MSc Informatics Eng.

2013/14

A.J.Proença

Data Parallelism 1 (*vector, SIMD ext., GPU*)

(most slides are borrowed)

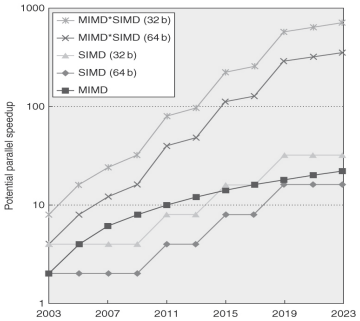
- When exploiting ILP, goal is to minimize CPI
  - Pipeline CPI =>
    - Ideal pipeline CPI + ✔
    - Structural stalls + ✔
    - Data hazard stalls + ✔
    - Control stalls + ✔
    - Memory stalls ... *cache techniques ...* ✔
  - Multiple issue =>
    - find enough parallelism to keep pipeline(s) occupied ✔
  - Multithreading =>
    - find ways to keep pipeline(s) occupied ✔
- Insert data parallelism features: SIMD...

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processing
- 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 programmers to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)
- For x86 processors:
  - Expected grow: 2 more cores/chip/year
  - SIMD width: 2x every 4 years
  - Potential speedup: SIMD 2x that from MIMD!



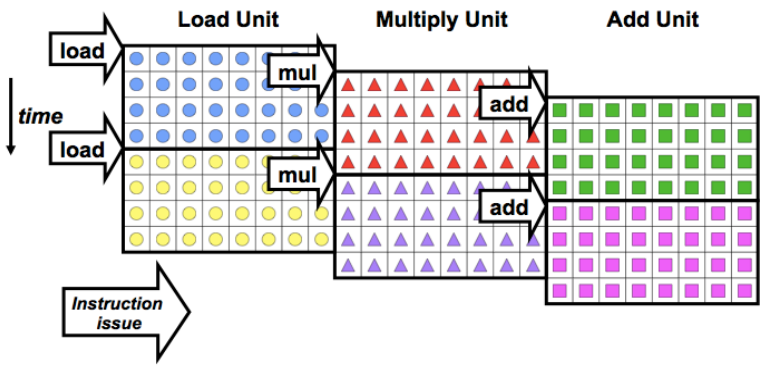
Vector Architectures

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

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

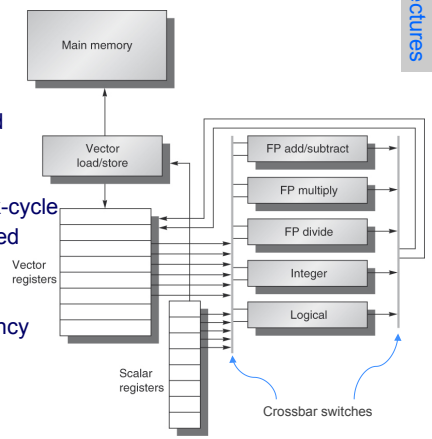
- Consider machine with 32 elements per vector register and 8 lanes:



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

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, new op each clock-cycle
    - Data & control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - 1 word/clock-cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers



VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
- 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 the execution of 6 instructions *versus* almost 600 for MIPS

Vector Execution Time

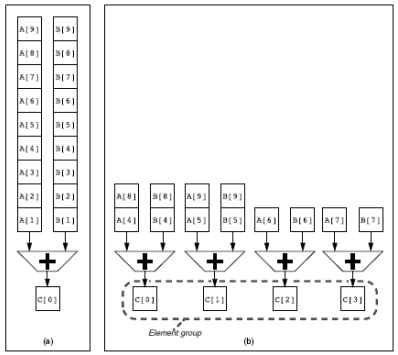
- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together in one unit of time, *chime*

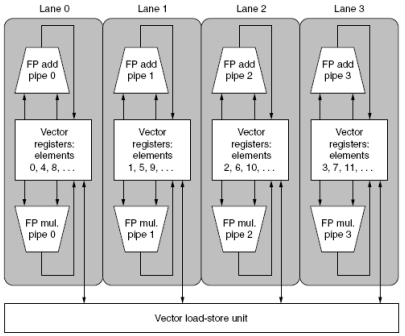
Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements:
  - > 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

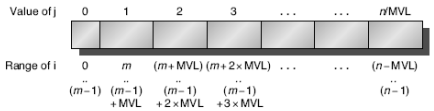




Vector Length Register

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

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
  for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
    Y[i] = a * X[i] + Y[i] ; /*main operation*/
  low = low + VL; /*start of next vector*/
  VL = MVL; /*reset the length to maximum vector length*/
}
```



Vector Mask Registers

- Handling IF statements in Vector Loops:

```
for (i = 0; i < 64; i=i+1)
      if (X[i] != 0)
          X[i] = X[i] – Y[i];
```

- Use vector mask register to “disable” elements:

```
LV        V1,Rx       ;load vector X into V1
LV        V2,Ry       ;load vector Y
L.D       F0,#0       ;load FP zero into F0
SNEVS.D   V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D   V1,V1,V2    ;subtract under vector mask
SV        Rx,V1       ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth 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 (Cray T932):
  - 32 processors, each generating 4 loads and 2 stores per cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?

Stride

- Handling multidimensional arrays in Vector Architectures:

```
for (i = 0; i < 100; i=i+1) {
    for (j = 0; j < 100; j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
}
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride* (in VMIPS: load/store vector with stride)
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / Least_Common_Multiple (stride, #banks) < bank busy time

Scatter-Gather

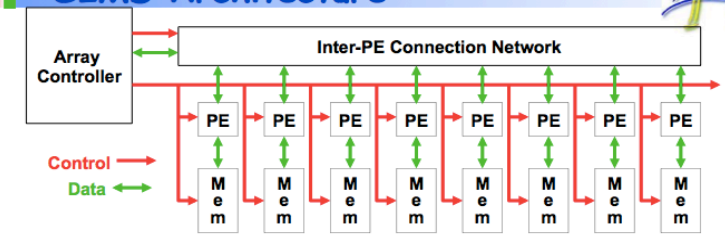
- Handling sparse matrices in Vector Architectures:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Use index vector:

```
LV       Vk, Rk          ;load K
LVI      Va, (Ra+Vk)     ;load A[K[]]
LV       Vm, Rm          ;load M
LVI      Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D  Va, Va, Vc      ;add them
SVI      (Ra+Vk), Va     ;store A[K[]]
```

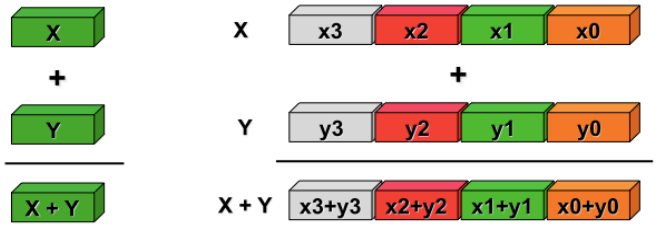
SIMD Architecture



- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized
- Recent Return to Popularity:
  - GPU (Graphics Processing Units) have SIMD properties
  - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- Dual between Vector and SIMD execution

Pseudo SIMD: (Poor-Man's SIMD?)

- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

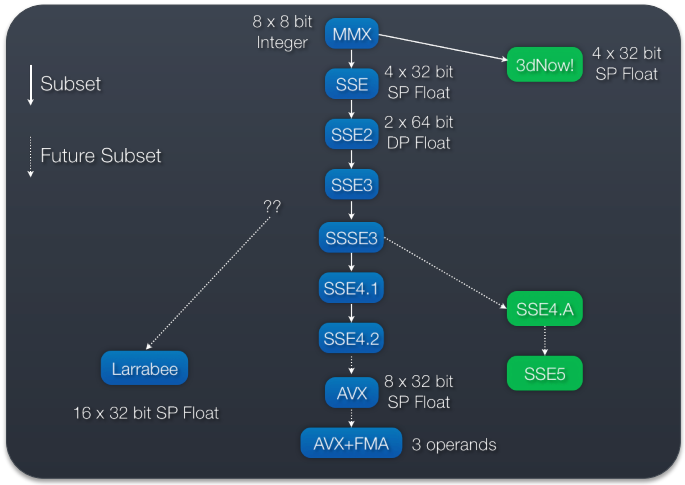
SIMD Extensions

- 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

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector eXtensions (2010)
    - Four 64-bit integer/fp ops
- Operands must be in consecutive and aligned memory locations

A Brief History of x86 SIMD



Example SIMD Code

- Example DAXPY:

```
        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    F4,0[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]
        L.4D    F8,0[Ry]    ;load Y[i], Y[i+1], Y[i+2], Y[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,Rx,#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
```


Graphical Processing Units

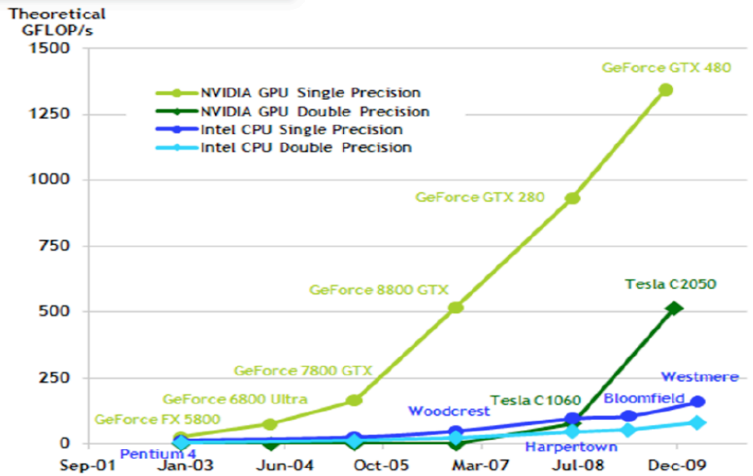
- Question to GPU architects:
  - *Given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?*
- Key ideas:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA_thread*
  - Programming model follows SIMT: "*Single Instruction Multiple Thread*"

Classifying GPUs

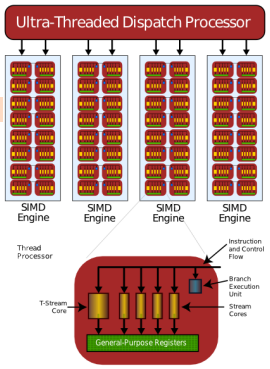
- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degradetion
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**GPU device**

Performance gap between GPUs and CPUs



What is a core?



- Is a core an ALU?
  - ATI claims 800 streaming processors!!
    - 5 way VLIW * 16 way SIMD * 10 "SIMD cores"
- Is a core a SIMD vector unit?
  - NVidia claims 512 streaming processors!!
    - 32 way SIMD * 16 "multiprocessors"
      - To match ATI, they could count another factor of 2 for dual-issue
- In these slides, we use core consistent with the CPU world
  - Superscalar, VLIW, SIMD are part of a core's architecture, not the #cores

NVIDIA GPU Architecture

Graphical Processing Units

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

The CUDA Programming Model

- *Compute Unified Device Architecture*
- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchr. & data sharing between small groups of threads
- CUDA programs are written in C with extensions
- OpenCL inspired by CUDA, but hw & sw vendor neutral
  - Programming model essentially identical