CS4961 Parallel Programming

Lecture 6: SIMD Parallelism in SSE-3

Mary Hall
September 10, 2009

Administrative

- Programming assignment 1 to be posted Friday, Sept. 11
- Due, Tuesday, September 22 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 prog1 <gzipped tar file>"
- Mailing list set up: cs4961@list.eng.utah.edu
- Sriram office hours:
  - MEB 3115, Mondays and Wednesdays, 2-3 PM

Homework 2

Problem 1 (#10 in text on p. 111):

The Red/Blue computation simulates two interactive flows. An n x n board is initialized so cells have one of three colors: red, white, and blue, where white is empty, red moves right, and blue moves down. Colors wrap around on the opposite side when reaching the edge.

In the first half step of an iteration, any red color can move right one cell if the cell to the right is unoccupied (white). On the second half step, any blue color can move down one cell if the cell below it is unoccupied. The case where red vacates a cell (first half) and blue moves into it (second half) is okay.

Viewing the board as overlaid with t x t tiles (where t divides n evenly), the computation terminates if any tile's colored squares are more than c% one color. Use Peril-L to write a solution to the Red/Blue computation.

Homework 2, cont.

Problem 2:

For the following task graphs, determine the following:

(1) Maximum degree of concurrency.

(2) Critical path length.

(3) Maximum achievable speedup over one process assuming an arbitrarily large number of processes is available.

(4) The minimum number of processes needed to obtain the maximum possible speedup.

(5) The maximum achievable speedup if the number of processes is limited to (a) 2 and (b) 8.

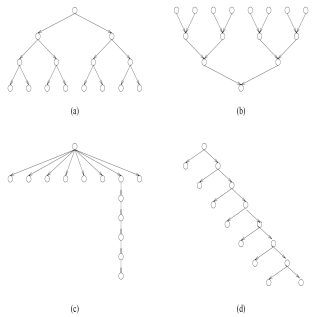


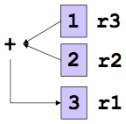
Figure 3.42 Task-dependency graphs for Problem 3.2.

Today's Lecture

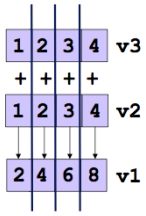
- SIMD for multimedia extensions (SSE-3 and Altivec)
- Sources for this lecture:
  - Jaewook Shin
    http://www-unix.mcs.anl.gov/~jaewook/slides/vectorization-uchicago.ppt
  - Some of the above from "Exploiting Superword Level Parallelism with Multimedia Instruction Sets", Larsen and Amarasinghe (PLDI 2000).
- References:
  - "Intel Compilers: Vectorization for Multimedia Extensions," http://www.aartbik.com/SSE/index.html
  - Programmer's guide
    http://developer.intel.com/design/processor/manuals/248966.pdf
  - List of SSE intrinsics
    http://developer.intel.com/design/pentiumii/manuals/243191.htm

Scalar vs. SIMD in Multimedia Extensions

```
Scalar: add r1,r2,r3
```

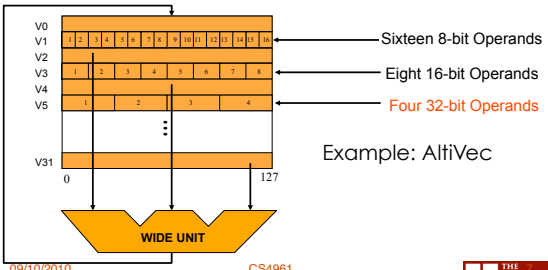



```
SIMD: vadd<sws> v1,v2,v3
```



Multimedia Extensions

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
    Vector length = register width / type size



Example: AltiVec

Multimedia / Scientific Applications

- Image
  - Graphics : 3D games, movies
  - Image recognition
  - Video encoding/decoding : JPEG, MPEG4
- Sound
  - Encoding/decoding: IP phone, MP3
  - Speech recognition
  - Digital signal processing: Cell phones
- Scientific applications
  - Double precision Matrix-Matrix multiplication (DGEMM)
  - Y[] = a*X[] + Y[] (SAXPY)

Characteristics of Multimedia Applications

- Regular data access pattern
  - Data items are contiguous in memory
- Short data types
  - 8, 16, 32 bits
- Data streaming through a series of processing stages
  - Some temporal reuse for such data streams
- Sometimes …
  - Many constants
  - Short iteration counts
  - Requires saturation arithmetic

Programming Multimedia Extensions

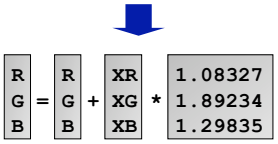
- Language extension
  - Programming interface similar to function call
  - C: built-in functions, Fortran: intrinsics
  - Most native compilers support their own multimedia extensions
    - GCC: -faltivec, -msse2
    - AltiVec: dst= vec_add(src1, src2);
    - SSE2: dst= _mm_add_ps(src1, src2);
    - BG/L: dst= __fpadd(src1, src2);
    - No Standard !
- Need automatic compilation

Programming Complexity Issues

- High level: Use compiler
  - may not always be successful
- Low level: Use intrinsics or inline assembly tedious and error prone
- Data must be aligned,and adjacent in memory
  - Unaligned data may produce incorrect results
  - May need to copy to get adjacency (overhead)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked

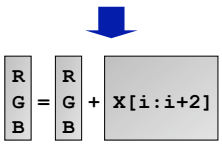
1. Independent ALU Ops

```
R = R + XR * 1.08327
G = G + XG * 1.89234
B = B + XB * 1.29835
```



2. Adjacent Memory References

```
R = R + X[i+0]
G = G + X[i+1]
B = B + X[i+2]
```



3. Vectorizable Loops

```
for (i=0; i<100; i+=1)
   A[i+0] = A[i+0] + B[i+0]
```


3. Vectorizable Loops

```
for (i=0; i<100; i+=4)
   A[i+0] = A[i+0] + B[i+0]
   A[i+1] = A[i+1] + B[i+1]
   A[i+2] = A[i+2] + B[i+2]
   A[i+3] = A[i+3] + B[i+3]
```

```
for (i=0; i<100; i+=4)
   A[i:i+3] = B[i:i+3] + C[i:i+3]
```


4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=1)
   L = A[i+0] - B[i+0]
   D = D + abs(L)
```

4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=2)
   L = A[i+0] - B[i+0]
   D = D + abs(L)
   L = A[i+1] - B[i+1]
   D = D + abs(L)
```

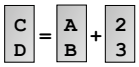
```
for (i=0; i<16; i+=2)
   L0 = A[i:i+1] - B[i:i+1]
   L1
   D = D + abs(L0)
   D = D + abs(L1)
```


Exploiting SLP with SIMD Execution

- Benefit:
  - Multiple ALU ops → One SIMD op
  - Multiple ld/st ops → One wide mem op
- Cost:
  - Packing and unpacking
  - Reshuffling within a register
  - Alignment overhead

Packing/Unpacking Costs

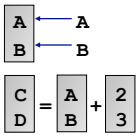
```
C = A + 2
D = B + 3
```



Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

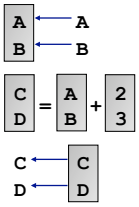
```
A = f()
B = g()
C = A + 2
D = B + 3
```



Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory
- Unpacking destination operands
  - Copying back to location

```
A = f()
B = g()
C = A + 2
D = B + 3
E = C / 5
F = D * 7
```



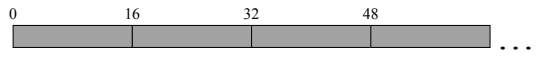
Alignment

- Most multimedia extensions require aligned memory accesses.
- Aligned memory access ?
  - A memory access is aligned to a 16 byte boundary if the address is a multiple of 16.
  - Ex) For 16 byte memory accesses in AltiVec, the last 4 bits of the address are ignored.

Alignment Code Generation

- Aligned memory access
  - The address is always a multiple of 16 bytes
  - Just one superword load or store instruction

```
float a[64];
for (i=0; i<64; i+=4)
   Va = a[i:i+3];
```

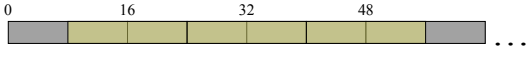


Alignment Code Generation (cont.)

- Misaligned memory access
  - The address is always a non-zero constant offset away from the 16 byte boundaries.
  - Static alignment: For a misaligned load, issue two adjacent aligned loads followed by a merge.

```
float a[64];
for (i=0; i<60; i+=4)
   Va = a[i+2:i+5];
```

```
float a[64];
for (i=0; i<60; i+=4)
   V1 = a[i:i+3];
   V2 = a[i+4:i+7];
   Va = merge(V1, V2, 8);
```



- Statically align loop iterations

```
float a[64];
for (i=0; i<60; i+=4)
  Va = a[i+2:i+5];

float a[64];
Sa2 = a[2]; Sa3 = a[3];
for (i=2; i<62; i+=4)
  Va = a[i+2:i+5];
```

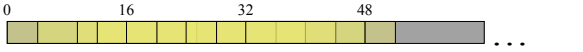

Alignment Code Generation (cont.)

- Unaligned memory access
  - The offset from 16 byte boundaries is varying or not enough information is available.
  - Dynamic alignment: The merging point is computed during run time.

```
float a[64];
for (i=0; i<60; i++)
   Va = a[i:i+3];
```

➡

```
float a[64];
for (i=0; i<60; i++)
   V1 = a[i:i+3];
   V2 = a[i+4:i+7];
   align = (&a[i:i+3])%16;
   Va = merge(V1, V2, align);
```



SIMD in the Presence of Control Flow

```
for (i=0; i<16; i++)
  if (a[i] != 0)
    b[i]++;
```

```
for (i=0; i<16; i+=4){
  pred = a[i:i+3] != (0, 0, 0, 0);
  old  = b[i:i+3];
  new  = old + (1, 1, 1, 1);
  b[i:i+3] = SELECT(old, new, pred);
}
```

Overhead:

Both control flow paths are always executed !

An Optimization: Branch-On-Superword-Condition-Code

```
for (i=0; i<16; i+=4){
  pred = a[i:i+3] != (0, 0, 0, 0);
  branch-on-none(pred) L1;
    old  = b[i:i+3];
    new  = old + (1, 1, 1, 1);
    b[i:i+3] = SELECT(old, new, pred);
  L1:
}
```

Control Flow

- Not likely to be supported in today's commercial compilers
  - Increases complexity of compiler
  - Potential for slowdown
  - Performance is dependent on input data
- Many are of the opinion that SIMD is not a good programming model when there is control flow.
- But speedups are possible!

Nuts and Bolts

- What does a piece of code really look like?

```
for (i=0; i<100; i+=4)
    A[i:i+3] = B[i:i+3] + C[i:i+3]
```

```
for (i=0; i<100; i+=4) {
    __m128 btmp = _mm_load_ps(float B[I]);
    __m128 ctmp = _mm_load_ps(float C[I]);
    __m128 atmp = _mm_add_ps(__m128 btmp, __m128 ctmp);
    void_mm_store_ps(float A[I], __m128 atmp);
}
```


Wouldn't you rather use a compiler?

- Intel compiler is pretty good
  - icc -msse3 -vecreport3 <file.c>
- Get feedback on why loops were not "vectorized"
- First programming assignment
  - Use compiler and rewrite code examples to improve vectorization
  - One example: write in low-level intrinsics

Summary of Lecture

- SIMD parallelism for multimedia extensions (SSE-3)
  - Widely available
  - Portable to AMD platforms, similar capability on other platforms
- Parallel execution of
  - "Vector" or superword operations
  - Memory accesses
  - Partially parallel computations
  - Mixes well with scalar instructions
- Performance issues to watch out for
  - Alignment of memory accesses
  - Overhead of packing operands
  - Control flow
- Next Time:
  - More SSE-3