Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions (MMX, SSE)
- Graphics Processor Units (GPUs)
- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

VMIPS

- Example architecture: VMIPS
  - Loosely based on Cray-1
  - Vector registers
    - Each register holds a 64-element, 64 bits/element vector
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers, 32 floating-point registers

VMIPS: Architecture
SAXPY and DAXPY

//SAXPY
//Single precision
float X[64], Y[64];
float const a=av;
for (i=0;i<64;i++)
    Y[i]=a.X[i]+Y[i];

//DAXPY
//Double precision
double X[64], Y[64];
double const a=av;
for (i=0;i<64;i++)
    Y[i]=a.X[i]+Y[i];

// Execution time depends on three factors:
• Length of operand vectors
• Structural hazards
• Data dependencies

• VMIPS functional units consume one element per clock cycle
• Execution time is approximately the vector length

• Convoy
• Set of vector instructions that could potentially execute together

Example
LV V1,Rx ; load vector X
MULVS.D V2,V1,F0 ; vector-scalar multiply
LV V3,Ry ; load vector Y
ADDVV V4,V2,V3 ; add
SV Ry,V4 ; store the result

 RAW A Sahu 2

A Sahu 2

VMIPS Instructions
• ADDVV.D: add two vectors
• ADDVS.D: add vector to a scalar
• LV/SV: vector load and vector store from address
  Y = a.X + Y (Double prec. AXPY)

• Example: DAXPY
  L.D F0,a ; load scalar a
  LV V1,Rx ; load vector X
  MULVS.D V2,V1,F0 ; vector-scalar multiply
  LV V3,Ry ; load vector Y
  ADDVV V4,V2,V3 ; add
  SV Ry,V4 ; store the result
• Requires 6 instructions vs. almost 600 for MIPS

A Sahu 3

Challenges
• Start up time
  • Latency of vector functional unit
    • All are pipelined
      • Assume the same as Cray-1
        • Floating-point add => 6 clock cycles (6 stages)
        • Floating-point multiply => 7 clock cycles (7 stages)
        • Floating-point divide => 20 clock cycles (20 stages)
        • Vector load => 12 clock cycles (12 stages)
Challenges

• Improvements:
  – Can we process > 1 element per clock cycle
  – Non-64 wide vectors
  – IF statements in vector code
  – Memory system optimizations to support vector processors
  – Multiple dimensional matrices
  – Sparse matrices
  – Programming a vector computer

Multiple Lanes

• Element \( n \) of vector register \( A \) is "hardwired" to element \( n \) of vector register \( B \)
  – Allows for multiple hardware lanes

Vector Length Register

• Vector length not known at compile time?
• Use Vector Length Register (VLR)
• Use strip mining for vectors over the maximum length:

Vector Mask Registers

• Consider:

\[
\begin{align*}
\text{for } (i = 0; i < 64; i = i+1) & \\
& \text{if } (X[i] \neq 0) \\
& \quad X[i] = X[i] - Y[i]; \\
& \text{// Predicate Instruction} \\
\text{for } (i = 0; i < 64; i = i+1) & \\
& \text{if } (X[i] = 0) \quad M[i] = 1; \\
& \text{// Predicate Statements} \\
& \text{M[i] ? } X[i] = X[i] - Y[i]; \quad \text{// Predicate Statements} \\
\end{align*}
\]

Use vector mask register to "disable" elements:

\[
\begin{align*}
\text{LV} & \quad V1.Rx \quad \text{load vector X into V1} \\
\text{LV} & \quad V2.Ry \quad \text{load vector Y} \\
\text{LD} & \quad F0,R0 \quad \text{load FP zero into F0} \\
\text{SNEVS.D} & \quad V1,F0 \quad \text{sets VM(i) to 1 if V3(i)=F0} \\
\text{SUBV.D} & \quad V1,V1,V2 \quad \text{subtract under vector mask} \\
\text{SV} & \quad Rx,V1 \quad \text{store the result in X} \\
\end{align*}
\]

• GFLOPS rate decreases!
Memory Banks

- Memory system must be designed to support high BW for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 LD and 2 ST/cycle
  - Proc cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?

  \[ \text{SRAM is busy for } \text{ceil}(15/2.167)=7 \text{ cycle, } 32\times 6\times 7=1344 \text{ banks} \]

Stride

\[
\begin{array}{ll}
\text{for } (i = 0; i < 100; i=i+1) \\
\text{for } (j = 0; j < 100; j=j+1) \{ \\
A[i][j] = A[i][j] + B[i][k] \times D[k][j]; \\
\}
\end{array}
\]

- Vector processor can handle stride >1
- Once a vector is loaded vector register
  - It act as if it had logically adjacent element
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - \#banks / LCM(stride,\#banks) < bank busy time

Example Stride

- 8 memory bank, bank busy time =6 clock
- Memory latency=12 cycle
- How long it will take time to load 64 element vector load with stride=1
  - \(12 + 64 = 76 \text{ clock} \)
  - With Stride=32
  - \(12 + 1 + 6 \times 63 = 391 \text{ clock} \)

Scatter-Gather: how to handle space matrix addition?

\[
\begin{array}{ll}
\text{for } (i = 0; i < n; i=i+1) \\
\end{array}
\]

- Gather: Takes the index vector and fetch the vector whole elements are at the address
  - Result is a dense vector in a vector register
- Scatter: Store in expanded form using the same index vector
  - LVI Load vector indexed or GATHER
  - SVI Store vector indexed or SCATTER

Scatter-Gather: how to handle space matrix addition?

\[
\begin{array}{ll}
\text{Consider:} \\
\text{for } (i = 0; i < n; i=i+1) \\
\end{array}
\]

- Use index vector:
  - LV \(V_k, R_k\) ; load \(K\)
  - LVI \(V_a, (Ra+V_k)\) ; load \(A[K]\)
  - LV \(V_m, R_m\) ; load \(M\)
  - LVI \(V_c, (Rc+V_m)\) ; load \(C[M]\)
  - ADDV LV \(V_a, V_a, V_c\) ; add them
  - SVI \(V_a, (Ra+V_k), V_a\) ; store \(A[K]\)
SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

Example SIMD Code

- Example DXPY:
  ```
  L.D F0,a ; load scalar a
  MOV F1,F0 ; copy a into F1 for SIMD MUL
  MOV F2,F0 ; copy a into F2 for SIMD MUL
  MOV F3,F0 ; copy a into F3 for SIMD MUL
  DADDIU R4,Rx,#512 ; last address to load
  Loop: L.4D F4,R[Rx] ; load X[i], X[i+1], X[i+2], X[i+3]
  MUL.4D F4,F4,F0 ; a×X[i], a×X[i+1], a×X[i+2], a×X[i+3]
  ADD.4D F8,F8,F4 ; a×X[i]+Y[i], ... , a×X[i+3]+Y[i+3]
  S.4D 0[Ry],F8 ; store into Y[i], Y[i+1], Y[i+2], Y[i+3]
  DADDIU Rx,Ry,#32 ; increment index to X
  DADDIU Ry,Ry,#32 ; increment index to Y
  DSUBU R20,R4,Rx ; compute bound
  BNEZ R20,Loop ; check if done
  ```

Roofline Performance Model

- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read

Examples

- Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)

Architecture Trend

- 1999: Shift to multithreading: Superscalar and Pipeline
- 2002: ILP and HT
- 2011: Shift to multithreaded, Reduced emphasis on ILP, Introduce thread level P.
If we extrapolate the trend, in a few generations, Pentium will look like:

Of course, we know it did not happen.

Q: What happened instead? Why?

Evolution of Multi-core CPUs

Penryn
Chip area breakdown

Bloomfield

Gulftown

Beckton

Q: What can you observe? Why?

Less than 10% of total chip area is used for the real execution.

Q: Why?

NVIDIA’s Motivation of Simple Core

“This [multiple IA-core] approach is analogous to trying to build an airplane by putting wings on a train.” – Bill Dally, NVIDIA

1. 2010 Eckert-Mauchly (ENIVAC inventor) Award, considered the highest prize in computer architecture
2. 2004 IEEE Computer Society Seymour Cray Computer Engineering Award
3. 2000 ACM Maurice Wilkes Award.

The Brick Wall -- UC Berkeley’s View

Power Wall: power expensive, transistors free
Memory Wall: Memory slow, multiplies fast
ILP Wall: diminishing returns on more ILP HW

Power Wall + Memory Wall + ILP Wall = Brick Wall
Hide the memory latency through fine-grained interleaved threading.

The granularity of interleaved multi-threading:
- 100 cycles: hide off-chip memory latency
- 10 cycles: + hide cache latency
- 1 cycle: + hide branch latency, instruction dependency

Fine-grained interleaved multi-threading:
**Pros:**
- remove branch predictor, OOO scheduler, large cache

**Cons:**
- register pressure, etc.

Without and with fine-grained interleaved threading

Pros:
- reduce cache size,
- no branch predictor,
- no OOO scheduler

Cons:
- register pressure,
- thread scheduler,
- require huge parallelism

Reducing large cache gives 2x computational density.

Q: Can we make further improvements?

Hint:
We have only utilized thread level parallelism (TLP) so far.

GPU uses wide SIMD: 8/16/24/... processing elements (PEs)
CPU uses short SIMD: usually has vector width of 4.

SSE has 4 data lanes
GPU has 8/16/24/... data lanes

Supporting interleaved threading + SIMD execution
Hide vector width using scalar threads.

Assume 32 threads are grouped into one warp.

The Stream Multiprocessor (SM) is a light weight core compared to IA core.

Light weight PE:
Fused Multiply Add (FMA)

SFU: Special Function Unit

NVIDIA Fermi, 512 Processing Elements (PEs)