Outline

- Program structure – PPE code and SPE code
- SIMD and vectorization
- Communication between processing elements – DMA, mailboxes
- Programming models
- Code example
PPE Code and SPE Code

- **PPE code – Linux processes**
  - a Linux process can initiate one or more “SPE threads”

- **SPE code – “local” SPE executables (“SPE threads”)**
  - SPE executables are packaged inside PPE executable files

- **An SPE thread:**
  - is initiated by a task running on the PPE
  - is associated with the initiating task on the PPE
  - runs asynchronously from initiating task
  - has a unique identifier known to both the SPE thread and the initiating task
  - completes at return from main in the SPE code

- **An SPE group:**
  - a collection of SPE threads that share scheduling attributes
  - there is a default group with default attributes
  - each SPE thread belongs to exactly one SPE group
Code Sample

- **PPE code:**

```c
#include <stdio.h>
#include <libspe.h>
extern spe_program_handle_t hello_spu;
int main(void)
{
    speid_t speid;
    int status;
    speid = spe_create_thread (0, &hello_spu, NULL, NULL, -1, 0);
    spe_wait(speid, &status, 1);
    return 0;
}
```

- **SPE code:**

```c
#include <stdio.h>
#include <cbe_mfc.h>
#include <spu_mfcio.h>
int main(speid_t speid, unsigned long long argp, unsigned long long envp)
{
    printf("Hello world!\n");
    return 0;
}
```

➤ (but “printf” from SPE only in the simulator environment …)
SIMD Architecture

- **SIMD = “single-instruction multiple-data”**
- **SIMD exploits data-level parallelism**
  - a single instruction can apply the same operation to multiple data elements in parallel
- **SIMD units employ “vector registers”**
  - each register holds multiple data elements
- **SIMD is pervasive in the BE**
  - PPE includes VMX (SIMD extensions to PPC architecture)
  - SPE is a native SIMD architecture (VMX-like)
- **SIMD in VMX and SPE**
  - 128bit-wide datapath
  - 128bit-wide registers
  - 4-wide fullwords, 8-wide halfwords, 16-wide bytes
  - SPE includes support for 2-wide doublewords
A SIMD Instruction Example

- **Example is a 4-wide add**
  - each of the 4 elements in reg VA is added to the corresponding element in reg VB
  - the 4 results are placed in the appropriate slots in reg VC
SIMD “Cross-Element” Instructions

- **VMX and SPE architectures include “cross-element” instructions**
  - shifts and rotates
  - permutes / shuffles

- **Permute / Shuffle**
  - selects bytes from two source registers and places selected bytes in a target register
  - byte selection and placement controlled by a “control vector” in a third source register
  ➢ extremely useful for reorganizing data in the vector register file
### Shuffle / Permute – A Simple Example

- **Bytes selected from regs VA and VB based on byte entries in control vector VC**
- **Control vector entries are indices of bytes in the 32-byte concatenation of VA and VB**
- **Operation is purely byte oriented**
  - SPE has extended forms of the shuffle / permute operation

<table>
<thead>
<tr>
<th>Reg VC</th>
<th>0 1 14 18 10 06 15 19 1a 1c 1c 13 08 1d 1b 0e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg VA</td>
<td>A.0 A.1 A.2 A.3 A.4 A.5 A.6 A.7 A.8 A.9 A.a A.b A.c A.d A.e A.f</td>
</tr>
<tr>
<td>Reg VB</td>
<td>B.0 B.1 B.2 B.3 B.4 B.5 B.6 B.7 B.8 B.9 B.a B.b B.c B.d B.e B.f</td>
</tr>
<tr>
<td>Reg VT</td>
<td>A.1 B.4 B.8 B.0 A.6 B.5 B.9 B.a B.c B.c B.c B.3 A.8 B.d B.b A.e</td>
</tr>
</tbody>
</table>
SIMD Programming

- **“Native SIMD” programming**
  - algorithm vectorized by the programmer
  - coding in high-level language (e.g. C, C++) using intrinsics
  - intrinsics provide access to SIMD assembler instructions
    - e.g. c = spu_add(a,b) → add vc,va,vb

- **“Traditional” programming**
  - algorithm coded “normally” in scalar form
  - compiler does auto-vectorization
    - but auto-vectorization capabilities remain limited
C/C++ Extensions to Support SIMD

- **Vector datatypes**
  - e.g. “vector float”, “vector signed short”, “vector unsigned int”, ...
  - SIMD width per datatype is implicit in vector datatype definition
  - vectors aligned on quadword (16B) boundaries
  - casts from one vector type to another in the usual way
  - casts between vector and scalar datatypes not permitted

- **Vector pointers**
  - e.g. “vector float *p”
  - p+1 points to the next vector (16B) after that pointed to by p
  - casts between scalar and vector pointer types

- **Access to SIMD instructions is via intrinsic functions**
  - similar intrinsics for both SPU and VMX
  - translation from function to instruction dependent on datatype of arguments
  - e.g. spu_add(a,b) can translate to a floating add, a signed or unsigned int add, a signed or unsigned short add, etc.
Vectorization

- For any given algorithm, vectorization can usually be applied in several different ways.
- Example: 4-dim. linear transformation (4x4 matrix times a 4-vector) in a 4-wide SIMD:

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix}
= 
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44} \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
\end{bmatrix}
\]

- Consider two possible approaches:
  - dot product: each row times the vector
  - sum of vectors: each column times a vector element

- Performance of different approaches can be VERY different.
Vectorization Example – Dot-Product Approach

- Assume:
  - each row of the matrix is in a vector register
  - the $x$-vector is in a vector register
  - the $y$-vector is placed in a vector register

- Process – for each element in the result vector:
  - multiply the row register by the $x$-vector register
  - perform vector reduction on the product (sum the 4 terms in the product register)
  - place the result of the reduction in the appropriate slot in the result vector register
Vectorization Example – Sum-of-Vectors Approach

- **Assume:**
  - each column of the matrix is in a vector register
  - the $x$-vector is in a vector register
  - the $y$-vector is placed in a vector register (initialized to zero)

- **Process – for each element in the input vector:**
  - copy the element into all four slots of a register (“splat”)
  - multiply the column register by the register with the “splatted” element and add to the result register
Vectorization Trade-offs

- Choice of vectorization technique will depend on many factors, including:
  - organization of data arrays
  - what is available in the instruction-set architecture
  - opportunities for instruction-level parallelism
  - opportunities for loop unrolling and software pipelining
  - nature of dependencies between operations
  - pipeline latencies
Communication Mechanisms

- **Mailboxes**
  - between PPE and SPEs

- **DMA**
  - between PPE and SPEs
  - between one SPE and another
Programming Models

- One focus is on how an application can be partitioned across the processing elements
  - PPE, SPEs

- Partitioning involves consideration of and trade-offs among:
  - processing load
  - program structure
  - data flow
  - data and code movement via DMA
  - loading of bus and bus attachments
  - desired performance

- Several models:
  - “PPE-centric” vs. “SPE-centric”
  - “data-serial” vs. “data-parallel”
  - others …
"PPE-Centric" & "SPE-Centric" Models

- **“PPE-Centric”:**
  - an offload model
  - main line application code runs in PPC core
  - individual functions extracted and offloaded to SPEs
  - SPUs wait to be given work by the PPC core

- **“SPE-Centric”:**
  - most of the application code distributed among SPEs
  - PPC core runs little more than a resource manager for the SPEs (e.g. maintaining in main memory control blocks with work lists for the SPEs)
  - SPE fetches next work item (what function to execute, pointer to data, etc.) from main memory (or its own memory) when it completes current work item
A Pipelined Approach

- Data-serial
- Example: three function groups, so three SPEs
- Dataflow is unidirectional
- Synchronization is important
  - time spent in each function group should be about the same
  - but may complicate tuning and optimization of code
- Main data movement is SPE-to-SPE
  - can be push or pull
A Data-Partitioned Approach

- Data-parallel
- Example: data blocks partitioned into three sub-blocks, so three SPEs
- May require coordination among SPEs between functions
  - e.g. if there is interaction between data sub-blocks
- Essentially all data movement is SPE-to main memory or main memory-to-SPE
Software Management of SPE Memory

- An SPE has load/store & instruction-fetch access only to its local store
  - Movement of data and code into and out of SPE local store is via DMA
- SPE local store is a limited resource
- SPE local store is (in general) explicitly managed by the programmer
Overlapping DMA and Computation

- DMA transactions see latency in addition to transfer time
  - e.g. SPE DMA get from main memory may see a 475-cycle latency

- Double (or multiple) buffering of data can hide DMA latencies under computation, e.g. the following is done simultaneously:
  - process current input buffer and write output to current output buffer in SPE LS
  - DMA next input buffer from main memory
  - DMA previous output buffer to main memory
  - requires blocking of inner loops

- Trade-offs because SPE LS is relatively small
  - double buffering consumes more LS
  - single buffering has a performance impact due to DMA latency
A Code Example – Complex Multiplication

- In general, the multiplication of two complex numbers is represented by

\[(a + ib)(c + id) = (ac - bd) + i(ad + bc)\]

- Or, in code form:

```c
/* Given two input arrays with interleaved real and imaginary parts */
float input1[2N], input2[2N], output[2N];

for (int i=0; i<N; i+=2) {
    float ac = input1[i]*input2[i];
    float bd = input1[i+1]*input2[i+1];
    output[i] = (ac - bd);
    /*optimized version of (ad+bc) to get rid of a multiply*/
    /* (a+b) * (c+d) a c - bd = ac + ad + bc + bd -ac b d = ad + bc */
    output[i+1] = (input1[i]+input1[i+1])*(input2[i]+input2[i+1]) - ac - bd;
}
```
Complex Multiplication SPE - Shuffle Vectors

Input

\[
\begin{array}{c|c|c}
I1 &=& \text{spu_shuffle}(A1, A2, I\_Perm\_Vector); \\
&\text{a1} & \text{b1} & \text{a2} & \text{b2} \\
&\text{c1} & \text{d1} & \text{c2} & \text{d2} \\
&\text{a3} & \text{b3} & \text{a4} & \text{b4} \\
&\text{c3} & \text{d3} & \text{c4} & \text{d4} \\
\end{array}
\]

Shuffle patterns

\[
\begin{array}{l|c|c|c|c}
I\_Perm\_Vector &=& \text{0-3} & \text{8-11} & \text{16-19} & \text{24-27} \\
&\text{a1} & \text{b1} & \text{a2} & \text{b2} \\
&\text{c1} & \text{d1} & \text{c2} & \text{d2} \\
&\text{a3} & \text{b3} & \text{a4} & \text{b4} \\
&\text{c3} & \text{d3} & \text{c4} & \text{d4} \\
\end{array}
\]

\[
\begin{array}{l|c|c|c|c|c|c|c|c}
Q\_Perm\_Vector &=& \text{4-7} & \text{12-15} & \text{20-23} & \text{28-31} \\
&\text{a1} & \text{b1} & \text{a2} & \text{b2} \\
&\text{c1} & \text{d1} & \text{c2} & \text{d2} \\
&\text{a3} & \text{b3} & \text{a4} & \text{b4} \\
&\text{c3} & \text{d3} & \text{c4} & \text{d4} \\
\end{array}
\]

\[
\begin{array}{l|c|c|c|c|c|c|c|c}
Q\_Perm\_Vector &=& \text{4-7} & \text{12-15} & \text{20-23} & \text{28-31} \\
&\text{a1} & \text{b1} & \text{a2} & \text{b2} \\
&\text{c1} & \text{d1} & \text{c2} & \text{d2} \\
&\text{a3} & \text{b3} & \text{a4} & \text{b4} \\
&\text{c3} & \text{d3} & \text{c4} & \text{d4} \\
\end{array}
\]

\[
\begin{array}{l|c|c|c|c|c|c|c|c}
Q\_Perm\_Vector &=& \text{4-7} & \text{12-15} & \text{20-23} & \text{28-31} \\
&\text{a1} & \text{b1} & \text{a2} & \text{b2} \\
&\text{c1} & \text{d1} & \text{c2} & \text{d2} \\
&\text{a3} & \text{b3} & \text{a4} & \text{b4} \\
&\text{c3} & \text{d3} & \text{c4} & \text{d4} \\
\end{array}
\]

I2 = spu_shuffle(B1, B2, I\_Perm\_Vector); By analogy

Q1 = spu_shuffle(A1, A2, Q\_Perm\_Vector);

Q2 = spu_shuffle(B1, B2, Q\_Perm\_Vector); By analogy
Complex Multiplication

A1 = spu_nmsub(Q1, Q2, v_zero);

A2 = spu_madd(Q1, I2, v_zero);

Q1 = spu_madd(I1, Q2, A2);

I1 = spu_madd(I1, I2, A1);
Complex Multiplication – Shuffle Back

D1 = spu_shuffle(I1, Q1, vcvmrgh);

D2 = spu_shuffle(I1, Q1, vcvmrgl);
Complex Multiplication – SPE - Summary

vector float A1, A2, B1, B2, I1, I2, Q1, Q2, D1, D2; /* in-phase (real), quadrature (imag), temp, and output vectors*/
vector float v_zero = (vector float)(0,0,0,0);

vector unsigned char I_Perm_Vector = (vector unsigned char)(0,1,2,3,8,9,10,11,16,17,18,19,24,25,26,27);
vector unsigned char Q_Perm_Vector = (vector unsigned char)(4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31);
vector unsigned char vcvmrgh = (vector unsigned char) (0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23);
vector unsigned char vcvmrgl = (vector unsigned char) (8,9,10,11,24,25,26,27,12,13,14,15,28,29,30,31);

/* input vectors are in interleaved form in A1,A2 and B1,B2 with each input vector representing 2 complex numbers
and thus this loop would repeat for N/4 iterations */

I1 = spu_shuffle(A1, A2, I_Perm_Vector); /* pulls out 1st and 3rd 4-byte element from vectors A1 and A2 */
I2 = spu_shuffle(B1, B2, I_Perm_Vector); /* pulls out 1st and 3rd 4-byte element from vectors B1 and B2 */
Q1 = spu_shuffle(A1, A2, Q_Perm_Vector); /* pulls out 2nd and 4th 4-byte element from vectors A1 and A2 */
Q2 = spu_shuffle(B1, B2, Q_Perm_Vector); /* pulls out 3rd and 4th 4-byte element from vectors B1 and B2 */
A1 = spu_nmsub(Q1, Q2, v_zero); /* calculates -(bd - 0) for all four elements */
A2 = spu_madd(Q1, I2, v_zero); /* calculates (bc + 0) for all four elements */
Q1 = spu_madd(I1, Q2, A2); /* calculates ad + bc for all four elements */
I1 = spu_madd(I1, I2, A1); /* calculates ac - bd for all four elements */
D1 = spu_shuffle(I1, Q1, vcvmrgh); /* spreads the results back into interleaved format */
D2 = spu_shuffle(I1, Q1, vcvmrgl); /* spreads the results back into interleaved format */
Complex Multiplication – code example

- **Example uses an “offload” model**
  - ‘complexmult’ function is extracted from PPC code to SPE

- **Overall code uses PPC to initiate operation**
  - PPC code tells SPE what to do (run ‘complexmult’) via mailbox
  - PPC code passes pointer to function’s arglist in main memory to SPE via mailbox
  - PPC waits for SPE to finish

- **SPE code**
  - fetches arglist from main memory via DMA
  - fetches input data via DMA and writes back output data via DMA
  - reports completion to PPC via mailbox

- **SPE operations are double-buffered**
  - $i^{th}$ DMA operation is started before main loop
  - $(i+1)^{th}$ DMA operation is started at beginning of main loop at secondary storage area
  - main loop then waits on $i^{th}$ DMA to complete

- **Notice that all storage areas are 128B aligned**
Complex Multiplication – SPE – Blocked Inner Loop

```c
for (jjo = 0; jjo < outerloopcount; jjo++)
{
    mfc_write_tag_mask (src_mask);
    mfc_read_tag_status_any ();
    mfc_get ((void *) a[(jo+1)&1], (unsigned int)input1, BYTES_PER_TRANSFER, src_tag, 0, 0);
    input1 += COMPLEX_ELEMENTS_PER_TRANSFER;
    mfc_get ((void *) b[(jo+1)&1], (unsigned int)input2, BYTES_PER_TRANSFER, src_tag, 0, 0);
    input2 += COMPLEX_ELEMENTS_PER_TRANSFER;
    voutput = ((vector float *) d1[jo&1]);
    vdata  = ((vector float *) a[jo&1]);
    vweight  = ((vector float *) b[jo&1]);
    ji = 0;
    for (jji = 0; jji < innerloopcount; jji+=2)
    {
        A1 = vdata[ji];
        A2 = vdata[ji+1];
        B1 = vweight[ji];
        B2 = vweight[ji+1];
        I1 = spu_shuffle(A1, A2, I_Perm_Vector);
        I2 = spu_shuffle(B1, B2, I_Perm_Vector);
        Q1 = spu_shuffle(A1, A2, Q_Perm_Vector);
        Q2 = spu_shuffle(B1, B2, Q_Perm_Vector);
        A1 = spu_nmsub(Q1, Q2, v_zero);
        A2 = spu_madd(Q1, I2, v_zero);
        Q1 = spu_madd(I1, Q2, A2);
        I1 = spu_madd(I1, I2, A1);
        D1 = spu_shuffle(I1, Q1, vcvmrgh);
        D2 = spu_shuffle(I1, Q1, vcvmrgl);
        voutput[ji] = D1;
        voutput[ji+1] = D2;
        ji += 2;
    }
    mfc_write_tag_mask (dest_mask);
    mfc_read_tag_status_all ();
    mfc_put ((void *)d1[jo&1], (unsigned int) output, BYTES_PER_TRANSFER, dest_tag, 0, 0);
    output += COMPLEX_ELEMENTS_PER_TRANSFER;
    jo++;
}
```
Links

- **Cell Broadband Engine resource center**

- **CBE forum at alphaWorks**