Cell BE Porting Tips & Techniques

Cell Programming Workshop
Cell/Quasar Ecosystem & Solutions Enablement

Course Objectives

- To familiar with some common tips and techniques used in code porting

References

- Gordon Fossum, Quasar Design Center

Course Agenda

- Major steps in code porting
- Code samples
- Code details

Major Steps in Porting to Cell Broadband Engine

Get code running on PPU (easy port)

Deal with memory alignment concerns

Identify candidate code for porting to SPUs

Parallelize problem for SPE utilization

Establish communication methodology

Port scalar code to SPUs

Optimize data transfers (double-buffering)

SIMDize code and unroll loops

Code Samples

What is the problem?

- **Goals**
  - To show multiply-add operations,
  - Demonstrate how to handle arrays of different types and sizes,
  - Why it's useful to use "select" operations instead of if-then-else, whenever possible.
- **Problem statement**
  - Given coefficients for two polynomials, A(x) and B(x) and a big list of input x values, accompanied by a big list of flags to indicate which polynomial is to be used for each input. For convenience, we'll assume that the size of these big lists is a multiple of 16, as that will simplify some of the computations.
  - The coefficients are in order of decreasing exponent, so for example, if the degree of A(x) is 3, and A(x) is 5.0 * x^3 + 2.0 * x^2 + 8.0 * x + 12.0, then the array A_coeff would contain {5.0, 2.0, 8.0, 12.0}.
  - The output you're asked to produce is a list of computed polynomial values, as well as the sum of the A computations, the sum of the B computations, and the counts of how many of each there were.

What's in the code directory?

```
/opt/cell_class/Hands-on-30/polynomial-libspe2

clean          Phase0    Phase11   Phase4   Phase7   README
comp_run_all   Phase1    Phase2    Phase5   Phase8   runall
Makefile       Phase10   Phase3    Phase6   Phase9   runall.out
```

README contains a good overview of what we're doing.

The Phases show our progress through the conversion.

Coding Details

Basic code

```
void poly_compute(int poly_degree,  /* degree of the two polynomials   */
                  float *A_coeff,            /* array of (degree+1) floats,     */
                                             /* specifying the A polynomial     */
                  float *B_coeff,            /* array of (degree+1) floats,     */
                                             /* specifying the B polynomial     */
                  int array_size,            /* size of the next three arrays   */
                  float *x,                  /* the input values                */
                  float *y,                  /* the output values               */
                  char *AB_flag,             /* flags to indicate whether to    */
                                             /* compute A(x) (flag == 0) or     */
                                             /* compute B(x) (flag == 1)        */
                  int *counts,               /* how many A, how many B          */
                  double *sums)              /* array of two floats, to contain */
                                             /* the computed sums of the        */
                                             /* A(x) and B(x) values            */
{
int i, j;
  counts[0] = 0;
  counts[1] = 0;
  sums[0] = 0.0;
  sums[1] = 0.0;
  for (i=0; i<array_size; ++i) {
    float val;
    if (AB_flag[i]) {
      val = B_coeff[0];
      for (j=0; j<poly_degree; ++j) val = val * x[i] +
   B_coeff[j+1];
      ++counts[1];
      y[i] = val;
      sums[1] += (double) val;
    }
    else {
      val = A_coeff[0];
      for (j=0; j<poly_degree; ++j) val = val * x[i] +
   A_coeff[j+1];
      ++counts[0];
      y[i] = val;
      sums[0] += (double) val;
    }
  }
}
```

Coding phases

- Phase 0 is the basic code, wrapped inside code that exercises it.
- In Phase 1, we enforce alignment on the large arrays, to ensure efficient DMAs.
- In Phase 2, we split the problem into NSPUS pieces, where NSPUS is defined in a new .h file.
- In Phase 3, we create a "control block" to manage the parameters in a more compact fashion.
- In Phase 4, we move the code that solves the problem to the SPUs, leaving the wrapper code in the PPU.
- In Phase 5, we double-buffer the DMAs.
- In Phase 6, we start SIMDizing, by converting the large arrays into quadword arrays.
- In Phase 7, we prepare the internals for SIMDization by replacing the if test with selection statements.
- In Phase 8, we unroll some loops to prepare for conversion to vector variables.
- In Phase 9, we implement SIMDization.
- In Phase10, we implement loop unrolling.

Mapping the Steps to the Code Example

Get code running on PPU (easy port) (Phase 0)

Deal with memory alignment concerns (Phase 1)

Identify candidate code for porting to SPUs (Phase 2)

Parallelize problem for SPE utilization (Phase 2)

Establish communication methodology (Phase 3)

Port scalar code to SPUs (Phase 4)

Optimize data transfers (double-buffering) (Phase 5)

SIMDize code and unroll loops (Phases 6 – 10)

Deal with memory alignment concerns (Phase 1)

Main memory accesses are 128-byte aligned (both in MM and in LS)

SPU Local Store accesses are 16-byte aligned.

OLD:

```
float big_array[N];
float scalar1;
unsigned char scalar2;
```

NEW:

```
float big_array[N] __attribute__ ((aligned(128)));
float scalar1 __attribute__ ((aligned(16)));
unsigned char scalar2 __attribute__ ((aligned(16)));
```

(no need to worry about data that is only accessed by PPU)

Identify candidate code

Some people advocate using multi-core architectures in pipelined fashion, sending a job through a serial cascade of SPUs. We've not seen good results with that. Better to split a big job into symmetric pieces.

Each local store has 256 kbytes. We try to keep memory buffers in LS down to half of that, and let code, stack and heap contend for the rest of it.

If it all (code and text) fits in Local Store, use SPUlets! (Mark Nutter will discuss next week)

If you really want to put more into SPU than will statically fit, consider overlays.

Use "spu-size" to tell you how big your SPU binary is.

Gordon's Rule of thumb: 500 lines of regular scalar code should likely fit in 60K to 100K of Local Store, even after you expand it with SIMDization and loop unrolling.

Parallelize problem for SPE utilization

Start with

```
for (i=0; i<N; ++i) {
    compute something
}
```

Convert to

```
[compute starting points and lengths in the span of N]
for (i=0; i<NSPUS; ++i) {
    for (j=start[i]; j<start[i]+length[i]; ++j) {
        compute something
    }
}
```

Note that it’s important to ensure that `start[i]` corresponds to a cache line boundary for all associated variables.

Establish communication methodology (setup)

The PPU creates SPU threads and hands them an address.

Our typical method of communicating data is that this address is the location of a “Control Block” whose structure is known to both PPU and SPU.

The PPU loads control blocks for each SPU with data that is particular to that SPU, and then each SPU can read their control block to get the information they need.

Typically, the control block contains information about loop counts and addresses for locations of interest within arrays in main memory.

Establish communication methodology (synchronization)

Sometimes the SPU code contains two or more separate work items, and it's important to ensure that all SPUs finish a work item before any of them proceed to the next one.

This is a good time to use the mailbox facility.

The PPU plays traffic cop and "listens" for all SPUs to report that they are "done," and then the PPU blasts a message to all of them saying "you may now proceed to the next phase."

Optimize data transfers (double-buffering)

The major method of moving data around in this architecture is a DMA transfer.

While these can be initiated by the PPU, we've found that it's almost always better to let the SPUs move data between main memory and their own local stores.

Any DMA transfer no matter how small, is going to burn an entire 128-byte block of memory bandwidth, so it's encouraged to make most of your DMA transfers large.

Also, since DMA transfers take hundreds of cycles, it's very important to hide their latency by double-buffering.

SIMDize code

```
/* SCALAR */
for (i=0; i<n; ++i) a[i] = b[i] * c[i] + d[i];

/* SIMDized */
aV = (vector float *) a;
bV = (vector float *) b;
cV = (vector float *) c;
dV = (vector float *) d;
for (i=0; i<(n>>2); ++i)
    aV[i] = spu_madd(bV[i], cV[i], dV[i]);
```

Note: this only works when the arrays a, b, c, and d are aligned on quadword boundaries!

Unroll code

```
aV = (vector float *) a;
bV = (vector float *) b;
cV = (vector float *) c;
dV = (vector float *) d;

/* SIMDized */
for (i=0; i<(n>>2); ++i)
   aV[i] = spu_madd(bV[i], cV[i], dV[i]);

/* unrolled and SIMDized */
for (i=0; i<(n>>4); ++i) {
   aV[i+0] = spu_madd(bV[i+0], cV[i+0], dV[i+0]);
   aV[i+1] = spu_madd(bV[i+1], cV[i+1], dV[i+1]);
   aV[i+2] = spu_madd(bV[i+2], cV[i+2], dV[i+2]);
   aV[i+3] = spu_madd(bV[i+3], cV[i+3], dV[i+3]);
}
```

Notes on SIMDizing and unrolling

You have to manage loop variable when not a nice power of 2

Unrolling is especially useful when multiple statements in the loop are dependent (S2 depends on result in S1)

Often, it's best to load array elements into scalar variables separately, which makes code get larger, but speeds it up.

We have the luxury in the Cell Architecture of 128 registers in each SPU, meaning that unrolling actually works well!

We'll see examples in the code.

Code Walkthrough

/opt/cell_class/Hands-on-30/polynomial-libspe2

Special Notices -- Trademarks

Special Notices (Cont.) -- Trademarks

Special Notices - Copyrights

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