Jonathan Bachrach

EECS UC Berkeley

September 2, 2014
Today

- Parlab / Aspire
- RISC-V + Rocket
- Iron Law + Optimizations
- Accelerators + RoCC
- — break —
- Decoupled Interfaces in Chisel
- RoCC Implementation in Chisel
- Towards General Purpose Accelerators
Got parallel computers but how do we write parallel software?

Principle Investigators: Krste Asanovic, Ras Bodik, Jim Demmel, Armando Fox, Tony Keavney, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, David Wessel, Kathy Yelick

Founding Companies: Intel and Microsoft
Got parallel computers but how do we write parallel software?
In a new general-purpose parallel language?
- An oxymoron?
- Won’t get adopted?
- Most big applications written in >1 languages

Par Lab bet on Patterns at all levels of programming
- Patterns provide a good vocabulary for domain experts
- Also comprehensible to efficiency-level experts or hardware architects
- Lingua franca between the different levels in ParLab
Patterns and Hardware Platforms

Only a few types of hardware platform

- Multicore
- GPU
- “Cloud”
Specializers: Pattern-specific and platform-specific compilers

*aka. “Stovepipes”*

Allow maximum efficiency and expressibility in specializers by avoiding mandatory intermediary layers
Algorithms and Specializers for Provably Optimal Implementations with Resiliency and Efficiency

http://aspire.eecs.berkeley.edu

Principle Investigators: Krste Asanovic (Director), Jonathan Bachrach, Armando Fox, Jim Demmel, Kurt Keutzer, Borivoje Nikolic, David Patterson, Koushik Sen, and John Wawrzynek
Future App Drivers

- Pervasive Speech
- Robotics
- Social Networks
- BIG DATA
- Environment
- Personalized Medicine
- Augmented Reality
performance = power \times \text{energy efficiency} \\
\left( \frac{\text{tasks}}{\text{second}} \right) = \left( \frac{\text{joules}}{\text{second}} \right) \times \left( \frac{\text{tasks}}{\text{joule}} \right)

- when power is constrained, need better energy efficiency for more performance
- where performance is constrained (real-time), want better energy efficiency to lower power

*Improving energy efficiency is critical goal for all future systems and workloads*
Good News: Moore’s Law Continues

More Transistors/Chip

Cheaper!

“Cramming more components onto integrated circuits”, Gordon E. Moore, Electronics, 1965
Bad News: Dennard Scaling Over

Data courtesy S. Borkar/Intel 2011
Keep computers’ performance and energy efficiency improving past end of CMOS transistor scaling until new switch technology deployed

Modern CMOS gives
- billions of transistors,
- reliably interconnected,
- clocking at GHz,
- for a few dollars

[Graph from “Advancing Computers without Technology Progress”, Hill, Kozyrakis, et al., DARPA ISAT 2012 ]
End of Sequential Processor Era

Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
use more, slower cores for better energy efficiency, either

- simpler cores
  - Limited by smallest sensible core

or

- run cores at lower Vdd/frequency
  - Limited by Vdd/Vt scaling, errors

Now what?
Dark Silicon

Opportunity: If only 10% die usable, build 10 different specialized engines and only use one at a time.

Exploitable Si
(in 45nm power budget)

[Source: ITRS 2008]

[Muller, ARM CTO, 2009]
Most computing happens in specialized, heterogeneous processors
- Can be 100-1000X more efficient than general-purpose processor

Challenges:
- Hardware design costs
- Software development costs

Nvidia Tegra2
As transistors become smaller and cheaper, communication dominates performance and energy.

All scales:
- Across chip
- Up and down memory hierarchy
- Chip-to-chip
- Board-to-board
- Rack-to-rack
1) Prove lower bounds on communication for a computation
2) Develop algorithm that achieves lower bound for system
3) Find that communication time/energy cost is >90% of resulting implementation
4) We know we’re within 10% of optimal!

Supporting technique: Optimizing software stack and compute engines to reduce compute costs and unavoidable communication costs
Intel Ivy Bridge (22nm)

Qualcomm Snapdragon MSM8960 (28nm)

- Future server and mobile SoCs will have many fixed-function accelerators and a general-purpose programmable multicore
- Well-known how to customize hardware engines for specific task
- ESP challenge is using specialized engines for general-purpose code.
• General-purpose hardware, flexible but inefficient
• Fixed-function hardware, efficient but inflexible
• ParLab Insight: Patterns capture common operations across many applications, each with unique communication and computation structure
• Build an ensemble of specialized engines, each individually optimized for particular pattern but collectively covering application needs
• Aspire Bet: ESP will give efficiency and flexibility
Optimize compute and data movement per pattern

- **Dense Engine**: Provide sub-matrix load/store operations, support in-register reuse

- **Structured Grid Engine**: Supports in-register operand reuse across neighborhood

- **Sparse Engine**: Support load/store of various sparse data structures

- **Graph Engine**: Provide load/store of bitmap vertex representations, support many outstanding request

- Richer semantics of new load/stores preserved throughout memory system for memory-side optimizations
RISC-V

Background
- Designed at Berkeley
- Fifth Berkeley RISC design

Advantages
- Open source with modified BSD license www.riscv.org
- Efficient to implement
- Extensible

State
- 2.0 Spec out
- Fast functional simulator
- GCC tool chain
- LLVM tool chain
- Boots linux
- lowRISC and india government support
31 General Purpose Integer Registers
Register to Register Operations
Load / Store with Addressing Modes
Control Transfer Operations
RV32 Instruction Encoding

- simple symmetric format
- easy and efficient to decode

integer instruction format

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
</tbody>
</table>

R-type

| imm[11:0] | rs1 | funct3 | rd | opcode |

I-type


S-type

| imm[31:12] | rd | opcode |

U-type

coprocessor instruction format

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>xd</td>
<td>xs1</td>
<td>xs2</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

roccinst[6:0] src2 src1 dest custom-0/1/2/3
```c
int a[64];
for (int i = 0; i < 64; i++)
    a[i] += 1;
```

```assembly
gcc -O3 -S ...

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>move x3,x0</td>
<td>x3 count</td>
</tr>
<tr>
<td>li x6,64</td>
<td>x6 lim</td>
</tr>
<tr>
<td>lw x5,0(x4)</td>
<td>x4 idx</td>
</tr>
<tr>
<td>addw x2,x3,1</td>
<td>inc count</td>
</tr>
<tr>
<td>move x3,x2</td>
<td></td>
</tr>
<tr>
<td>addw x5,x5,1</td>
<td>inc val</td>
</tr>
<tr>
<td>sw x5,0(x4)</td>
<td>update</td>
</tr>
<tr>
<td>add x4,x4,4</td>
<td>inc idx</td>
</tr>
<tr>
<td>bne x2,x6,$LOOP</td>
<td></td>
</tr>
</tbody>
</table>
```

7 cycles / element inc
gcc -O3 -S ...

```assembly
move    x3,x0
li      x6,64
$LOOP: lw    x5,0(x4)
addw   x2,x3,1
move   x3,x2
addw   x5,x5,1
sw     x5,0(x4)
add     x4,x4,4
bne     x2,x6,$LOOP
```

7 cycles / element inc

- -

optimized by hand

```assembly
lw    t0, a
lw    t1, a+64*4
$LOOP: lw    t2, 0(t0)
addw   t0, t0, 4 // inc idx
addw   t2, t2, 1 // inc val
sw     t2, -4(t0)
bne    t0, t1, $LOOP
```

5 cycles / element inc
\[
\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}
\]

- Instructions / program depends on source code, compiler, and ISA
- CPI = cycles/instruction – depends on ISA and microarchitecture
- Time / cycle depends on microarchitecture + underlying technology
- By pipelining can lower time / cycle without increasing CPI
- By issuing multiple instructions can lower CPI further
- in-order 6 stage pipeline
- single issue
- CPI = 1 with no hazards
Pipelining CPI

Unpipelined machine

Inst 1  Inst 2  Inst 3

3 instructions, 3 cycles, CPI=1

Pipelined machine

Inst 1
Inst 2
Inst 3

3 instructions, 3 cycles, CPI=1

5-stage pipeline CPI≠5!!!
- 64 byte cache line
- non-blocking L1 cache with four cache line misses in flight
- 1 cycle L1 hit read but 50-60 cycles for miss
- locality of accesses to cache lines important

cache organized as $n$ 64B lines

memory hierarchy
allows coordinating memory between threads
fence waits until all outstanding memory reads/writes are complete

producer
1. write input data
2. fence
3. request execution on data

consumer
1. request execution on data
2. fence
3. read result data
- branch resolution
  - exceed capacity
  - mismatches
  - CPI = 1 with hit and CPI = 3 with branch mispredict

- bypassing limitations
  - 1 cycle delay between load and its use
  - loads have address calculation that adds a cycle (versus alu ops)
  - can have instruction right behind to fill load to use delay slot

- core can continue to execute after cache miss but ...
  - cache is non blocking and can allow multiple requests in parallel
  - will stall as soon as produced register is accessed
    - so only works for up to 31 registers which is big limitation
Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.

- Inst 1
- Inst 2
- Inst 3

3 instructions finish in 3 cycles
CPI = 3/3 = 1

- Inst 1
- Inst 2
- Bubble
- Inst 3

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

- Inst 1
- Bubble 1
- Inst 2
- Bubble 2
- Inst 3

3 instructions finish in 5 cycles
CPI = 5/3 = 1.67

from Krste's CS152 slide
- replicate loop body
- amortizes loop overhead

```
li t0, a
lw t1, a+64*4

$LOOP:
lw t2, 0(t0)
addw t2, t2, 1  // 1 cycle stall
sw t2, 0(t0)
lw t3, 4(t0)
addw t3, t3, 1  // 1 cycle stall
sw t3, 4(t0)
addw t0, t0, 8
bne t0, t1, $LOOP
```

4 instructions / element in limit
- avoid ld / st hazard by moving ld up
- achieves approximately 3 instructions / element

4 instructions / element

3 instructions / element
pipeline memory operations to fully saturate memory

```
lw  t0, a
lw  t1, a+64*8
$LOOP: lw  t2, 0(t0)
lw  t3, 4(t0)
lw  t4, 8(t0)
...
addw t2, t2, 1
addw t3, t3, 1
addw t4, t4, 1
...
sw  t2, 0(t0)
sw  t3, 4(t0)
sw  t4, 8(t0)
...
addw t0, t0, n
bne  t0, t1, $LOOP
```
in fact gcc can unroll and schedule perfectly for this example

```
move    x3,x0
li      x13,64
$L2:
  lw     x5,0(x4)
lw     x2,4(x4)
lw     x19,8(x4)
lw     x18,12(x4)
lw     x17,16(x4)
lw     x16,20(x4)
lw     x15,24(x4)
lw     x14,28(x4)
addw   x12,x5,1
addw   x11,x2,1
addw   x10,x19,1
addw   x9,x18,1
addw   x8,x17,1
addw   x7,x16,1
addw   x6,x15,1
addw   x5,x14,1
addw   x2,x3,8
sw      x12,0(x4)
sw      x11,4(x4)
sw      x10,8(x4)
sw      x9,12(x4)
sw      x8,16(x4)
sw      x7,20(x4)
sw      x6,24(x4)
sw      x5,28(x4)
move    x3,x2
add      x4,x4,32
bne      x2,x13,$L2
```
Reasons

- split functionality that wouldn’t fit on chip
- off load computation

Examples

- x87 floating point coprocessor
- MIPS coprocessor interface
- AXI SOC coprocessor interface
Accelerator Metrics

- efficiency
  - power
  - latency
  - throughput
  - bottlenecks?

- programmability
  - sharing data
  - coordination
  - hazards
  - language / compiler friendliness
- decoupled interfaces
- 2 src regs + 1 dst reg
- stalls on dst reg access
- mcmd is load, store, ...
- mtype is 1,2,4,8 bytes
- loads + stores tagged
- ctrl is busy and error
- latency 5-6 cycles min
coordinating

- input <= 2 scalars to coprocessor
- input data to coprocessor
- output data from coprocessor
- output scalar from coprocessor

techniques

- memory fences
- stall on reading dst register
<table>
<thead>
<tr>
<th>Rocket Core</th>
<th>Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- write input vec data</td>
<td>- ...</td>
</tr>
<tr>
<td>- fence</td>
<td>- ...</td>
</tr>
<tr>
<td>- coprocessor instruction</td>
<td>- ...</td>
</tr>
<tr>
<td>- fence</td>
<td>- executes + writes mem</td>
</tr>
<tr>
<td>- use result data</td>
<td>- ...</td>
</tr>
</tbody>
</table>
Rocket Core
- write vec data in a
- set $x_1 = a$, $x_2 = 64$
- fence
- vecinc $x_1$, $x_2$
- fence
- ...
- ...
- result data in $x_1$

Coprocessor
- ...
- ...
- ...
- busy = true
- vec inc writing $x_1$ data
- busy = false
- ...
int sum = 0;
ing a[64];
for (int i = 0; i < 64; i++)
    sum += a[i];
Rocket Core
- write input vec data
- fence
- coprocessor instruction
- use res value
- ...
- ...

Coprocessor
- ...
- ...
- ...
- exec + store res in reg
- ...
- ...
Rocket Core
- write vec data
- set vec x1 = a, x2 = 64
- fence
- vecsum x1, x2, x3
- use x3 stalls
- ...
- use x3 completes

Coprocessor
- ...
- ...
- ...
- vec sum
- x3 = sum
- ...
int* vec = { 33, 17, ... };  
int n  = 64;

// vecinc opcode = 0  
asm volatile // don’t move  
  ("fence; custom0 0, %0, %1, 0; fence",
   : // destination  
     : "r"(vec), "r"(n) // sources  
     : "memory"); // clobbers

for (int i = 0; i < n; i++)  
  printf("elt[%d] = %d\n", i, vec[i]);
int sum;
int* vec = { 33, 17, ... };
int n = 64;

// vecsum opcode = 1
asm volatile
    ("fence; custom0 %0, %1, %2, 1",
    : "=r"(sum)
    : "r"(vec), "r"(n)
    : "memory");

printf("sum = %d\n", sum);
class DecoupledIO[T <: Data](data: T)
    extends Bundle {
    val ready = Bool(INPUT)
    val valid = Bool(OUTPUT)
    val bits = data.clone.asOutput
}

object Decoupled {
    def apply(data: Data) =
        new DecoupledIO(data)
}

val results =
    Decoupled(UInt(width = 64))
Decoupled(T) Decoupled(T).flip
Ready / Valid + Queue

Producer

Queue

Consumer
Two Ready / Valid Transfers

Clock

Valid

Ready

transfer

transfer
No Ready / Valid Transfer Both Low

Clock

Valid

Ready
No Ready / Valid Transfer Valid Low

Clock

Valid

Ready
Clock

Valid

Ready
No Ready / Valid Transfer Out of Phase

Clock

Valid

Ready
No Ready / Valid Combinational Loops

Producer

Combinational Logic

bits

T
valid
Bool
ready
Bool

Consumer

Combinational Logic

bits

T
valid
Bool
ready
Bool
- producer is valid regardless of whether consumer is ready
- consumer is ready regardless of whether producer is valid

but how do you know when to move on?
How to Update Valid from Ready
Using Decoupled Interfaces in Chisel

```
producer

```val results =
  Decoupled(UInt(width = 64))
val result =
  Reg(UInt(width = 64))
val isResult =
  Reg(Bool())
...
results.valid := isResult
results.bits := result
...
when (results.ready) {
  // update state inc isResult
}
```

```
consumer

```val cmds =
  Decoupled(UInt(width = 32)).flip
val cmd =
  Reg(UInt(width = 32))
val isReady =
  Reg(Bool())
...
cmds.ready := isReady

```

```
cmd := result
...
when (cmds.valid) {
  // update state inc isReady
}
```

Rule is never

- ready in terms of valid or
- valid in terms of ready
class RoCCInstruction extends Bundle {
    val funct = Bits(width = 7)
    val rs2  = Bits(width = 5)
    val rs1  = Bits(width = 5)
    val xd   = Bool()
    val xs1  = Bool()
    val xs2  = Bool()
    val rd   = Bits(width = 5)
    val opcode = Bits(width = 7)
}
def class MemReq extends Bundle {
    val cmd = UInt(width = 2)
    val mtype = UInt(width = 2)
    val tag = UInt(width = 9)
    val addr = UInt(width = 64)
    val data = UInt(width = 64)
}

def class MemResp extends Bundle {
    val cmd = UInt(width = 2)
    val tag = UInt(width = 9)
    val mtype = UInt(width = 2)
    val data = UInt(width = 64)
}

def class OpReq extends Bundle {
    val code = new RoccInst()
    val a = UInt(width = 64)
    val b = UInt(width = 64)
}

def class OpResp extends Bundle {
    val idx = UInt(width = 5)
    val data = UInt(width = 64)
}

def class RoccIO extends Bundle {
    val busy = Bool(OUTPUT)
    val isIntr = Bool(OUTPUT)
    val memReq = Decoupled(new MemReq).flip
    val memResp = Decoupled(new MemResp)
    val opReq = Decoupled(new OpReq)
    val opResp = Decoupled(new OpResp).flip
}
- tags on read / write commands,
- responses to read / write commands,
- must keep busy asserted until all reads and writes have completed, and
- memory system has single port with accelerator having priority
- two cycles per element assuming no cache misses
- saturate single memory op per cycle
- need to pipeline this because memreq takes 4 cycle min latency
use vec idx as tag

```scala
val i = Reg(init = UInt(0, 32))
val v = Reg(init = UInt(0, 64))
val n = Reg(init = UInt(0, 32))
when (io.opRequests.valid) {
  val op = io.opRequests.deq()
  i := UInt(0)
  v := op.a
  n := op.b
  // is load coming back?
} .elsewhen (io.memResponses.valid && io.memRequests.ready) {
  val resp = io.memResponses.deq()
  when (resp.cmd === M_LOAD) {
    io.memRequests.enq(memWrite(v + resp.tag, resp.bits + 1))
  }
  // else issue more loads
} .elseWhen (i < n && io.memRequests.ready) {
  io.memRequests.enq(memRead(v + i, i))
  i := i + i(1)
}
set busy based on i and n

```scala
val i = Reg(init = UInt(0, 32))
val v = Reg(init = UInt(0, 64))
val n = Reg(init = UInt(0, 32))
io.busy := i != n
when (io.opRequests.valid) {
    val op = io.opRequests.deq()
    i := UInt(0)
    v := op.a
    n := op.b
    io.busy := Bool(true)
    // is load coming back?
} .elsewhen (io.memResponses.valid && io.memRequests.ready) {
    val resp = io.memResponses.deq()
    when (resp.cmd === M_LOAD) {
        io.memRequests.enq(memWrite(v + resp.tag, resp.bits + 1))
    }
    // else issue more loads
} .elseWhen (i < n && io.memRequests.ready) {
    io.memRequests.enq(memRead(v + i, i))
    i := i + UInt(1)
}
```
What if vec is bigger than 512 max tag size?
- have mapping from tags to indices
  - manage free list but could be expensive
- break up vec into chunks
  - don’t run ahead until done with previous chunk
- or just restrict vec ops to specific size
How could we do better?

can we achieve $\geq$ one element / cycle?

- 8 bytes / cycle so could add 8/4/2 1/2/4 byte numbers
- fatter memory interface with banked memory?
What are goals of CPU / Coprocessor?

- CPU sets up coprocessor (like scripting language)
- Coprocessor performs bigger compute
- Run at point of stalling in order pipeline with most work accomplished in coprocessor
- Saturate memory if memory bound
- Overlap CPU and coprocessor
General Purpose Processor as Accelerator

pros
- More applications work well
- Easier to program (in C)

cons
- Large
- Power inefficient
Out of Order Core Comparison

- Good at soaking up ILP from C code
- Datapath small portion of energy consumption
- Bigger consumer is all control logic and data traffic
- Lots of dynamic dataflow control logic to reorder operation
- Can achieve similar sustained Incs / Cycle but
- Lots of overhead in reg renaming, load / store unit etc

Energy Breakdown for CPU by Horowitz et al.
- Wide instruction with multiple ops / cycle
- Statically scheduled (so less energy)
- Still need to read / decode instructions
- Might not use all ops / instructions every cycle
- Non determinism in memory system causes stalls
- Hard to Justify Vec Inc (or VecSum) Operation as Accelerator
- Allow Range of Operations with Similar Form

Examples:
- Dense Linear Algebra Operations
- FFT Accelerator
Vector Programming Model

Scalar Registers
r15
r0

Vector Registers
v15
v0
[0] [1] [2] [VLRMAX-1]

Vector Length Register
VLR

Vector Arithmetic Instructions
ADDV v3, v1, v2

Vector Load and Store Instructions
LV v1, r1, r2

Base, r1
Stride, r2
Memory

from Krste’s CS152 slide
Vector Registers

```c
int a[64];
for (int i = 0; i < 64; i++)
  a[i] += 1;
```

3 cycle / element in limit

```
li vlr, 64
lv v1, x1
addvi.w v2, v1, 1
sv v2, x1
```

1 or 2 cycle / element in limit
Vector Machine Improvements

domains
- Sparse Matrix
- Structured Grids
- Convolution
- FFT

ideas
- shared infrastructure
- specialized memory access patterns
- specialized ALU
Acknowledgements

- parlab and aspire slides by Krste Asanovic
- “Ready / Valid” based on Chris Fletcher’s CS150 Writeup which is based on Greg Gibeling’s Writeup
- some computer architecture slides by Krste Asanovic