Systems and Technology Group

Using the SDK Sample Programs

Course Code: L2T2H1-11
Cell Ecosystem Solutions Enablement

Class Objectives – Things you will learn

- **Examine the contents of the SDK in details, how it was organized, and the software provided in such directories as Samples, Tests, Tools, Workloads, and Application-oriented samples**
- **Analyze the complex multiplication code to understand how DMA and double buffering were used**
- **Learn some tips and techniques to better use the SDK**

Class Agenda

- **The Software Development Kit**
  - SDK contents
    - Samples
    - Tests
    - Tools
    - Workloads
    - Application-oriented code samples
- **A cell programming example**
  - SPE programming key points
  - Complex multiplication – code example
- **SDK tips and techniques**

The Software Development Kit

SDK Contents

- **The SDK source code is organized into the following categories:**
- **samples ($SDK_TOP/src/samples)**
  - simple and concise code examples to demonstrate specific functions, use of tools, libraries, and/or HW features
- **tests ($SDK_TOP/src/tests)**
  - self-verifying tests use to assure standards compliance, validate libraries and tools
- **tools ($SDK_TOP/src/tools)**
  - utilities used to generate content or ease programming burden
- **workloads ($SDK_TOP/src/workloads)**
  - code samples used to characterize the performance of the architecture
- **lib ($SDK_TOP/src/lib)**
  - libraries and reusable header files

Samples

- **cesof (CBE™ Embedded SPU Object Format)**
  - sample code to demonstrate the object format used to embed SPU objects into PowerPC binaries
- **DMA**
  - sample code to demonstrate non-trivial DMA calls
- **resample**
  - audio resampling code for SP/DP monotonic/stereo audio samples
- **simpleDMA**
- **spu_clean**
  - sample SPU program that clears the register file and local store (including itself)
- **spu_entry**
  - sample crt0 – initializes the stack and stack pointer; calls main; returns main's return value to a controlling PU program in an ABI compliant fashion (exit function).
- **spu_interrupt**
  - sample first level interrupt handler and second level interrupt handler registration function. Demonstration second level decrementer interrupt handler.
- **spulet**
  - C-library functions made to run on SPU (printf(), read(), etc.)
- **sync**
  - conditional wait, mutex, and atomic operation sample code
- **tutorial**
  - contains some of the source code used within the tutorial document

Tests

- **abi**
  - set of tests used to validate conformance to the SPU and BE ABI standards
- **asm**
  - set of tests used to verify assembler support of all instructions, parameter forms, and parameter ranges
- **events**
  - set of tests used to validate and demonstrate the handling of user-defined SPU events
- **intrinsics**
  - set of tests used to validate all VMX and SPU intrinsics
- **lib**
  - suite of self-validating tests used to verify correct operation of the libraries

Tools

- **callthru**
  - callthru source code
- **idl**
  - IDL compiler tool reads a high-level specification describing an interface to a SPU function
  - produces special stub functions to implement the interface in C
  - stubs allow the PU and SPU to communicate through what appear to be ordinary, local procedure calls or method invocations
- **oprofile (in progress)**
  - system-wide profiler for Linux
  - kernel dirver and daemon for collecting data
  - several post-profiling tools

Workloads

- **FFT16M**
  - hand-tuned program performing 4-way SIMD SP complex FFT of 16M elements
- **matrix_mul**
  - workload calculates C = A * B where A, B, and C are N x N squared matrices comprised of SP floats.
  - uses block-partitioning algorithm to reduce bandwidth (block size fixed to 64)
- **oscillator**
  - workload used to synthesize two stereo sound files
- **vse_subdiv**
  - workload demonstrating subdivision using contours of variable sharpness
  - displays result in OpenGL output window

Application-oriented Code Samples

- **C**
  - SPE-only library containing functions typically found in standard C99 library
  - includes functions executed by the SPE natively, functions initiated by the SPE but executed by the PPC, and SPE local store functions
  - provides or enhanced common high-level programming functionality
- **audio resample**
  - provides sample audio resampling functions that include
    - monophonic and stereophonic audio
    - unsigned short or FP samples
    - SP and DP computation
- **curves and surfaces**
  - support routines for evaluating quadratic and cubic Bezier curves as well as biquadric/bicubic Bezier surfaces and curved point-normal triangles
- **FFT**
  - highly tuned 1-D FFT as well as base kernel functions that can be used to implement 2-D FFTs

Application-oriented Code Samples (cont.)

- **game math**
  - set of routines implemented with the notion that precision and mathematical accuracy can at times be sacrificed for performance
- **image**
  - includes routines for various size convolutions as well as generation of histograms of byte data
- **large matrix**
  - various utility functions that operate on large vectors/matrices of SP FP numbers
  - size of input vectors and matrices limited by SPE local storage size (no matrix partitioning)
- **math**
  - general purpose math routines tuned to exploit SIMD features
  - most only support SP
  - intended to mimic standard math library functions

Application-oriented Code Samples (cont.)

- **matrix**
  - utility library to operate on matrices and quaternions including inversion, identity, perspective projection, and multiplication
- **misc**
  - routines that do not logically fit into other categories (min, max, rand, clamp, etc.)
- **multi-precision math**
  - performs mathematical functions on unsigned integers of a large number of bits
- **noise**
  - 1-D, 2-D, 3-D, and 4-D noise
  - lattice and non-lattice noise
  - turbulance

Application-oriented Code Samples (cont.)

- **oscillator**
  - two oscillator libraries to create a synthetic environment of configurable directional microphones, a large number of oscillators moving along defined paths, all relative to static microphones
  - computes time delays, volume changes, doppler effects
- **sim**
  - services useful to the full system simulator such as callthru
- **sync**
  - libraries making use of the load-with-reservation and store-conditional functions within CBEA
  - atomic operations, mutexes, conditional variables, and completion variables
  - sample code included in samples dir
- **vector**
  - 15 general purpose routines to operate on vectors

A Cell Programming Example

SPE Programming Key Points

- **SPE memory architecture**
  - each SPE has its own “flat” local memory
  - management of this memory is by explicit software control
  - code and data are moved into and out of this memory by DMA
  - programming the DMA is explicit in SPE code (or in PPC code)
- **Many DMA transactions can be “in flight” simultaneously**
  - e.g. each SPE can have 16 simultaneous outstanding DMA requests
  - a DMA request can be a list of DMA requests
  - DMA latencies can be hidden using multiple buffers and loop blocking in code
  - in contrast to traditional, hierarchical memory architectures that support few simultaneous memory transactions
- **Implications for programming the BE**
  - applications must be partitioned across the processing elements, taking into account the limited local memory available to each SPE

Complex Multiplication – code example

- **(See complex multiplication full code example)**
- **Overall code uses PPC to initiate operation as well as transmit data**
- **PPC waits for SPE to finish**
- **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

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

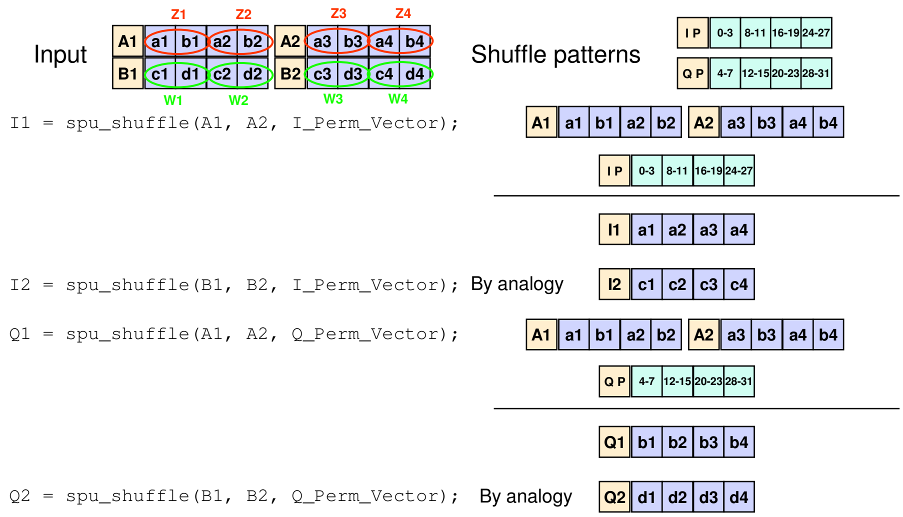
$$(a+ib)(c+id)=(ac-bd)+i(ad+bc)$$

- **Or, in code form:**

```
/* 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



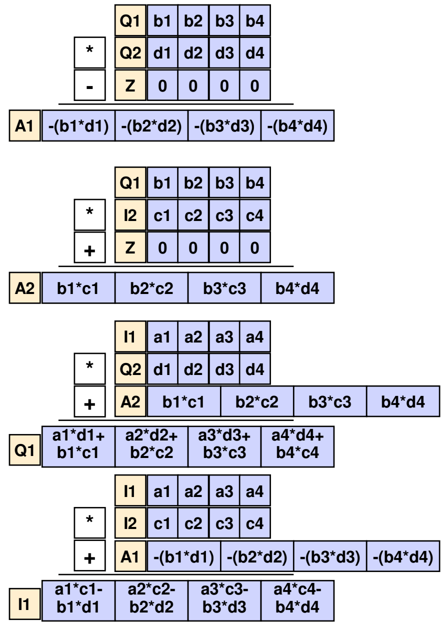
Complex Multiplication

```
A1 = spu_nmsub(Q1, Q2, v_zero);
```

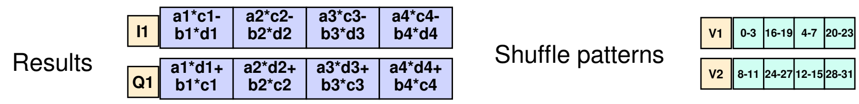
```
A2 = spu_madd(Q1, I2, v_zero);
```

```
Q1 = spu_madd(I1, Q2, A2);
```

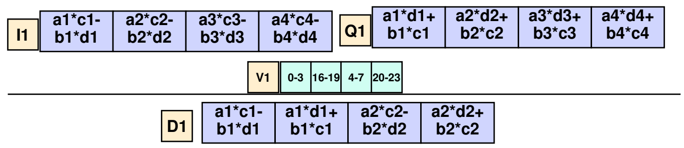
```
I1 = spu_madd(I1, I2, A1);
```



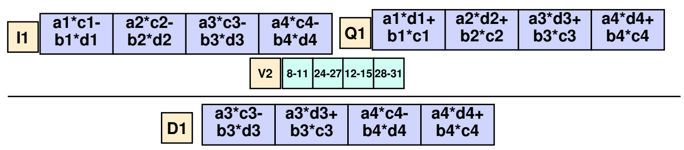
Complex Multiplication – Shuffle Back




```
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 */
```

SDK Tips and Techniques

- **$SDK_TOP/make.env contains environment variables to change compiler and compiler settings**
  - SPU_COMPILER
    - tells make to use gcc or xlc for SPU code
  - SPU_TIMING
    - if timing tools RPM is installed will generate a static timing analysis of SPU code to determine pipe stalls and register dependencies
  - SCE_VERSION
    - used to change the toolchain version employed
- **Several internal Systemsim parameters can be accessed and changed via the TCL/TK command line prompt**
  - use the HELP subsystem to access general information about these commands
- **Debugging tools are limited to GDB although there are both PPC and SPU versions**
  - PPC and SPU code must be debugged separately

SDK Tips and Techniques (cont.)

- **Profiling tools**
  - dynamic profiling
    - prof_start(), prof_stop(), prof_clear()
    - provides branch, pipe utilization, timing, issue, dependency, and other useful performance information during runtime
    - model must be in pipeline mode
  - static profiling
    - setting SPU_TIMING definition produces a .timing file containing a very useful static timing analysis for quickly determining code bottlenecks

Performance Measurement: Profile Checkpoints

NOTE: Execute application on simulator in pipeline mode

```
#include "profile.h"
prof_cp0(); // clear performance info
prof_cp30(); // start recording performance info
< something interesting >
prof_cp31(); // stop recording performance info
```

Generates

SPU0: CP31, 83(82), 4565

Instructions **Instructions** **Instructions**

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