Vector Computers

Joel Emer
Computer Science & Artificial Intelligence Lab
M.I.T.
### Loop Unrolled Code Schedule

**loop:**
- `ld f1, 0(r1)`
- `ld f2, 8(r1)`
- `ld f3, 16(r1)`
- `ld f4, 24(r1)`
- `add r1, 32`
- `fadd f5, f0, f1`
- `fadd f6, f0, f2`
- `fadd f7, f0, f3`
- `fadd f8, f0, f4`
- `sd f5, 0(r2)`
- `sd f6, 8(r2)`
- `sd f7, 16(r2)`
- `sd f8, 24(r2)`
- `add r2, 32`
- `bne r1, r3, loop`

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ld f4</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fadd f5</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fadd f6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fadd f7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fadd f8</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sd f5</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sd f6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sd f7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sd f8</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add r2</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bne</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sd f8</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Schedule**

---

Sanchez & Emer

April 25, 2016
Vector Supercomputers

_Epitomized by Cray-1, 1976:_

- **Scalar Unit**
  - Load/Store Architecture

- **Vector Extension**
  - Vector Registers
  - Vector Instructions

- **Implementation**
  - Hardwired Control
  - Highly Pipelined Functional Units
  - No Data Caches
  - Interleaved Memory System
  - No Virtual Memory
Cray-1 (1976)
Cray-1 (1976)

- Single Port Memory
- 16 banks of 64-bit words + 8-bit SECDED
- 80MW/sec data load/store
- 320MW/sec instruction buffer refill

memory bank cycle 50 ns

- Processor cycle 12.5 ns (80MHz)

4 Instruction Buffers

64-bitx16

64 Element Vector Registers

- V0, V1, V2, V3, V4, V5, V6, V7
- V. Mask
- V. Length

- FP Add
- FP Mul
- FP Recip
- Int Add
- Int Logic
- Int Shift
- Pop Cnt
- Addr Add
- Addr Mul

- NIP
- CIP
- LIP

April 25, 2016
Vector Programming Model

Scalar Registers

Vector Registers

Vector Arithmetic Instructions

ADDV v3, v1, v2
Vector Programming Model

Scalar Registers
- r15
- r0

Vector Registers
- v15
- v0

[0] [1] [2] [VLRMAX-1]

Vector Length Register (VLR)

Vector Load and Store Instructions
LV v1, r1, r2

Base, r1
Stride, r2
Memory
Compiler-based Vectorization

Scalar code

\[
\begin{align*}
\text{Ld } A_i \\
\text{Ld } B_i \\
\text{Add} \\
\text{St } C_i \\
\text{Ld } A_{i+1} \\
\text{Ld } B_{i+1} \\
\text{Add} \\
\text{St } C_{i+1}
\end{align*}
\]

Vector code

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

\[
C[i] = A[i] + B[i];
\]

Compiler recognizes independent operations with loop dependence analysis
Vector Code Example

# C code
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];

# Scalar Code
LI R4, 64
loop:
    L.D F0, 0(R1)
    L.D F2, 0(R2)
    ADD.D F4, F2, F0
    S.D F4, 0(R3)
    DADDIU R1, 8
    DADDIU R2, 8
    DADDIU R3, 8
    DSUBIU R4, 1
    BNEZ R4, loop

# Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
Vector ISA Attributes

- **Compact**
  - one short instruction encodes N operations
  - many implicit bookkeeping/control operations

- **Expressive, tells hardware that these N operations:**
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory
    (unit-stride load/store)
  - access memory in a known pattern
    (strided load/store)
Vector ISA Hardware Implications

- Large amount of work per instruction
  - Less instruction fetch bandwidth requirements
  - Allows simplified instruction fetch design

- Implicit bookkeeping operations
  - Bookkeeping can run in parallel with main compute

- Disjoint vector element accesses
  - Banked rather than multi-ported register files

- No data dependence within a vector
  - Amenable to deeply pipelined/parallel designs

- Known regular memory access pattern
  - Allows for banked memory for higher bandwidth
Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

```
V3 <- v1 * v2
```

Six stage multiply pipeline
Vector Instruction Execution

ADDV C, A, B

Execution using one pipelined functional unit

Execution using four pipelined functional units

Vector Unit Structure

Function Unit (Adder)

Vector Registers

Function Unit (Mult)

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Memory Subsystem
Vector Instruction Parallelism

Can overlap execution of multiple vector instructions
- example machine has 32 elements per vector register and 8 lanes

Load Unit

Multiply Unit

Add Unit

Load

mul

mul

add

add

Instruction issue

Complete 24 operations/cycle while issuing 1 short instruction/cycle
Vector Chaining

Problem: Long latency for RAW register dependencies

- Vector version of register bypassing
  - introduced with Cray-1

```
LV v1
MULV v3, v1, v2
ADDV v5, v3, v4
```
Vector Chaining Advantage

- 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.
Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- Bank busy time: Cycles between accesses to same bank
- Allows 16 parallel accesses (if data in different banks)

Vector Registers

Address Generator

Base  Stride

Memory Banks
Vector Stripmining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit in registers, "Stripmining"

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```

```
ANDI R1, N, 63  # N mod 64
MTC1 VLR, R1    # Do remainder
```

```
loop:
    LV V1, RA
    DSLL R2, R1, 3  # Multiply by 8
    DADDU RA, RA, R2 # Bump pointer
    LV V2, RB
    DADDU RB, RB, R2
    ADDV.D V3, V1, V2
    SV V3, RC
    DADDU RC, RC, R2
    DSUBU N, N, R1  # Subtract elements
    LI R1, 64
    MTC1 VLR, R1    # Reset full length
    BGTZ N, loop    # Any more to do?
```
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```c
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers
- vector version of predicate registers, 1 bit per element

...and *maskable* vector instructions
- vector operation becomes NOP at elements where mask bit is clear

Code example:
```
CVM          # Turn on all elements
LV vA, rA    # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
LV vA, rB    # Load B vector into A under mask
SV vA, rA    # Store A back to memory under mask
```
Masked Vector Instructions

Simple Implementation

- execute all N operations, turn off result writeback according to mask

Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks

\[
\begin{align*}
M[0] &= 0 & C[0] \\
\end{align*}
\]

Write Enable

Write data port

\[
\begin{align*}
M[0] &= 0 & C[0] \\
\end{align*}
\]

Write data port
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

```c
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (*Gather*)

```assembly
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base
LV vB, rB       # Load B vector
ADDV.D vA, vB, vC # Do add
SV vA, rA       # Store result
```
Vector Scatter/Gather

Scatter example:

```c
for (i=0; i<N; i++)
    A[B[i]]++;
```

Is following a correct translation?

```
LV vB, rB   # Load indices in B vector
LVI vA, rA, vB # Gather initial A values
ADDV vA, vA, 1 # Increment
SVI vA, rA, vB # Scatter incremented values
```
Multimedia Extensions

• Short vectors added to existing general-purpose ISAs

• Initially, 64-bit registers split into 2x32b or 4x16b or 8x8b

• Limited instruction set:
  – No vector length control
  – No vector masks
  – No strided load/store or scatter/gather
  – Loads must be aligned to 64-bit boundary

• Limited vector register length:
  – Requires superscalar dispatch to keep multiply/add/load units busy
  – Loop unrolling to hide latencies increases register pressure

• Trend towards fuller vector support in microprocessors
  – e.g. x86: MMX → SSEx (128 bits) → AVX (256 bits) → AVX-512 (512 bits/masks)
Knight's Landing (KNL) CPU

- 2-wide decode/retire
- 6-wide execute
- 72-entry ROB
- 64B cache ports
- Fast unaligned access
- Fast scatter/gather
- OoO int/fp RS
- In-order mem RS
- 4 thread SMT
- Many shared resources
  - ROB, rename, RS...
  - Caches, TLB
- Several thread choosers

Knight’s Landing (KNL) Mesh

Mesh of Rings
- Rows/columns (half) ring
- YX routing
- Messages arbitration on:
  - Injection
  - Turns

Cache Coherent Interconnect
- MESIF protocol
- Distributed directory
to filter snoops

Partitioning modes
- All-to-all
- Quadrant
- Sub-NUMA

Next: GPUs

Thank you!