Got parallel computers but how do we write parallel software?

Principle Investigators: Krste Asanovic, Ras Bodik, Jim Demmel, Armando Fox, Tony Keaveny, 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
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

Augmented Reality

Big Data

Environment

Social Networks

Personalized Medicine
Compute Energy Iron Law

\[ \text{performance} = \text{power} \times \text{energy efficiency} \]

\[
\left( \frac{\text{tasks}}{\text{second}} \right) = \left( \frac{\text{joules}}{\text{second}} \right) \times \left( \frac{\text{tasks}}{\text{joule}} \right)
\]

- 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

“Cramming more components onto integrated circuits”, Gordon E. Moore, Electronics, 1965
Bad News: Dennard Scaling Over
For reliable high-performance digital computation, no plausible replacement for CMOS transistor ready to take over in the next 10-15 years. Modern CMOS gives

- billions of transistors,
- reliably interconnected,
- clocking at GHz,
- for a few dollars
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.

<table>
<thead>
<tr>
<th>Node</th>
<th>45nm</th>
<th>22nm</th>
<th>11nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>2008</td>
<td>2014</td>
<td>2020</td>
</tr>
<tr>
<td>Area⁻¹</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Peak freq</td>
<td>1</td>
<td>1.6</td>
<td>2.4</td>
</tr>
<tr>
<td>Power</td>
<td>1</td>
<td>1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

(4 x 1)⁻¹ = 25%  
(16 x 0.6)⁻¹ = 10%

[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
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
Background
- Designed at Berkeley
- Fifth Berkeley RISC design

Advantages
- Open source with modified BSD license
- Efficient to implement
- Extensible

State
- 2.0 Spec out
- Fast functional simulator
- GCC tool chain
- LLVM port in progress
- Boots linux
- 31 General Purpose Integer Registers
- Register to Register Operations
- Load / Store with Addressing Modes
- Control Transfer Operations
- simple symmetric format
- easy and efficient to decode

**integer instruction format**

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
<td>funct10</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>upper immediate [19:0]</td>
<td>funct3</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump offset [24:0]</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**coprocessor instruction format**

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
<td>funct7</td>
<td>xd</td>
<td>xs1</td>
<td>xs2</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>addr</td>
<td>src</td>
<td>roccinst[6:0]</td>
<td>\textit{custom-0/1/2/3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int a[64];
for (int i = 0; i < 64; i++)
    a[i] += 1;

gcc -O3 -S ...

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
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*8
$LOOP: lw   t2, 0(t0)
        addw t0, t0, 8
        addw t2, t2, 1
        sw   t2, -8(t0)
        bne  t0, t1, $LOOP
```

5 cycles / element inc
Iron Law

\[
\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}
\]

- Instructions / program depends on source code, compiler, and ISA
- CPI = \text{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!!!

from Krste's CS152 slide
- 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

```
0 1 2 3     60 61 62 63
 lines

0 1 2 3 60 61 62 63
```

Cache organized as n 64B lines

```
        Lk
        Cache
    L1 Cache
        CPU

64KB    1 cycle

...    ...
1GB    50-60 cycles

memory hierarchy
```
Memory Fence Instruction

- 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
Rocket Hazards

- 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.

<table>
<thead>
<tr>
<th>Time</th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Bubble</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Inst 1</th>
<th>Bubble 1</th>
<th>Inst 2</th>
<th>Bubble 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```assembly
li t0, a
lw t1, a+64*8

$LOOP:
lw t2, 0(t0)
addw t2, t2, 1 // 1 cycle stall
sw t2, 0(t0)
lw t3, 8(t0)
addw t3, t3, 1 // 1 cycle stall
sw t3, 8(t0)
addw t0, t0, 16
bne t0, t1, $LOOP
```

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

```
li t0, a
lw t1, a+64*8
$LOOP: lw t2, 0(t0)
    addw t2, t2, 1  // stall
    sw t2, 0(t0)
    lw t3, 8(t0)
    addw t3, t3, 1  // stall
    sw t3, 8(t0)
    addw t0, t0, 16
    bne t0, t1, $LOOP
4 instructions / element
```

```
li t0, a
lw t1, a+64*8
$LOOP: lw t2, 0(t0)
    lw t3, 8(t0)  // reschedule
    addw t2, t2, 1
    sw t2, 0(t0)
    addw t3, t3, 1
    sw t3, 8(t0)
    addw t0, t0, 16
    bne t0, t1, $LOOP
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, 8(t0)      
lw   t4, 16(t0)     
...  
addw t2, t2, 1     
addw t3, t3, 1     
addw t4, t4, 1     
...  
sw    t2, 0(t0)     
sw    t3, 8(t0)     
sw    t4, 16(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 bytes
- loads + stores tagged
- ctrl is busy and error
Rocket Pipeline with Coprocessor

- latency 5-6 cycles min
coordinating

- input $\leq$ 2 scalars to coprocessor
- input data to coprocessor
- output data from coprocessor
- output scalar from coprocessor

techniques

- memory fences
- stall on reading dst register
Rocket Core
- Write input vec data
- Fence
- Coprocessor instruction
- Fence
- Use result data

Coprocessor
- ...
- ...
- Executes + writes mem
- ...

Programming Template for Memory Result
Rocket Core
- write vec x1, x2 = 64
- fence
- vecinc x1, x2
- fence
- ...
- ...
- result data in x1

Coprocessor
- ...
- ...
- ...
- busy = true
- vec inc writing x1 data
- busy = false
- ...

Programming Vec Inc
int sum = 0;
int a[64];
for (int i = 0; i < 64; i++)
    sum += a[i];
Rocket Core
- write input vec data
- fence
- coprocessor instruction
- use result and stall
- ...
- ...

Coprocessor
- ...
- ...
- execute sum
- store result
- ...
Rocket Core
- write vec x1, 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(OUTPUT)
    val valid = Bool(INPUT)
    val bits = data.clone.asInput
}

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

val results =
    Decoupled(UInt(width = 64))
Using Decoupled Interfaces in Chisel

**producer**

```chisel
val results =
  Decoupled(UInt(width = 64))
val result =
  Reg(UInt(width = 64))
results.valid := Bool(false)
results.bits := UInt(0)
...
when (isResult && results.ready) {
  // enq
  results.valid := Bool(true)
  results.bits := result
}
```

**consumer**

```chisel
val cmds =
  Decoupled(UInt(width = 32)).flip
val cmd =
  Reg(UInt(width = 32))
cmds.ready := Bool(false)
...
...
when (cmds.valid) {
  // deq
  results.ready := Bool(true)
  cmd := result
}
```
def class RoccInst extends Bundle {
    val rd = UInt(width = 5)
    val rs1 = UInt(width = 5)
    val rs2 = UInt(width = 5)
    val inst = UInt(width = 7)
    val isXd = Bool()
    val isXs1 = Bool()
    val isXs2 = Bool()
    val opcode = UInt(width = 7)
}
```scala
def class MemReq extends Bundle {
  val cmd = UInt(width = 2)
  val mtype = UInt(width = 3)
  val tag = UInt(width = 9)
  val mask = UInt(width = 8)
  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 mask = UInt(width = 8)
  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 r = UInt(width = 64)
}
def class RoccIO extends Bundle {
  val busy = Bool(OUTPUT)
  val isInstr = Bool(OUTPUT)
  val memReq = Decoupled(new MemReq).flip
  val memResp = Decoupled(new MemResp)
  val opReq = Decoupled(new OpReq)
  val opResp = Decoupled(new OpResp).flip
}
```
- 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

val rdIdx = 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()
  rdIdx := 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 (rdIdx < n && io.memRequests.ready) {
  io.memRequests.enq(memRead(v + i, i))
  rdIdx := rdIdx + UInt(1)
}
count mem responses

```scala
val rdIdx = Reg(init = UInt(0, 32))
val v = Reg(init = UInt(0, 64))
val n = Reg(init = UInt(0, 32))
io.busy := cnt != UInt(0)

when (io.opRequests.valid) {
  val op = io.opRequests.deq()
  rdIdx := UInt(0)
  v := op.a
  cnt := 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))
    cnt := cnt - UInt(1)
  }
  // else issue more loads
} .elseWhen (rdReg < n && io.memRequests.ready) {
  io.memRequests.enq(memRead(v + i, i))
  rdIdx := rdIdx + 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 >= 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:
- v0
- v1
- v2
- v3

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 Chaining

LV v1
MULV v3, v1, v2
ADDV v5, v3, v4

from Krste's CS152 slide
Vector Chaining CPI

Without chaining, must wait for last element of result to be written before starting dependent instruction

With chaining, can start dependent instruction as soon as first result appears

from Krste’s CS152 slide
- multiple coprocessor instructions in flight
- coordinate between instructions
domains
- Sparse Matrix
- Structured Grids
- Convolution
- FFT

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

- parlab and aspire slides by Krste Asanovic
- some computer architecture slides by Krste Asanovic