Developing Code for Cell - SIMD
Course Objectives

- Learn how to vectorize a scalar program to exploit the power of Cell BE
- How to use SPU intrinsics
Course Agenda

- **Vector Programming (SIMD)**
  - Data types for vector programming
  - Application partitioning
- **SPU Intrinsics**

**Trademarks** - Cell Broadband Engine™ is a trademark of Sony Computer Entertainment, Inc.
Vector Programming
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
Single Instruction Multiple Data (SIMD) Computation

Process multiple “b[i]+c[i]” data per operations

16-byte boundaries

R1

b0 b1 b2 b3

R2

c0 c1 c2 c3

R3

b0+c0 b1+c1 b2+c2 b3+c3

16-byte boundaries
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>vector regs</th>
<th>shuffle VT, VA, VB, VC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg VC</td>
<td>01 14 18 10 06 15 19 1a 1c 1c 13 08 1d 1b 0e</td>
</tr>
<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. \( c = \text{spu}\_\text{add}(a,b) \rightarrow \text{add vc,va,vb} \)

- "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**
  - vector [unsigned] {char, short, float, double}
    - e.g. “vector float”, “vector signed short”, “vector unsigned int”, …
  - SIMD width per datatype is implicit in vector datatype definition
  - casts from one vector type to another in the usual way
  - vectors aligned on quadword (16B) boundaries

- **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
Generic SPU Intrinsics

- Generic / Buit-In
  - Constant formation (spu_splats)
  - Conversion (spu_convtf, spu_convt, ..)
  - Arithmetic (spu_add, spu_madd, spu_nmadd, ...)
  - Byte operations (spu_absd, spu_avg, ...)
  - Compare and branch (spu_cmpeq, spu_cmpgt, ...)
  - Bits and masks (spu_shuffle, spu_sel, ...)
  - Logical (spu_and, spu_or, ...)
  - Shift and rotate (spu_rlqwbyte, spu_rlqw, ...)
  - Control (spu_stop, spu_ienable, spu_idisable, ...)
  - Channel Control (spu_readch, spu_writetch, ...)
  - Scalar (spu_insert, spu_extract, spu_promote)

- Composite
  - DMA (spu_mfcdma32, spu_mfcdma64, spu_mfcstat)
Fitting SIMD Data Types into Registers

<table>
<thead>
<tr>
<th>Preferred Slot</th>
<th>Preferred Scalar Slot:</th>
<th>Byte Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Byte</td>
<td>Halfword</td>
<td>Address</td>
</tr>
</tbody>
</table>

msb

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
<th>byte 4</th>
<th>byte 5</th>
<th>byte 6</th>
<th>byte 7</th>
<th>byte 8</th>
<th>byte 9</th>
<th>byte 10</th>
<th>byte 11</th>
<th>byte 12</th>
<th>byte 13</th>
<th>byte 14</th>
<th>byte 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 0</td>
<td>word 1</td>
<td>word 2</td>
<td>word 3</td>
<td>doubleword 0</td>
<td>doubleword 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halfword 0</td>
<td>halfword 1</td>
<td>halfword 2</td>
<td>halfword 3</td>
<td>halfword 4</td>
<td>halfword 5</td>
<td>halfword 6</td>
<td>halfword 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char 0</td>
<td>char 1</td>
<td>char 2</td>
<td>char 3</td>
<td>char 4</td>
<td>char 5</td>
<td>char 6</td>
<td>char 7</td>
<td>char 8</td>
<td>char 9</td>
<td>char 10</td>
<td>char 11</td>
<td>char 12</td>
<td>char 13</td>
<td>char 14</td>
<td>char 15</td>
</tr>
</tbody>
</table>

lsb
Accessing SIMD instructions

- **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.
  - Examples
    - t_v = spu_mul(t_v, (vector float)spu_splats((float)0.5));
    - t_v = spu_sub( (vector float) spu_splats( (float)47.11), t_v);
    - t_v = spu_mul( t_v, s_v);
Vectorizing a Loop – A Simple Example

- Loop does term-by-term multiply of two arrays
  - arrays assumed here to remain scalar outside of subroutine
- If arrays not quadword-aligned, extra work is necessary
- If array size is not a multiple of 4, extra work is necessary

```c
/* scalar version */
int mult1(float *in1, float *in2, float *out, int N)
{
    int i;
    for (i = 0, i < N, i++)
    {
        out[i] = in1[i] * in2[i];
    }
    return 0;
}

/* vectorized version */
int vmult1(float *in1, float *in2, float *out, int N)
{
    int i, Nv;
    Nv = N >> 2; /* N/4 vectors */
    vector float *vin1 = (vector float *) (in1);
    vector float *vin2 = (vector float *) (in2);
    vector float *vout = (vector float *) (out);
    for (i = 0, i < Nv, i++)
    {
        vout[i] = spu_mul(vin1[i],vin2[i]);
    }
    return 0;
}
```
Another SIMD Example – data incrementing

```c
void load_data(int *dest)
{
    int i;
    for (i=0; i<4096; ++i) {
        ++dest[i];
    }
}
```

```c
void load_data_SIMD(int *dest)
{
    int i;
    vector unsigned int *vdest;
    vector unsigned int v1 = (vector unsigned int) {1, 1, 1, 1};
    vdest = (vector unsigned int *) dest;
    for (i=0; i<1024; ++i) {
        vdest[i] = spu_add(vdest[i], v1);
    }
}
```
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
**SIMD Data Partitioning Strategies**

- **Choose SIMD strategy appropriate for algorithm**
  - **vec-across**
    
    ![Image](image)

    - 4 independent 4-component vectors (cyan, red, green and yellow)
    - Easy to SIMD – program as if scalar, operating on 4 independent objects at a time
    - Data must be maintained in separate arrays or SPU must shuffle vec-across data into a parallel array form
  - **parallel-array**

- **Consider unrolling affects when picking SIMD strategy**
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
Porting SIMD Code from the PPE to the SPEs

General approach

- Write the SIMD programs first for the PPE, then porting them to the SPEs.
- This approach postpones some SPE-related considerations of dealing with the local store (LS) size, data movements, and debug until after the port.
- The approach can also allow partitioning of the work into simpler (perhaps more digestible) steps on the SPEs.
- After the Vector/SIMD Multimedia Extension code is working properly on the PPE, a strategy for parallelizing the algorithm across multiple SPEs can be developed.
- Convert from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics, adding data-transfer and synchronization constructs, and tuning for performance.
- Test the impact of various techniques, such as DMA double buffering, loop unrolling, branch elimination, alternative intrinsics, number of SPEs, and so forth.
- Use debugging tools such as the static timing-analysis tool and the IBM Full System Simulator for the Cell Broadband Engine to assist this effort,

Alternatively, experienced Cell Broadband Engine programmers may prefer to skip the Vector/SIMD Multimedia Extension coding phase and go directly to SPU programming.

- In some cases, SIMD programming can be easier on an SPE than the PPE because of the SPE’s unified register file.
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) –ac – bd = ac + ad + bc + bd –ac –bd = ad + bc */
    output[i+1] = (input1[i]+input1[i+1])*(input2[i]+input2[i+1]) – ac – bd;
}
```
Complex Multiplication SPE - Shuffle Vectors

Input

Shuffle patterns

I1 = spu_shuffle(A1, A2, I_Perm_Vector);

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

\[ A1 = \text{spu}_n\text{msub}(Q1, Q2, v\_zero); \]

\[ A2 = \text{spu}_m\text{add}(Q1, I2, v\_zero); \]

\[ Q1 = \text{spu}_m\text{add}(I1, Q2, A2); \]

\[ I1 = \text{spu}_m\text{add}(I1, I2, A1); \]
Complex Multiplication – Shuffle Back

\[ D_1 = \text{spu\_shuffle}(I_1, Q_1, \text{vcvmrgh}); \]

\[ D_2 = \text{spu\_shuffle}(I_1, Q_1, \text{vcvmrgl}); \]
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 */
Special Notices -- Trademarks

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquires, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Many of the features described in this document are operating system dependent and may not be available on Linux. For more information, please check: http://www.ibm.com/systems/p/software/whitepapers/linux_overview.html

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

Revised January 19, 2006
Special Notices (Cont.) -- Trademarks


A full list of U.S. trademarks owned by IBM may be found at: http://www.ibm.com/legal/copytrade.shtml.

Cell Broadband Engine and Cell Broadband Engine Architecture are trademarks of Sony Computer Entertainment, Inc. in the United States, other countries, or both.
Rambus is a registered trademark of Rambus, Inc.
XDR and FlexIO are trademarks of Rambus, Inc.
UNIX is a registered trademark in the United States, other countries or both.
Linux is a trademark of Linus Torvalds in the United States, other countries or both.
Fedora is a trademark of Redhat, Inc.
Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.
Intel, Intel Xeon, Itanium and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.
AMD Opteron is a trademark of Advanced Micro Devices, Inc.
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.
TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).
SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECapc, SPEChpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corp (SPEC).
Altivec is a trademark of Freescale Semiconductor, Inc.
PCI-X and PCI Express are registered trademarks of PCI SIG.
InfiniBand™ is a trademark the InfiniBand® Trade Association
Other company, product and service names may be trademarks or service marks of others.

Revised July 23, 2006
Special Notices - Copyrights

(c) Copyright International Business Machines Corporation 2005.
All Rights Reserved. Printed in the United States September 2005.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.
IBM IBM Logo Power Architecture

Other company, product and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are
NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result
in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change
IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity
under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific
environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied
upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable
for damages arising directly or indirectly from any use of the information contained in this document.

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is
http://www.chips.ibm.com