CS4961 Parallel Programming

Lecture 7: Introduction to SIMD

Mary Hall
September 14, 2010

Homework 2, Due Friday, Sept. 10, 11:59 PM

- To submit your homework:
  - Submit a PDF file
  - Use the "handin" program on the CADE machines
  - Use the following command:
    "handin cs4961 hw2 <prob2file>"

Problem 1 (based on #1 in text on p. 59):

Consider the Try2 algorithm for "count3s" from Figure 1.9 of p.19 of the text. Assume you have an input array of 1024 elements, 4 threads, and that the input data is evenly split among the four processors so that accesses to the input array are local and have unit cost. Assume there is an even distribution of appearances of 3 in the elements assigned to each thread which is a constant we call NTPT. What is a bound for the memory cost for a particular thread predicted by the CTA expressed in terms of λ and NTPT.

Homework 2, cont

Problem 2 (based on #2 in text on p. 59), cont.:

Now provide a bound for the memory cost for a particular thread predicted by CTA for the Try4 algorithm of Fig. 114 on p. 23 (or Try3 assuming each element is placed on a separate cache line).

Problem 3:

For these examples, how is algorithm selection impacted by the value of NTPT?

Problem 4 (in general, not specific to this problem):

How is algorithm selection impacted by the value of λ?

Programming Assignment 1
Due Wednesday, Sept. 21 at 11:59PM

- Logistics:
  - You'll use water.eng.utah.edu (a Sun Ultrasparc T2), for which all of you have accounts that match the userid and password of your CADE Linux account.
  - Compile using "cc –O3 –xopenmp p01.c"
- Write the prefix sum computation from HW1 in OpenMP using the test harness found on the website.
  - What is the parallel speedup of your code as reported by the test harness?
  - If your code does not speed up, you will need to adjust the parallelism granularity, the amount of work each processor does between synchronization points. You can adjust this by changing numbers of threads, and frequency of synchronization. You may also want to think about reducing the parallelism overhead, as the solutions we have discussed introduce a lot of overhead.
  - What happens when you try different numbers of threads or different schedules?

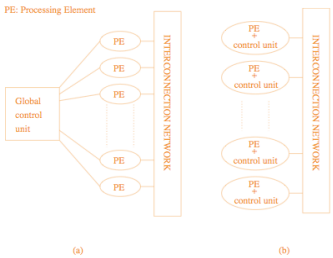
Programming Assignment 1, cont.

- What to turn in:
  - Your source code so we can see your solution
  - A README file that describes at least three variations on the implementation or parameters and the performance impact of those variations.
  - handin "cs4961 p1 <gzipped tarfile>"
- Lab hours:
  - Thursday afternoon and Tuesday afternoon

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?



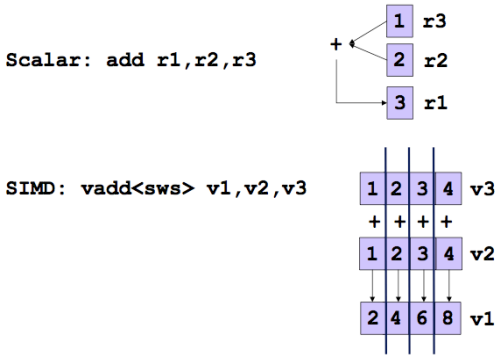
A typical SIMD architecture (a) and a typical MIMD architecture (b).

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

Overview of SIMD Programming

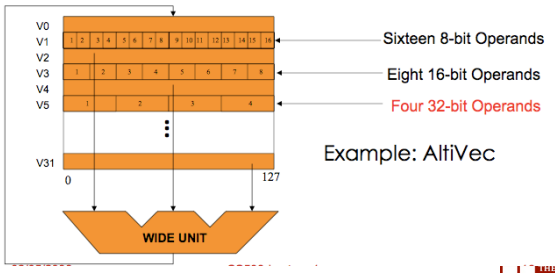
- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE and AltiVec
  - Graphics and games processors
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures 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



Multimedia Extension Architectures

- 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

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

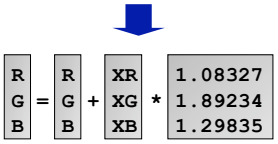
- Language extension
  - Programming interface similar to function call
  - C: built-in functions, Fortran: intrinsics
  - Most native compilers support their own multimedia extensions
    - GCC: -faltivec, -msse2
    - AltiVec: dst= vec_add(src1, src2);
    - SSE2: dst= _mm_add_ps(src1, src2);
    -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

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

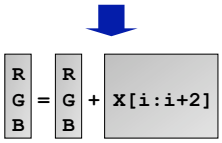
1. Independent ALU Ops

```
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)
```

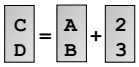
```
for (i=0; i<16; i+=2)
   L0 = A[i:i+1] - B[i:i+1]
   L1
   D = D + abs(L0)
   D = D + abs(L1)
```


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

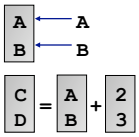
```
C = A + 2
D = B + 3
```



Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

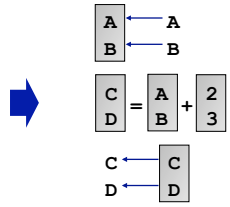
```
A = f()
B = g()
C = A + 2
D = B + 3
```



Packing/Unpacking Costs

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

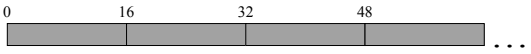
```
A = f()
B = g()
C = A + 2
D = B + 3
E = C / 5
F = D * 7
```



Alignment 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];
```

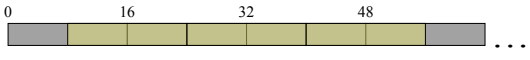


Alignment Code Generation (cont.)

- 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

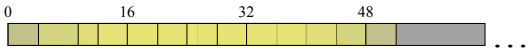
```
float a[64];
for (i=0; i<60; i+=4)
  Va = a[i+2:i+5];

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

Next Time

- Discuss Red-Blue computation, problem 10 on page 111 (not assigned, just to discuss)
- More on Data Parallel Algorithms