU Intrinsics

- Specific intrinsics
  - 1-to-1 mapping of assembly instructions
- Generic intrinsics
  - `spu_add` maps to `si_a`, `si_ah`, `si_ai`, `si_fa`, `si_dfa`, ...
- Composite intrinsics
  - DMA transfers
- SIMD Instruction usage
  - Compiler
  - Assembly language
  - Intrinsics (`#include <spu_intrinsics.h>`)

SPU Intrinsics

- Intrinsic classes
  - Constant Formation
  - Conversion
  - Arithmetic
  - Byte Operation
  - Compare, Branch, Halt
  - Bit and Mask
  - Logical
  - Rotate
  - Shift
  - Control
  - Scalar
  - Channel Control
- Available on PPU but not on SPU
  - Saturating math
  - Sum-across
  - Log, Power
  - Ceil/Floor
  - Pixel vectors
- Available on SPU but not on PPU
  - Double word (SLOW)
  - Integer multiply and accumulate

SPU Example

- Squared Euclidean Norm of large vectors

SPE Code

```
float bigvec[VECTOR_SIZE];
float acc0 = 0;
for(j=0; j<VECTOR_SIZE; j++)
  acc0 += bigvecs[j] * bigvec[j];
```

- Does not make use of SIMD

SPE Code

```
vector float bigvec[VECTOR_SIZE/4];
vector float acc0 = spu_splats(0.0f);
for(j=0; j<VECTOR_SIZE/4; j++)
  acc0 = spu_madd(bigvec[j], bigvec[j], acc0);
float result =  _sum_across_float4(acc0);
```

- `VECTOR_SIZE` must be a multiple of 4
- Otherwise: Special treatment for remaining components

SPU

- Executes two instructions simultaneously
- **Even Pipeline:** SP, DP, FP integer, fixed point, rotate, shift, byte operations
- **Odd Pipeline:** Load and Store, Branch hints, Branch resolution, Channel interface, special purpose registers, Shuffle
- SPU starts issuing two operations in one cycle (CPI typically < 1)

SPU

- Instructions are issued in program order
- 2 Pipelines (Even and Odd)
  - SPU fetches 2 instructions from LS
  - SPU tries issuing both instructions
  - If dual issue is not possible:
    - Issue 1st instruction, hold 2nd
  - Next instruction fetch after both instruction are issued

SPU Example

- Use `gcc-spu -O3` flag to turn on all optimizations
- Euclidean norm on SPU:

SPE Code

```
vector float acc0 = spu_splats(0.0f);
for(j=0; j<VECTOR_SIZE; j++)
  acc0 = spu_madd(bigvec[j], bigvec[j], acc0);
float result =  _sum_across_float4(acc0));
```

- What `gcc-spu` does:

```
.L9:
        ai      $5,$5,-1
        lqx     $15,$4,$6
        lqx     $14,$4,$6
        ai      $4,$4,16
        nop     $127
        nop     $127
        fma     $8,$15,$14,$8
.L17:
        brnz    $5,.L9
```

Issued in parallel

But: only 25 % in parallel
25 % single issue
25 % Stalls
25 % Nops

SPU Example

- Better: Software pipelining [4]

SPE Code

```
vector float temp0 = bigvec[j];
vector float temp1 = bigvec[j+1];
vector float temp2 = bigvec[j+2];
vector float temp3;
for (; j<VECTOR_SIZE-4;j+=4) {
  acc0 = spu_madd(temp0, temp0, acc0);
  temp3 = bigvec[j+3];
  acc1 = spu_madd(temp1, temp1, acc1);
  temp0 = bigvec[j+4];
  acc2 = spu_madd(temp2, temp2, acc2);
  temp1 = bigvec[j+5];
  acc3 = spu_madd(temp3, temp3, acc3);
  temp2 = bigvec[j+6];
}
temp3 = bigvec[j+1];
acc0 = spu_madd(temp0, temp0, acc0);
acc1 = spu_madd(temp1, temp1, acc1);
acc2 = spu_madd(temp2, temp2, acc2);
acc3 = spu_madd(temp3, temp3, acc3);
```

Now: ~ 70 % in parallel
~ 28 % single issue
< 1 % Stalls
0 % NOPs

- Operations have latencies
- Parallel issue of FP and Load/Store instructions

SPU Summary

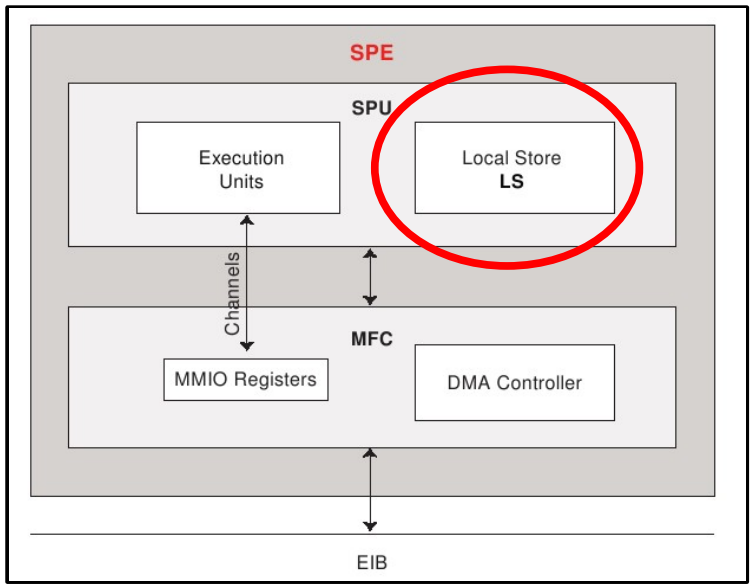
- All hand-crafted optimizations are possible with intrinsics
- Very close to assembly instructions
- Compiler optimizations supported up to a certain extinct
- Code is readable and easier to maintain
- No need to concern with register allocation

SPU - Supplement

- Code produced by IBM's `spuxlc` better than `spu-gcc`
- `spuxlc` performs automatic loop unrolling
  - Intrinsic Code gets unrolled eight times
  - Most simplest version gets unrolled 16 times (-O5)
- Hints
  - Generate assembler output: `make filename.s`
  - Change compiler in `make.env`

LS

- Instruction and Data
- Load/Store from/to the SPU
  - Constant latency of 6 cycles
  - 16-bytes-per-cycles
  - Align in LS to 16 Byte boundary

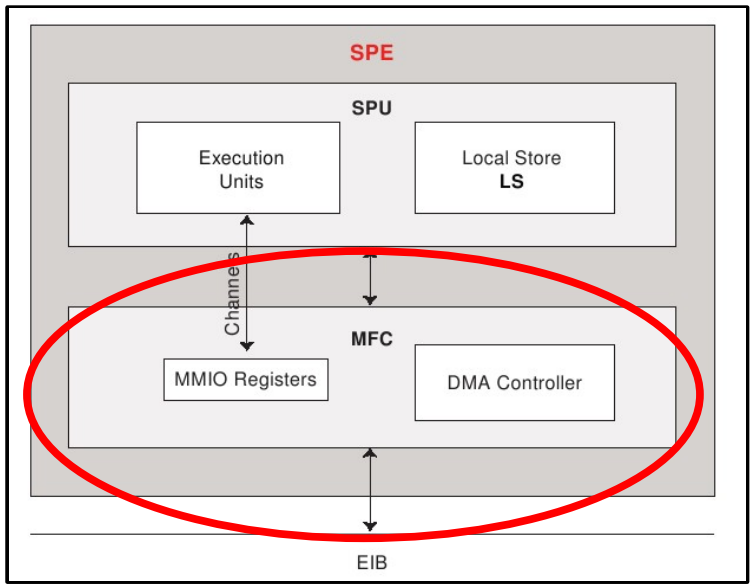


Taken from [0]

```
__attribute__ ((aligned (16)));
```

MFC

- Q: How to get data to LS?
- A: Use MFC transfers



Taken from [0]

Primary Function

- Connect SPU to EIB
- DMA transfers: LS <-> MFC

Access MFC

- From other SPEs/PPE: Memory Mapped IO registers
- From SPU: Channel Instructions

MFC

- DMA transfers [0], [1]
  - Bandwidth: 128-bytes-per-cycle
  - Sizes: 1,2,4,8 Bytes and N x 16B <= 16KB
  - SPU code continues while DMA fetches data
  - Best Performance:
    - 128 Byte alignment
    - Size: N x 256 Bytes
  - DMA list: 2K x 16KB transfers
- Two Queues (depending on the caller)
  - SPU: MFC SPU Command Queue
  - PPU/other SPEs: MFC Proxy Command Queue

MFC

- DMA transfer between MS and LS from local SPE
- Write necessary data to MFC channels, e.g.:

SPE Code

```
void dma_transfer(volatile void* lsa, unsigned int eah,
                  unsigned int eal, unsigned int size,
                  unsigned int tag_id, unsigned int cmd) {
  spu_writech(MFC_LSA,(unsigned int)lsa);
  spu_writech(MFC_EAH, eah);
  .
  .
  spu_writech(MFC_Cmd, cmd);
}
```

- Or use composite intrinsic

SPE Code

```
spu_mfcdma64(ls, eah, eal, size, tagid, cmd);
```

MFC

SPE Code

```
spu_mfcdma64(ls, eah, eal, size, tagid, cmd);
```

- ls: Address in local store
- eah, eal: Effective address
- Size: Max. 16 KB
- Tagid:
  - Values: 0..31
  - Helps ordering different DMA transfer
- Cmd: {GET|PUT}[l][b|f]
  - GET: MS to LS
  - PUT: LS to MS
  - l: List
  - f|b: synchronization

MFC

- DMA Example on SPE

SPE Code

```
/* select all groups to be included in query or wait operations*/
spu_writech(MFC_WrTagMask, -1);

/* start DMA command in order toget data */
spu_mfcdma32((void*)spearray, (unsigned int)ppearray, size, tag_id,
             MFC_GET_CMD);

/* wait for all dma transfer to be done    */
/* Possible parameters:                    */
/* 0 non-blocking                          */
/* 1 block for any commands to be complete */
/* 2 block for all commands to be complete */
spu_mfcstat(2);
```

MFC Signals Example

- Signal sending
  - Two 32 bit channels per SPE
- Example
  - PPE
    - Create threads
    - Send signal to all threads
  - SPEs
    - wait for signal
    - Print received 32 Bit variable

MFC Signals Example

PPE Code

```
extern spe_program_handle_t signal_spu;
int main() {
  int i;
  speid_t spe_ids[SPE_THREADS];
  for (i=0; i<SPE_THREADS; i++)
    spe_ids[i] = spe_create_thread(0, &signal_spu, NULL, NULL, -1, 0);
  for (i=0; i<SPE_THREADS; i++)
    spe_write_signal(spe_ids[i], SPE_SIG_NOTIFY_REG_1, i+42);
}
```

SPU Code

```
int main(unsigned long long spu_id, unsigned long long parm) {
  int ack;
  unsigned int signal;
  do {
    ack = spu_stat_signal1();
  } while (ack == 0);
  signal = spu_read_signal1();
  fprintf(stderr, "Recieved signal: %d\n", signal);
  return (0);
}
```

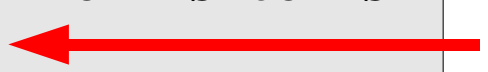
MFC - Mailboxes

- Used primarily for PPE-SPE communication
- Each SPU:
  - Inbound MB (4 Entries)
  - Outbound MB (1 Entry)
  - Interrupt handling
- Access from
  - SPU: Channel operations
  - PPU: Memory Mapped IO Registers

Coding methods – Double buffering

- SPE requires large data sets from MS
- Data divided into several Buffers B[i]

```
Foreach i
    Transfer B[i] from MS to LS
    Wait for B[i]
    Process B[i]
```



- Use double buffer scheme:

```
Transfer B[0]
Foreach i
    Transfer B[i]
    Wait for B[i-1]
    Process B[i-1]
Wait for completion of B[i-1]
Process[n]
```

Coding methods - Branches

- Branches disturb linear flow
  - Costs 20 cycles (regular instructions: 2-7 cycles)
- Means for reducing branches
  - Function-inlining
  - Loop-unrolling
  - Select-Bits Instruction
  - Branch Hints

Coding method - Branches

- Function inlining
  - No jump to function
  - No return
- Loop unrolling
  - Manually unrolling
  - Compiler automated (`spuxlc -qunroll[={yes|no|auto}]`)
  - Compiler directed (`#pragma unroll`)

Coding method - Branches

- Select Bits
  - Eliminates branches of simple control flow statements:

SPE Code

```
if (a > b)
  d += a;
else
  d += 1;
```

  - Calculate both results and select the right instead:

SPE Code

```
select = spu_cmpgt(a,b);
d1 = spu_add(d, a);
d2 = spu_add(d, 1);
d = spu_sel(d1, d2, select);
```

Coding methods - Branches

- Branch Hints

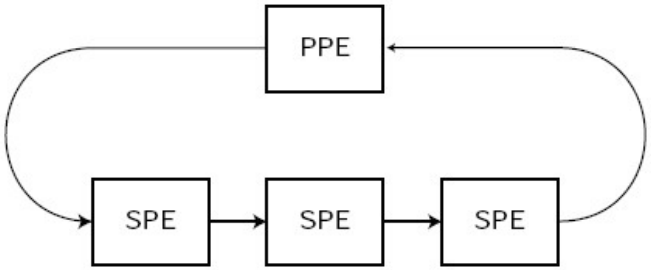
```
if(__builtin_expect(a > b, 0))
  c += a;
else
  d += 1;
```

- Types
  - Static Branch Prediction
    - Programmer's Knowledge
  - Dynamic Branch Prediction
    - Record branch history during runtime
- Branch hint is placed correctly in the resulting assembler code by compiler

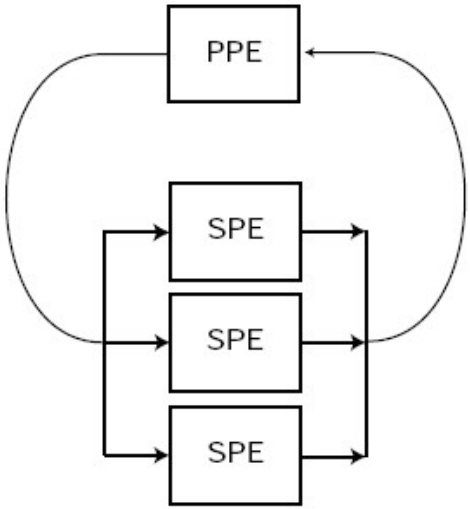
Application Partitioning

- PPE centric
  - PPE: Main Application
  - SPEs: Individual task

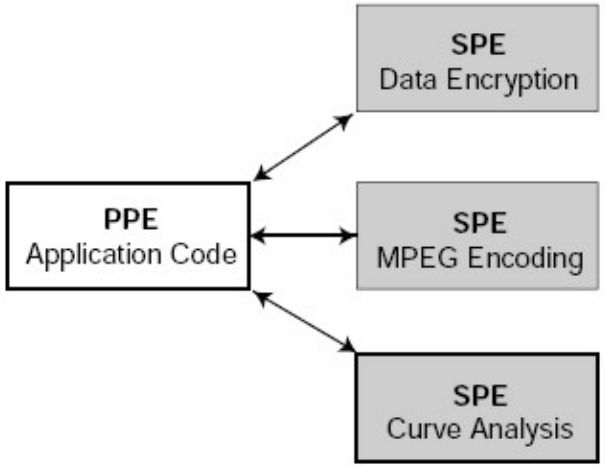
Multistage Pipeline



Parallel Stages



Services Model



SPE centric

PPE: centralized resource manager

SPEs: fetch work item; complete job; fetch next work item

Porting SIMD Code

- Suggested Developing
  - Write code on PPE
  - Port it to SPE
- Advantages:
  - No limited data size
  - No data movements
  - Better debug facilities
- Porting pitfalls
  - Different SIMD instruction sets
  - Different performance
  - Unmappable instructions
  - Limited size of LS
  - Precession [5]
- Porting strategy: Macro Trans.
  - Intrinsics that map 1-to-1
  - Data types
  - Vector-Literal Construction

Summary

- SPU
  - Single SPU Threads
  - SIMD
  - Simple optimization techniques
- MFC
  - DMA transfer
  - Mailboxes
  - Signals
- Coding Methods
- Porting code, Application Portioning

Discussion

What questions do you have?

References

[0] “Cell Broadband Engine Programming Tutorial”, Version 1.0, IBM, 2005

[1] Chen T., et. al. “Cell Broadband Engine Architecture and its first implementation”,IBM, 2005, http://www-128.ibm.com/developerworks/power/library/pa-cellperf/

[2] Srinivasan V., et. al. , “Cell Broadband Engine processor DMA engines”, IBM, 2005 http://www-128.ibm.com/developerworks/power/library/pa-celldmas/

[3] “SPE Runtime managment Library”, Version 1.1, IBM 2006

[4] Goedecker H., et. al. “Performance Optimization of Numerically Intensive Codes”, SIAM, 2001

[5] “SPU C/C++ Language Extensions”, IBM, 2005

[6] “Cell Broadband Engine Architecture”, IBM, 2005