Agenda

• Mystery Processor
• Quiz 3
• Lab 3
• Lab 4
• Quiz 4
• VLIW/Vector Processors
  – unrolling, software pipelining, etc.
Stuff Graded So Far...

- PS 1, 2, 3, 4
- Lab 1
- Quiz 1, 2, 3
Quiz 3 Grades

- **Quiz Grade (out of 80)**
  - Average: 47 points (59%)
  - Stdev: 10.5 points (13%)

- **Question 1 (out of 30)**
  - Average: 21 points
  - Stdev: 4 points

- **Question 2 (out of 12)**
  - Average: 8 points
  - Stdev: 2.9 points

- **Question 3 (out of 9)**
  - Average: 6.2 points
  - Stdev: 3.1 points

- **Question 4 (out of 28)**
  - Average: 10.7 points
  - Stdev: 5.2 points
Quiz 3 Grades

- **Quiz Grade (out of 80)**
  - Average: 47 points (59%)
  - Stdev: 10.5 points (13%)

- **Question 1 common mistakes**
  - issuing more than 2 instructions in a single cycle
  - not bypassing data to the store (could issue the store earlier)

- **Question 4**
  - widening processor can allow more (miss-)speculative instructions into the pipeline, which can take up resources and cause cache evictions
  - sometimes widening a stage will not help if other resources are not getting freed faster too
Processor?
T0 Vector Processor

- 1995
- designed & implemented by Krste
- 45 MHz
- 730,701 transistors
- 280mm² die area
- 12 Watts
- 1.0 micrometer process
- 10 person-years from scratch
- 8 lanes, 2 FUs per lane
- MIPS-II RISC core (scalar control processor)
DEC Alpha 21264 (for comparison)

- 1996/1997
- single-core
  - 4-way
  - out-of-order
  - highly speculative
  - 7-stage
  - up to 80 instructions in flight
  - tournament branch predictor
- 15.2M transistors
  - 6M for logic
  - rest is caching, history tables
- 350 nm
- 600 MHz
- 64KB I$, 64KB D$ (on-chip)
  - 1 to 16MB L2$ (off-chip)
- 314mm² die (fairly large)
T0 Vector Processor

Figure 1: T0 Die Photograph.

Figure 2: T0 Block Diagram.

http://www.icsi.berkeley.edu/real/spert/t0esscc.pdf

Friday, April 6, 2012
Lab 3

- Still grading... (but looking good!)
- Some results...
C++ Branch Predictor

- Only one submission
  - Richard H, Brian L. & Brian S.
- “idealistic” category

Branch Accurancy

<table>
<thead>
<tr>
<th></th>
<th>gzip</th>
<th>vpr</th>
<th>gcc</th>
<th>mcf</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPKIs</td>
<td>10.40</td>
<td>9.32</td>
<td>5.43</td>
<td>11.65</td>
<td>9.20</td>
</tr>
<tr>
<td>Accuracy</td>
<td>99.0%</td>
<td>99.1%</td>
<td>99.5%</td>
<td>98.8%</td>
<td>99.1%</td>
</tr>
</tbody>
</table>
• 3 teams
• no predictor was the best at all benchmarks!
• all groups tried a tournament predictor, then iterated from there
Lab 4
Lab 4

- Program a vector-thread processor
  - cross between a GPU and a vector processor
  - micro-architecture loops like a vector processor
  - programmed like a GPU
- We will provide you a Chisel implementation
- Write vector complex-multiply
- Write vector matrix-matrix multiply
  - how much performance can you get? (Contest!)
Lab 4: Regular Vector Machine

Each “micro-thread” has its own set of registers and functional units.

Programmer model:

- Loop: vload
- vadd
- vshift
- vstore
- branch

Physical implementation:
- Map multiple micro-threads to a single lane.
– Regular scalar code runs on “Control Processor”
– Send a PC to the vector unit (“vector-fetch”)
  • all “micro-threads” in the vector unit fetch and decode their own instructions!
  • start execution at the PC given to it by the Control Processor
– can perform vector loads, vector stores
Lab 4: Programmer’s View

– VERY similar to programming GPU except:
  • can perform vector loads, vector stores
  • can factor out common code into the Control Processor

Assembly
Programmer's View

Control Processor

Vector of Virtual Processors

VP means “virtual processor”, but we will use the term “micro-thread”
Lab 4: Programmer’s View, VVadd

# scalar version

loop:
  flw  f2, 0(rA)
  flw  f3, 0(rB)
  fadd.s f2, f2, f3
  fsw  f2, 0(rC)
  addi rN, rN, -1
  addi rA, rA, 4
  addi rB, rB, 4
  addi rC, rC, 4
  bne  rN, zero, loop

done:
  ret

flw = floating point load word
fadd.s = floating point add (single precision)
fsw = floating point store word
Lab 4: Programmer’s View, VVadd

# vector thread version
la t2, vtcode          # load address of vtcode
vflw vf2, rA           # vector load
vflw vf3, rB           # vector load
vf 0(t2)               # jump to vector-fetch code
vfsw vf2, rC           # vector store
cpdone:
fence.v.l             # make stores visible to CP
ret

vtcode:
fadd.s f2, f2, f3     # add scalar
stop
Lab 4: Programmer’s View, VVadd

```
vvcfgivl t0, rN, 32, 32  # configure the vector unit
  # t0 holds the given vlen,
  # rN is the desired vlen (aka, “application vlen”)
  # num of x-regs, num of f-regs

now the strip mined version!

stripmineloop:
  vsetvl t0, rN,           # set vector length
  vflw vf2, rA
  vflw vf3, rB
  vf 0(t2)                # jump to vector-fetch code
  vfsw vf2, rC
  sub rN, rN, t0          # reduce N by the given vlen
  slli t1, t0, 2          # turn num_elements into num_bytes
  add rA, rA, t1
  add rB, rB, t1
  add rC, rC, t1
  bne rN, zero, stripmineloop

cpdone:
  fence.v.l
  ret

vtcode:
  fadd.s f2, f2, f3
  stop
```

Assembly
Programmer's View
Control Processor
Vector of Virtual Processors

Virtual Memory

Friday, April 6, 2012
Lab 4: Vector-thread

Assembly
Programmer's View

Control Processor
Vector of Virtual Processors

Virtual Memory

loop:
  vload
  vfetch foo
  vstore branch
  foo:
  add
  shift
  vp.stop

Implementation
Control Processor
Vector Lanes

Vector Issue Unit

Lane0
  VP0
  VP2

Lane1
  VP1
  VP3

Vector Mem Unit

Memory System

Energy per Operation

Multithreaded
Single Core
Vector-Thread
Vector

Performance

Friday, April 6, 2012
VECTOR-VECTOR ADD

- Characteristics
  - Purely data parallel
  - No conditionals
  - Integer Add

```c
for (int i=0; i < vlen; i++)
    c[i] = a[i] + b[i];
```

Vlen = 4

<table>
<thead>
<tr>
<th>VP0</th>
<th>VP1</th>
<th>VP2</th>
<th>VP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Friday, April 6, 2012
COMPLEX MULTIPLY

\[ r + si = (a+bi)(c+di) \]

- Characteristics
  - Purely data parallel
  - No conditionals
  - Floating Point Add, Multiply

```c
for (int i=0; i < vlen; i++) {
    r = ac - bd;
    s = bc + ad;
}
```

Friday, April 6, 2012
Lab 4 Hints

• # 1) fences
• # 2) CPE
• # 3) matmul
• # 4) vvcfgivl & vsetvl
• # 5) read all piazza posts on lab 4!
• # 6) yes, your vt code should be faster than scalar C code
• # 7) how to talk between CP, VU, element loads/stores
• # 8) think very carefully about which loop you vectorize in matrix-multiply!!!!
Fences

• you MUST do a fence.v.l in the “done” section of your vector assembly code (before the “ret” instruction).
• otherwise, CP may start trying to read the results array to verify the code before it has been written to the cache by the vector unit!
• I will not count your GLOP score if you do not fence the CP before returning from vec_matmul_asm
• fence.v.l stalls CP until vector-unit is drained of instructions
  – ensures all stores have been sent to the L1 cache (CP and VU share the same L1)
  – required when a micro-thread’s store needs to be visible to CP or another micro-thread
  – not required for a store to be visible to the same micro-thread
  – aka, ordering guaranteed within a single micro-thread
CPE

- cycles per element ("vector element")
- trying to measure how much time is spent executing each vt function per "vector element" (micro-thread)
- 64x64 matmul, uses 64x64x64 "vector elements"
vvcfgivl & vsetvl

• vvcfgivl rd, rs1, imm1, imm2
  – configures how many registers each vector element needs
  – allows vector unit to give you more elements! (longer vlen)
  – only call this once (is expensive)
• use “vsetvl rd, rs1” on every stripmine loop
  – sets vector length (rs1)
CP and micro-threads

– can only communicate through memory (must fence first!)

– however, you are probably doing Lab 4 wrong if you feel you need to do this (outside of using it for debugging)
micro-threads

- vector fetch functions can contain ANY scalar code
- micro-threads can perform their own scalar loads and stores
  - (lower performance though. do not recommend using these)

- can contain any scalar code.... in theory. Most stuff, like conditionals and function calls doesn’t work yet. Come back in a few months!
- notice we gave you code to write that doesn’t require conditionals...
How to get higher matmul performance?

• 1) memory behavior is **critical**
  – think hard about how you will access memory
  – want as few cache misses as possible
• 2) use the entire 16kB vector register file
  – e.g., use vector configure to increase **vector length**
• 3) stay in the vector register file for as long as possible
  – minimize loads/stores
Lab 4 questions?

Assembly
Programmer's View
Control Processor
Vector of Virtual Processors

Virtual Memory

Implementation
Control Processor
Vector Lanes

Energy per Operation
- Single Core
- Multithreaded
- Vector-Thread
- Vector

Memory System

Loop:
- vload
- vfetch foo
- vstore
- branch

Foo:
- add
- shift
- vp.stop

Friday, April 6, 2012
Quiz 4

- VLIW
  - able to write assembly for VLIW
- software
  - instruction re-ordering
  - loop unrolling
  - software pipelining
  - how code will get scheduled on different pipelines
  - conditional execution (for VLIW, vector, and GPU)
  - types of parallelism (ILP, TLP, DLP)
- Vector processors
  - able to write vector assembly (including how to strip-mine loops!)
  - chaining
- Multithreading
  - fine-grain, course-grain, SMT
- GPUs/SIMT model
  - how do they handle conditional execution/branches?
    - (spoiler alert: branch divergence)
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

**Note:** Iron Law questions about CPI are about counting the *instructions*, not the individual ops
Loop Unrolling

for (i=0; i<N; i++)

for (i=0; i<N; i+=4)
{
}

Unroll inner loop to perform 4 iterations at once

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop
Software Pipelining
Loop Execution

for (i=0; i<N; i++)

Loop Execution

Compile:

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Schedule:

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

March 14, 2011
CS152, Spring 2011

Friday, April 6, 2012
Software Pipelining

for (i=0; i<N; i++)

Compile

loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop

Schedule?

How does one do software pipelining?

Let’s run through an example that does software pipelining **WITHOUT** loop unrolling
Software Pipelining

for (i=0; i<N; i++)

Compile

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Compile

NOTE: this is a suboptimal solution

March 14, 2011

CS152, Spring 2011
Software Pipelining

NOTE: this is a suboptimal solution

for (i=0; i<N; i++)

Compile

How many FLOPS/cycle?
1 fadds / 4 cycles = 0.25
Software Pipelining

• We can do **better** though!
  – fadd takes 4 cycles, so if fadd is issued at cycle X, F₁ isn’t updated until X+4
  – thus, instructions that read F₁ in cycles X, X+1, X+2, and X+3 will get the OLD value of F₁
  – this can be exploited
• notice, this isn’t violating a RAW hazard, because we know exactly which cycle the write will occur
Software Pipelining (Optimal Solution)

for (i=0; i<N; i++)

Compile

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

How many FLOPS/cycle?
1 fadds / 3 cycles = 0.34
Software Pipelining

• Note: the code on the previous page assumes that B != A+4
  – in such a scenario, when B[i] is stored, A[i+1] then tries to load that memory location it would get the wrong value
  – this is because the LD occurs 2 cycles after the SD in the solution presented on the previous slide
  – in general, we will state the assumption that arrays do not overlap, preventing this issue
Pset 4, Question 4 (Vector Processors)
Pset 4, Question 4 (Vector Processors)
Problem 1: VLIW
Why is predicated execution useful for a VLIW system? Why doesn’t every superscalar do it as well?
Problem 2: Vector
Vector machines often have a lot of memory bandwidth (SX-9 has 256GB/s!). Why do they need it and why do current superscalars not provide as much?
Problem 3: Comparison
Please fill in the table on the next page. At a high level we will try to get a qualitative view of the different architectures we have discussed and some of the tradeoffs between them. This table sums up a great deal of content, so it will be approximate and gloss over many details.

- Hardware Features - Comment on what makes up this architecture, especially what structures or components it might have that the others don’t.
- Performance/Cost - To simply compare these architectures on performance would be unfair since with more or less resources they could be made better or worse. Of course this is highly workload dependent, so just consider each architecture at its peak performance.
- Ideal Workloads - What programs will achieve the best performance on this architecture.
- Adversarial Workloads - What workloads will achieve the worst performance on this architecture.
- Variance in Performance - How big is the gap between the best and worst performance.
- Flexibility - How many workloads achieve peak or near peak performance?
<table>
<thead>
<tr>
<th></th>
<th>Simple 5 Stage In-Order</th>
<th>Out-of-Order Superscalar</th>
<th>Vector</th>
<th>VLIW</th>
<th>Multithreaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance/Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ideal Workloads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adversarial Workloads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variance in Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>