6.189 IAP 2007

Recitation 6

SIMD Programming on Cell
Agenda

- Overview of SIMD
- Vector Intrinsics on Cell
- SIMD Design Considerations
Most compute-bound applications are performing the same computations on a lot of data:
- Dependence between iterations is rare
- Opportunities for data parallelization across iterations and within iterations
Example: Scalar Operation

\[ C[0] = A[0] \times B[0] \]
Example: SIMD Vector Operation

for(i = 0; i < N/4; ++i)
    C[i] = vector_mul(A[i],B[i]);
Agenda

- Overview of SIMD
- Vector Intrinsics on Cell
- SIMD Design Considerations
Hardware Support for Data Parallelism

- Registers are 128-bits
- Can pack vectors of different data types into registers
- Operations consume and produce vector registers
  - Special assembly instructions
  - Access via C/C++ language extensions (intrinsics)
Vector Registers

- Only registers in SPU are 128-bit registers
  - Any type (including scalar types) can go into any register
- Scalar values go in a particular position in register

There is overhead associated with loading and storing scalars
Writing Efficient SIMD Code

- Used the `aligned` compiler directive to control placement
  - Quadword alignment for loads and stores (`aligned(16)`)  
- Transfer multiples of 16 bytes on loads and stores
  - Pad end of data if necessary
Vector Data Types

- Vector data types dictate how to interpret 128 bits
- Available on PPU and SPU:
  - 16x 8-bit int: `vector signed char`
  - 8x 16-bit int: `vector signed short`
  - 4x 32-bit int: `vector signed int`
  - 4x float: `vector float`
- Available on SPU:
  - 2x 64-bit int: `vector signed long long`
  - 2x double: `vector double`
- Pointer types, arrays, etc. work correctly
Vector Operations

- Compilers will insert vector instructions correctly for +, *, etc. when applied to vector types
- Intrinsics provide C/C++ access to vector instructions, including many which do not correspond to any operator
  - Example: `vector signed int c = spu_add(a, b);`
  - No need to worry about registers for operands
  - Looks like a function call
  - Compiler automatically generates instructions in assembly
  - Slightly different intrinsics available on PPU, SPU
Source Headers Necessary for Intrinsics

- **SPU intrinsics**
  - `#include <spu_intrinsics.h>`
  - `#include <spu_mfcio.h>`

- **PPU intrinsics**
  - `#include <ppu_intrinsicsics.h>`
  - `#include <vec_types.h>`
Initializing Vectors

- One of these cast notations should work (depending on your compiler):
  - `vector signed int a = (vector signed int)(10, 20, 30, 40);`
  - `... (vector signed int){10, 20, 30, 40};`

- Or use an intrinsic:
  - `vector signed int b = spu_splats(20);`
    // Same as (20, 20, 20, 20)
typedef union {
  int v[4];
  vector signed int vec;
} intVec;

Unpack scalars from vector:
- intVec a;
  a.vec = ...;
  ... = a.v[2];
- ... = spu_extract(va, 2);

Pack scalars into vector:
- a.v[0] = ...; a.v[1] = ...;
- ... = a.vec;

Interpret a segment of memory either as an array...

\[ v[0] \quad v[1] \quad v[2] \quad v[3] \]

... or as a vector type...

vec

so that values written in one format can be read in the other.
Vector Operations

- Integer instructions
- Floating-point instructions
- Permutation/formatting instructions
- Load and store instructions

Complete reference available from course web site
Vector Arithmetic and Logical Operations

- **PPU**
  - `vec_add, vec_sub, vec_madd`, ...
  - `vec_and, vec_or, vec_xor`, ...

- **SPU**
  - `spu_add, spu_sub, spu_madd, spu_mul, spu_re`, ...
  - `spu_and, spu_or, spu_xor`, ...

- Integer/FP operation associated with the correct vector types (char, int, float, etc.) is usually automatically selected by the compiler
Vector Shuffle Operation

- Rearrange bytes of vectors: $\text{spu\_shuffle}(A, B, \text{pattern})$
  - Each byte of the output is one of the bytes of $A$ or $B$
  - For each byte of output, corresponding byte of pattern specifies which byte of $A$ or $B$ to copy
    - Bit 4 of each pattern byte specifies $A$ or $B$
    - Bits 0-3 (4 low-order bits) of each pattern byte specify which byte (0-15) of source to take
      - Ex: 2$^{nd}$ byte of pattern is 0x14, so take byte 4 from $B$

$$VT = \text{spu\_shuffle}(VA, VB, VC)$$
Vector Shuffle Operation

● Generating the shuffle pattern:

```c
pattern =
    (vector unsigned char)( b0, b1, b2, b3,
                          b4, b5, b6, b7,
                          b8, b9,b10,b11,
                          b12,b13,b14,b15);
```

● Example: reverse the order of bytes in a

```c
a = spu_shuffle(a, a,
               (vector unsigned char)(15,14,13,12,
                             11,10, 9, 8,
                             7, 6, 5, 4,
                             3, 2, 1, 0);
```
Vector Rotate Operations

- Rotate shifts vector elements left or right
  - \texttt{spu\_rl(v, count)}
  - \texttt{vec\_rl(v, count)}
Review: \textit{sim} (Recitation 2)

- Simple 3D gravitational body simulator
- \( n \) objects, each with mass, initial position, initial velocity

```c
float mass[NUM_BODIES];
VEC3D pos[NUM_BODIES];
VEC3D vel[NUM_BODIES];
```

- Simulate motion using Euler integration
  - Calculate the force of each object on every other
  - Calculate net force on and acceleration of each object
  - Update position

```c
VEC3D d;
// Calculate displacement from \( i \) to \( j \)
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```
Re-engineering for SIMD

- One approach to SIMD: array of structs
  - Pad each \((x, y, z)\) vector to fill a quadword
  - Components \((x, y, z)\) correspond to first three words of a vector float
  - Quadwords for different vectors stored consecutively

```c
typedef union _VEC3D {
    struct {float x, y, z;};
    vector float vec;
} QWORD_ALIGNED VEC3D;
```
Now we can replace component-wise addition, subtraction, and multiplication with SIMD instructions:

```
VEC3D d;
// Calculate displacement from i to j
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```

```
vector float d;
// Calculate displacement from i to j
d = spu_sub(pos[j].vec, pos[i].vec);
```
Exercise 1 (15 minutes)

- Complete the SIMD implementation of sim
  - See example code in recitations section.
  - `tar xzf rec6.tar.gz`
  - `cd rec6/sim_aos`
  - `export CELL_TOP=/opt/ibm/cell-sdk/prototype`

- `spu/sim_spu.c`, line 49: implement `eltsumf4()`
  - Given a vector float \((a, b, c, d)\), return the vector float \((a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d)\)
  - You can do this with two shuffles and two adds
  - Note `vec_float4` is shorthand for `vector float`
  - Check your results with `./sim 1`
    - Will print “Verify succeeded” if your implementation is correct
Exercise 1

- Solution is in `sim_aos_soln`
- Sample implementation:

```c
vec_float4 b;

b = spu_shuffle(a, a,
          (vector unsigned char)(4, 5, 6, 7, 0, 1, 2, 3,
          12, 13, 14, 15, 8, 9, 10, 11));

a = spu_add(a, b);

b = spu_shuffle(a, a,
          (vector unsigned char)(8, 9, 10, 11, 12, 13, 14, 15,
          0, 1, 2, 3, 4, 5, 6, 7));

a = spu_add(a, b);

return a;
```
Agenda

- Overview of SIMD
- Vector Intrinsics on Cell
- SIMD Design Considerations
SIMD Design Considerations

- **Data layout: struct of arrays vs. array of structs**
  - Exercise 1 used an AOS layout
  - Alternatively we could use a SOA layout to lay out fields consecutively
  - Can apply different algorithms on new data layout

![Diagram comparing AOS and SOA layouts](attachment:image.png)
Struct of Array Layout

- Need 12 quadwords to store state for 8 objects
  - \(x, y, z\) position and velocity components
  - No padding component needed in SOA
- For each component, do four pair-interactions at once with SIMD instructions
  - Rotate quadword 3 more times to get all 16 pair-interactions between two quadwords
Performance Results Summary

- Example code in `rec6/sim_soa`
- 6144 objects, compiled with `-O2`
- Time per simulation step
  - SIMD array of structs: 300 ms
  - SIMD struct of arrays: 80 ms
Summary of Cell Optimizations That Were Covered

- Baseline native code was sequential and scalar
  - Scalar (PPU): 1510 ms \((\text{rec6/sim_spu, -O3})\)

- Parallelized code with double buffering for SPUs
  - Scalar (6 SPUs): 420 ms \((\text{rec6/sim_db})\)

- Applied SIMD optimizations
  - SIMD array of structs: 300 ms \((\text{rec6/sim_aos_soln})\)
  
- Redesigned algorithm to better suite SIMD parallelism
  - SIMD struct of arrays: 80 ms \((\text{rec6/sim_soa})\)

- Overall speedup compared to native sequential execution
  - Expected: \(~24\times (6 \text{ SPUs} \times 4 \text{ way SIMD})\)
  
  Achieved: 18x