Mestrado em Informática

2009/10

A.J.Proença

Tema

Arquitecturas Paralelas (1)

Adaptado de

Computer Organization and Design, 4th Ed, Patterson & Hennessy, © 2009, MK

Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, …
4. Topologias de interligação

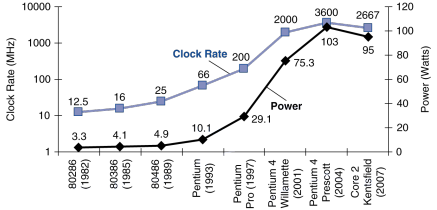
Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

The Power Wall: Power Trends



- **In CMOS IC technology**

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30 | 5V → 1V | ×1000

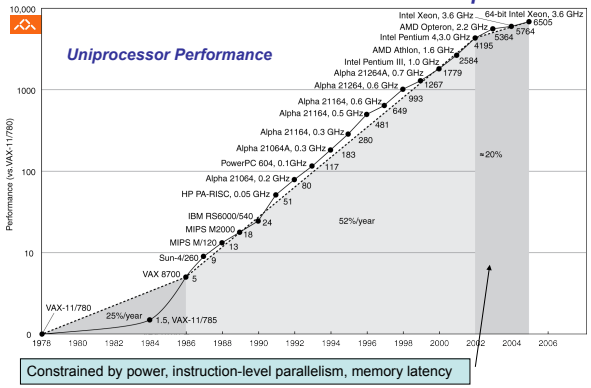
Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

The Sea Change: The Switch to Multiprocessors



Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Parallelism and Instructions: Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Parallelism and Computer Arithmetic: Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Parallelism and Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

ILP: Multiple Issue

- Static multiple issue
  - Compiler groups instructions into “issue packets”
  - An “issue packet” Specifies multiple concurrent operations
    - ➢Very Long Instruction Word (VLIW)
  - Compiler detects and avoids hazards
- Dynamic multiple issue (“superscalar” processor)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

ILP: Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Avoid load and cache miss delay
    - Roll back if location is updated

ILP: Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

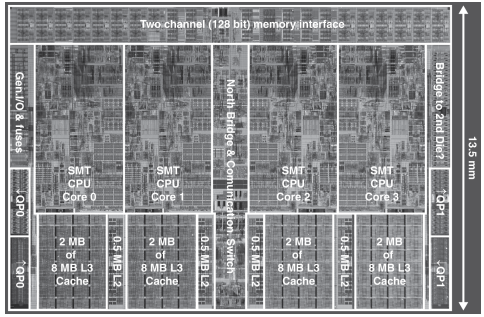
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Multilevel Cache Considerations

- Primary cache
  - Focus on minimizing hit time for a shorter clock cycle
    - Smaller with smaller block sizes
- L-2 cache
  - Focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
    - Higher levels of associativity
  - Hit time has less overall impact
- **Results**
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Multilevel On-Chip Caches: Intel Nehalem

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write -back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write -back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write -back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back /allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Measuring Performance: SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

Measuring Performance: CINT2006 for Opteron X4 2356 (Barcelona)

Name	Description	IC×10⁹	CPI	Tc (ns)	Exec time	Ref time	SPECratio
perl	Interpreted string processing	2,118	0.75	0.40	637	9,777	15.3
bzip2	Block-sorting compression	2,389	0.85	0.40	817	9,650	11.8
gcc	GNU C Compiler	1,050	1.72	0.47	24	8,050	11.1
mcf	Combinatorial optimization	336	10.00	0.40	1,345	9,120	6.8
go	Go game (AI)	1,658	1.09	0.40	721	10,490	14.6
hmmer	Search gene sequence	2,783	0.80	0.40	890	9,330	10.5
sjeng	Chess game (AI)	2,176	0.96	0.48	37	12,100	14.5
libquantum	Quantum computer simulation	1,623	1.61	0.40	1,047	20,720	19.8
h264avc	Video compression	3,102	0.80	0.40	993	22,130	22.3
omnetpp	Discrete event simulation	587	2.94	0.40	690	6,250	9.1
astar	Games/path finding	1,082	1.79	0.40	773	7,020	9.1
xalancbmk	XML parsing	1,058	2.70	0.40	1,143	6,900	6.0
Geometric mean							11.7

High cache miss rates

Measuring Performance: CINT2006 for Opteron X4 2356 (Barcelona)

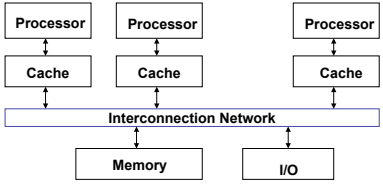
Name	CPI	L1 D cache misses/1000 instr	L2 D cache misses/1000 instr	DRAM accesses/1000 instr
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, …
4. Topologias de interligação

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

- **Multiprocessor** – a computer system with at least two processors



  - Can deliver high throughput for independent jobs via job-level parallelism or process-level parallelism
  - And improve the run time of a *single* program that has been specially crafted to run on a multiprocessor - a parallel processing program

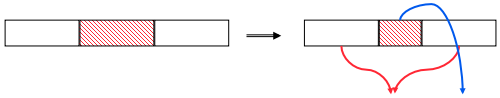
- The power challenge has forced a change in the design of microprocessors
  - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core – Chip Multicore microProcessors (CMPs) – in a single IC
  - The number of cores is expected to double every two years

Product	AMD Barcelona	Intel Nehalem	IBM Power 6	Sun Niagara 2
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Power	120 W	~100 W?	~100 W?	94 W

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



ExTime w/ E = ExTime w/o E × ((1-F) + F/S)

Speedup w/ E = 1 / ((1-F) + F/S)

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  Speedup w/ E = 1/(.091 + .909/10) = 1/0.1819 = 5.5
- What if there are 100 processors ?
  Speedup w/ E = 1/(.091 + .909/100) = 1/0.10009 = 10.0
- What if the matrices are100 by 100 (or 10,010 adds in total) on 10 processors?
  Speedup w/ E = 1/(.001 + .999/10) = 1/0.1009 = 9.9
- What if there are 100 processors ?
  Speedup w/ E = 1/(.001 + .999/100) = 1/0.01099 = 91

Not Fooling Yourself: Amdahl's Law

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Best you could ever hope to do:

$$\text{Speedup}_{maximum} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right)}$$

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - Strong scaling – when speedup can be achieved on a multiprocessor without increasing the size of the problem
  - Weak scaling – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors
- Load balancing is another important factor. Just a single processor with twice the load of the others cuts the speedup almost in half

Multiprocessor/Clusters Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- They come in two styles
  - Uniform memory access (UMA) multiprocessors
  - Nonuniform memory access (NUMA) multiprocessors
    - Programming NUMAs are harder
    - But NUMAs can scale to larger sizes and have lower latency to local memory

Summing 100,000 Numbers on 100 Proc. SMP

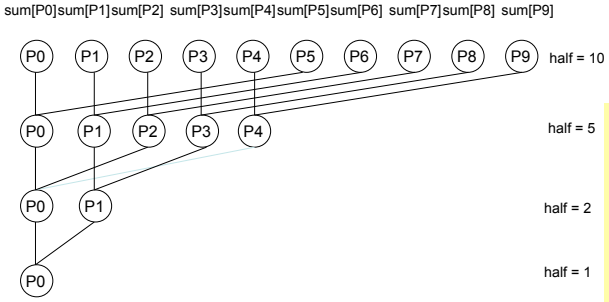
- Processors start by running a loop that sums their subset of vector `A` numbers (vectors `A` and `sum` are shared variables, `Pn` is the processor's number, `i` is a private variable)

```
sum[Pn] = 0;
for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- The processors then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of processors)) – reduction

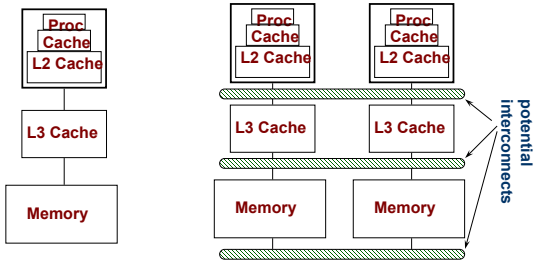
```
repeat
  synch();                    /*synchronize first
  if (half%2 != 0 && Pn == 0)
   sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half]
until (half == 1);  /*final sum in sum[0]
```


An Example with 10 Processors

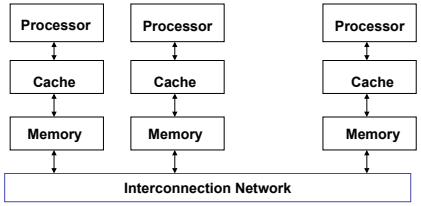


- Need to be able to coordinate processes working on a common task
- Lock variables (semaphores) are used to coordinate or synchronize processes
- Need an architecture-supported arbitration mechanism to decide which processor gets access to the lock variable
  - Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins
- Need an architecture-supported operation that locks the variable
  - Locking can be done via an atomic swap operation

Conventional Storage Hierarchy



- Each processor has its own private address space
- Q1 – Processors share data by *explicitly* sending and receiving information (message passing)
- Q2 – Coordination is built into message passing primitives (message send and message receive)



Summing 100,000 Numbers on 100 Proc. MPP

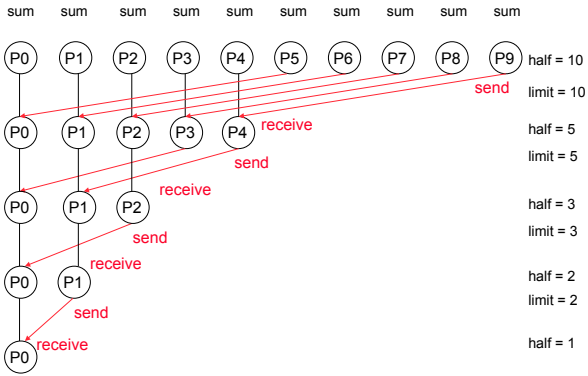
- Start by distributing 1000 elements of vector `A` to each of the local memories and summing each subset in parallel

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
  sum = sum + Al[i];      /* sum local array
  subset
```

- The processors then coordinate in adding together the sub sums (`Pn` is the number of processors, `send(x,y)` sends value `y` to processor `x`, and `receive()` receives a value)

```
half = 100;
limit = 100;
repeat
  half = (half+1)/2;      /*dividing line
  if (Pn>= half && Pn<limit) send(Pn-half,sum);
  if (Pn<(limit/2)) sum = sum + receive();
  limit = half;
until (half == 1);        /*final sum in P0's sum
```

An Example with 10 Processors



Pros and Cons of Message Passing

- Message sending and receiving is *much* slower than addition, for example
- But message passing multiprocessors and much easier for hardware designers to design
  - Don't have to worry about cache coherency for example
- The advantage for programmers is that communication is explicit, so there are fewer "performance surprises" than with the implicit communication in cache-coherent SMPs.
  - Message passing standard MPI-2 (www.mpi-forum.org)
- However, its harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
  - With cache-coherent shared memory the hardware figures out what data needs to be communicated

Networks of Workstations (NOWs) Clusters

- Clusters of off-the-shelf, whole computers with multiple private address spaces connected using the I/O bus of the computers
  - lower bandwidth than multiprocessor that use the processor-memory (front side) bus
  - lower speed network links
  - more conflicts with I/O traffic
- Clusters of N processors have N copies of the OS limiting the memory available for applications
- Improved system availability and expandability
  - easier to replace a machine without bringing down the whole system
  - allows rapid, incremental expandability
- Economy-of-scale advantages with respect to costs

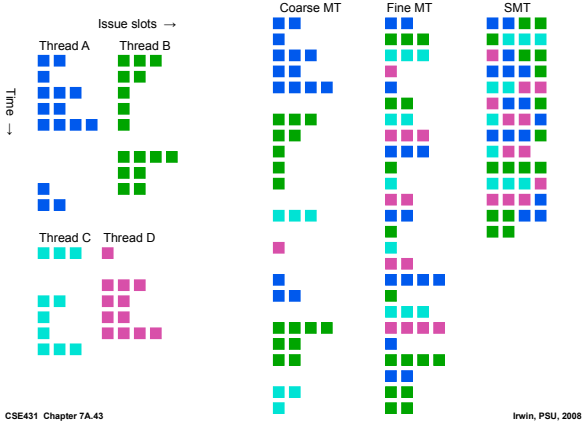
Multithreading on A Chip

- Find a way to "hide" true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions
- Hardware multithreading – increase the utilization of resources on a chip by allowing multiple processes (threads) to share the functional units of a single processor
  - Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
  - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
  - The memory can be shared through virtual memory mechanisms
  - Hardware must support *efficient* thread context switching

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT (*Hyper-Threading*)
  - Two threads: duplicated registers, shared function units and caches

Threading on a 4-way SS Processor Example



- Will it survive? In what form?
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Review: Multiprocessor Basics

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors?

			# of Proc
Communication model	Message passing		8 to 2048
	Shared address	NUMA	8 to 256
		UMA	2 to 64
Physical connection	Network		8 to 256
	Bus		2 to 36

Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, …
4. Topologias de interligação

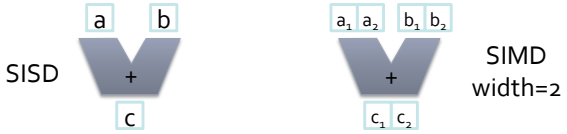
Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

SIMD



- Single Instruction Multiple Data architectures make use of data parallelism
- SIMD can be area and power efficient
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

- Operate element-wise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector Code Example

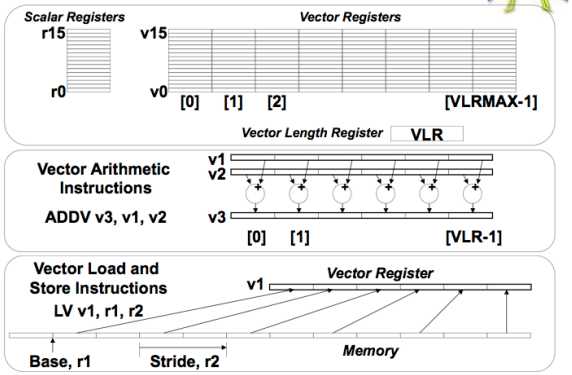
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# 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
```

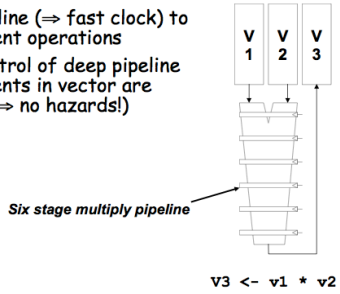
- Require programmer (or compiler) to identify parallelism
  - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model



Vector Arithmetic Execution

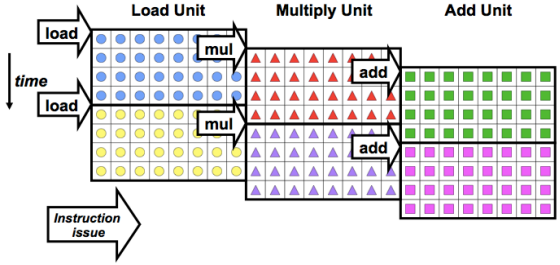
- Use deep pipeline (⇒ fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (⇒ no hazards!)



Vector Instruction Parallelism

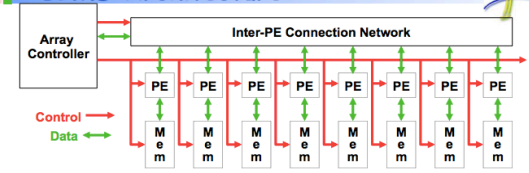
Can overlap execution of multiple vector instructions

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



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

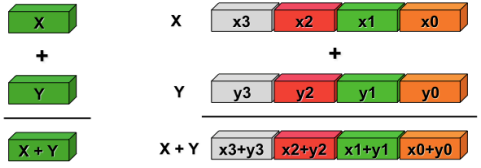
SIMD Architecture



- 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?)

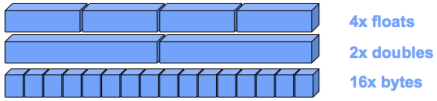
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

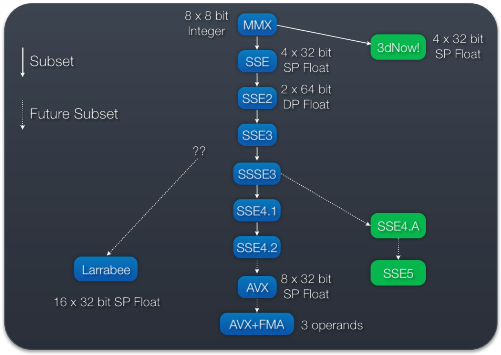
E.g.: SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,

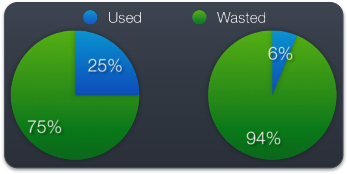


- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data from one part of register to another
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help

A Brief History of x86 SIMD



What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (LRB)

- Neglecting SIMD in the future will be more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

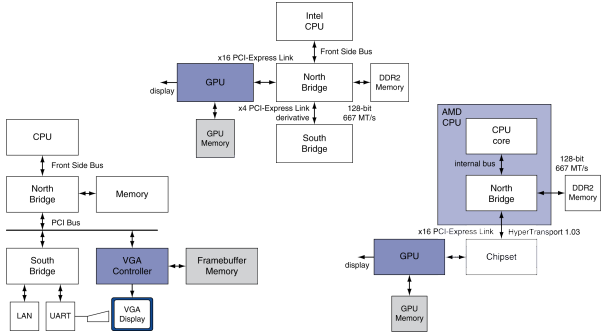
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law ⇒ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System



Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - 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	**Tesla Multiprocessor**

GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

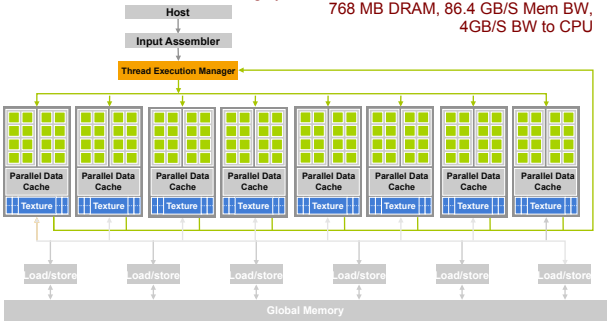
Massive thread level parallelism

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

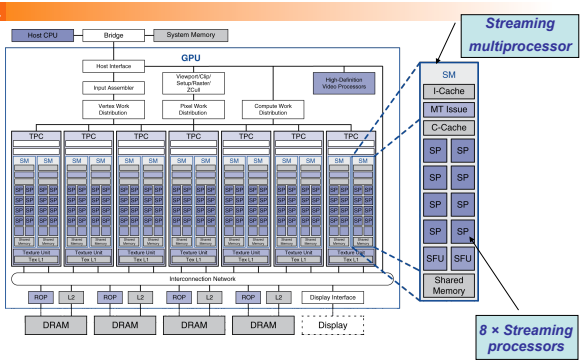
- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Example: NVidia GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU

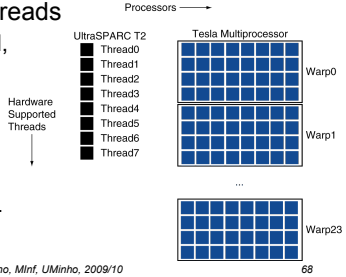


Example: NVIDIA Tesla



Example: NVIDIA Tesla

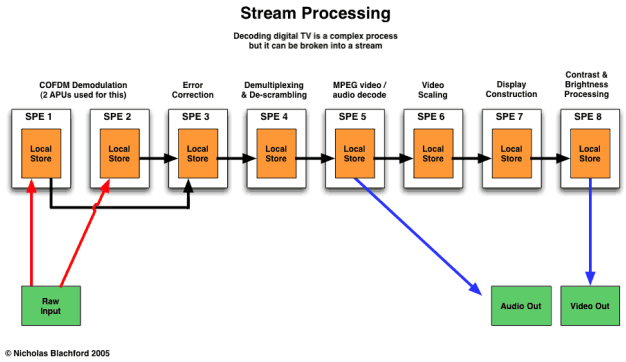
- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …



The PS3 "Cell" Processor Architecture

- Composed of a non-SMP architecture
  - 234M transistors @ 4Ghz
  - 1 Power Processing Element (PPE) "control" processor. The PPE is similar to a Xenon core
    - Slight ISA differences, and fine-grained MT instead of real SMT
- And 8 "Synergistic" (SIMD) Processing Elements (SPEs)
  - An attempt to 'fix' the memory latency problem by giving each SPE complete control over it's own 256KB "scratchpad" memory
    - Direct mapped for low latency
    - 4 vector units per SPE, 1 of everything else – 7M transistors
  - 512KB L2$ and a massively high bandwidth (200GB/s) processor -memory bus

How to make use of the SPEs



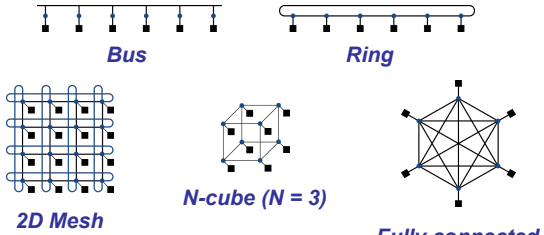
Arquitecturas Paralelas

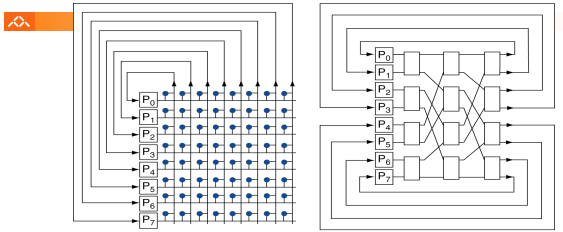
Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, …
4. Topologias de interligação

Multiprocessor Network Topologies: Interconnection Networks

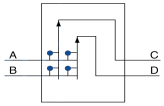
- Network topologies
  - Arrangements of processors, switches, and links





a. Crossbar

b. Omega network



c. Omega network switch box

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface
4 FOURTH EDITION

Hennessy & Patterson

Chapter 7

Multicores, Multiprocessors, and Clusters

Basic Reference for this set of slides