CS4961 Parallel Programming

Lecture 10: Introduction to SIMD

Mary Hall
September 27, 2011

Homework 3, Due Thursday, Sept. 29 before class

- To submit your homework:
  - Submit a PDF file
  - Use the "handin" program on the CADE machines
  - Use the following command:
    "handin cs4961 hw3 <prob2file>"

Problem 1 (problem 5.8 in Wolfe, 1996):

(a) Find all the data dependence relations in the following loop, including dependence distances:

```
for (i=0; i<N; i+=2)
    for (j=i; j<N; j+=2)
        A[i][j] = (A[i-1][j-1] + A[i+2][j-1])/2;
```

(b) Draw the iteration space of the loop, labeling the iteration vectors, and include an edge for any loop-carried dependence relation.

Homework 3, cont.

Problem 2 (problem 9.19 in Wolfe, 1996):

Loop coalescing is a reordering transformation where a nested loop is combined into a single loop with a longer number of iterations. When coalescing two loops with a dependence relation with distance vector (d1,d2), whas is the distance vector for the dependence relation in the resulting loop?

Problem 3:

Provide pseudo-code for a locality-optimized version of the following code:

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i][j] = b[j][i] + c[i][j]*5.0;
```


Homework 3, cont.

Problem 4: In words, describe how you would optimize the following code for locality using loop transformations.

```
for (l=0; l<N; l++)
  for (k=0; k<N; k++) {
    C[k][l] = 0.0;
    for (j=0; j<W; j++)
        for (i=0; i<W; i++)
            C[k][l] += A[k+i][l+j]*B[i][j];
  }
```

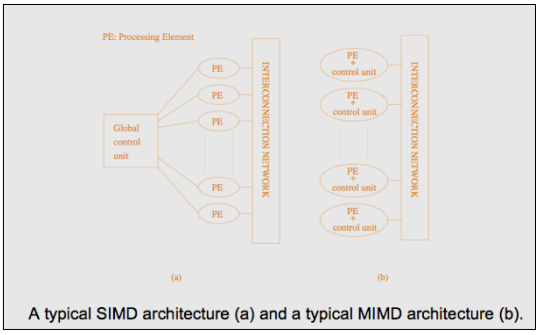
Today's Lecture

- Conceptual understanding of SIMD
- Practical understanding of SIMD in the context of multimedia extensions
- Slide source:
  - Sam Larsen, PLDI 2000, http://people.csail.mit.edu/slarsen/
  - Jaewook Shin, my former PhD student

Review: Predominant Parallel Control Mechanisms

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: `if ($myid == 0) { }`

SIMD and MIMD Architectures: What's the Difference?



A typical SIMD architecture (a) and a typical MIMD architecture (b).

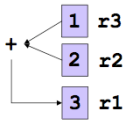
Slide source: Grama et al., Introduction to Parallel Computing, http://www.users.cs.umn.edu/~karypis/parbook

Overview of SIMD Programming

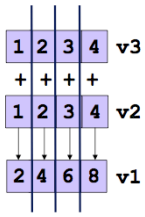
- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE and AltiVec
  - Graphics and games processors (CUDA, stay tuned)
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!)
  - GPUs are programmed by games developers (domain-specific libraries)

Scalar vs. SIMD in Multimedia Extensions

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



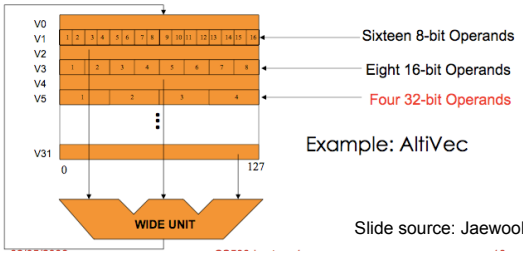

```
SIMD: vadd<sws> v1,v2,v3
```



Slide source: Sam Larsen

Multimedia Extension Architectures

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



Example: AltiVec

Slide source: Jaewook Shin

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
  - Array-based data-parallel computation, another level of parallelism

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

Why SIMD

+More parallelism

+When parallelism is abundant

+SIMD in addition to ILP

+Simple design

+Replicated functional units

+Small die area

+No heavily ported register files

+Die area: +MAX-2(HP): 0.1% +VIS(Sun): 3.0%

-Must be explicitly exposed to the hardware

-By the compiler or by the programmer

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
  - PhD student Jaewook Shin wrote a thesis on locality optimizations and control flow optimizations for SIMD multimedia extensions

Rest of Lecture

1. Understanding multimedia execution
2. Understanding the overheads
3. Understanding how to write code to deal with the overheads

Note:

A few people write very low-level code to access this capability.

Today the compilers do a pretty good job, at least on relatively simple codes.

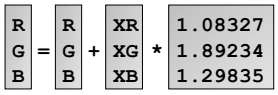
When we study CUDA, we will see a higher-level notation for SIMD execution on a different architecture.

Exploiting SLP with SIMD Execution

- Benefit:
  - Multiple ALU ops → One SIMD op
  - Multiple ld/st ops → One wide mem op
- What are the overheads:
  - Packing and unpacking:
    - rearrange data so that it is contiguous
  - Alignment overhead
    - Accessing data from the memory system so that it is aligned to a "superword" boundary
  - Control flow
    - Control flow may require executing all paths

1. Independent ALU Ops

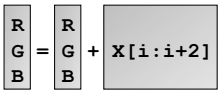
```
R = R + XR * 1.08327
G = G + XG * 1.89234
B = B + XB * 1.29835
```



Slide source: Sam Larsen

2. Adjacent Memory References

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



Slide source: Sam Larsen

3. Vectorizable Loops

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

Slide source: Sam Larsen

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]
```

Slide source: Sam Larsen

4. Partially Vectorizable Loops

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

Slide source: Sam Larsen

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
   L1  = A[i:i+1] - B[i:i+1]
   D = D + abs(L0)
   D = D + abs(L1)
```

Slide source: Sam Larsen

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

Packing/Unpacking Costs

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

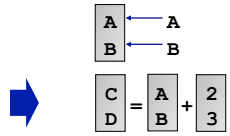
```
C     A     2
D  =  B  +  3
```

Slide source: Sam Larsen

Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

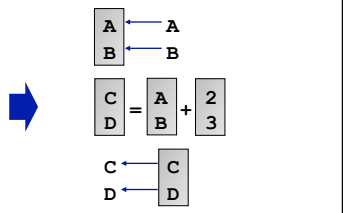
```
A = f()
B = g()
C = A + 2
D = B + 3
```



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
```

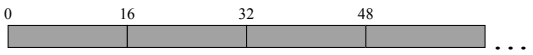


Slide source: Sam Larsen

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];
```

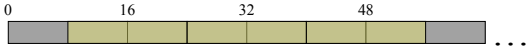


Alignment Code Generation (cont.)

- 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:i+3];
```


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];
start = read();
for (i=start; i<60; i++)
   Va = a[i:i+3];
```

```
float a[64];
start = read();
for (i=start; 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);
```