Advanced Processors

Instruction-Level Parallelism (ILP)

Multiple Issue

Fetch multiple (e.g. 2, 4) instructions in parallel, and then consider how many and which of them to execute in parallel

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards where none independent available, fills-in noop’s
- Dynamic multiple issue
  - 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

Static Multiple Issue

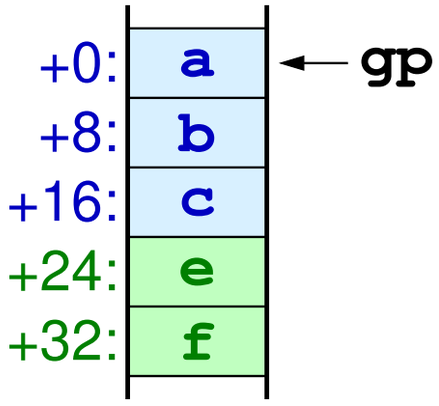
- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with in a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

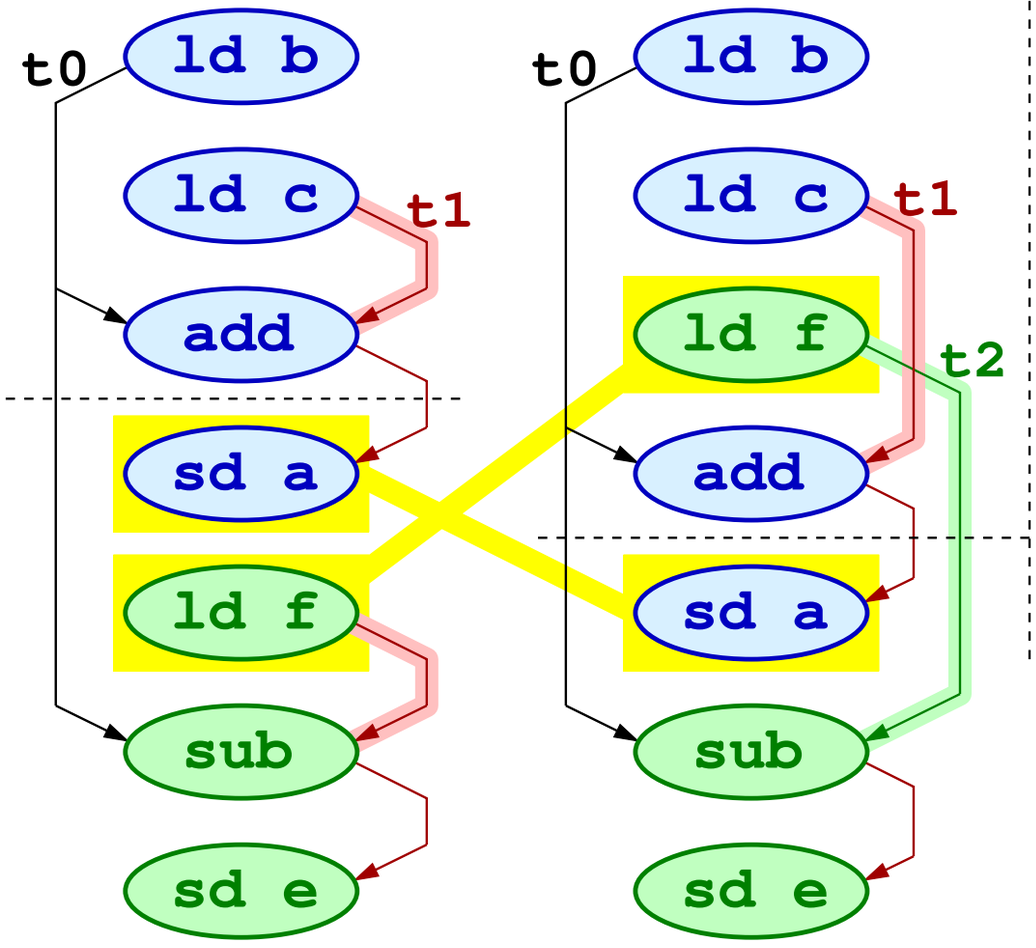
Instruction Scheduling

```
a = b + c;
e = b - f;
```




```
ld  t0,  8(gp)
ld  t1, 16(gp)
add t1, t0, t1
sd  t1,  0(gp)
ld  t1, 32(gp)
sub t1, t0, t1
sd  t1, 24(gp)
```

2 extra clock cycles lost



This is 'Static' Scheduling, at Compile Time

the more things you have 'up in the air' (in parallel), the more temporary registers you need in order to 'name' those 'pending' values

```
ld  t0,  8(gp)
ld  t1, 16(gp)
ld  t2, 32(gp)
add t1, t0, t1
sd  t1,  0(gp)
sub t1, t0, t2
sd  t1, 24(gp)
```

No extra clock cycle lost

What if the program is?:
RAW dependence?

```
a[i] = b + c;
e = b - a[j];
```

- Does the compiler know for sure if i!=j (OK to reorder sd–ld) or i==j (fwd in reg.)?
- If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle (checks dependencies and)
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Allows executables to run on newer processors, with same ISA but different pipeline, without needing to be recompiled

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - Can start sub while add is waiting for ld

Out-of-Order (ooo) Execution

In-Order Commit

(so as to flush results of mis-speculated instructions, and also allow precise exceptions)

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Parallelism

Pitfall: Amdahl's Law

Επιτάχυνση ενός μέρους μόνον του προγράμματος και όχι ολόκληρου

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

e.g. parallelizable

e.g. non-parallelizable

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

e.g. available parallelism

80 seconds out of total 100 seconds

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = (10 + 100) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Instruction and Data Streams

- An alternate classification

Multimedia (e.g. MMX)
Streaming SIMD Extension (SSE)

		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

Vector

often

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

SIMD

- Operate elementwise 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

Vector Processors

Data-Level Parallelism
Identical & Independent operations on all elements of a vector (array) - one vector instr. replaces a loop

- 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 RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
  - Vector instructions
    - `fld.v`, `fsd.v`: load/store vector
    - `fadd.d.v`: add vectors of double
    - `fadd.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector Length Register assists in counting the number of remaining elements to process

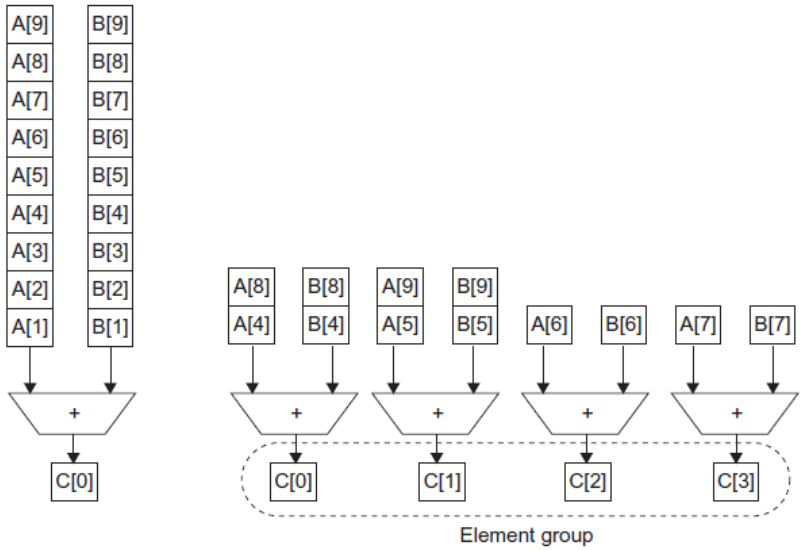
Sequential memory addresses, or strided (e.g. for 2D/3D arrays), or scatter-gather (via array of pointers

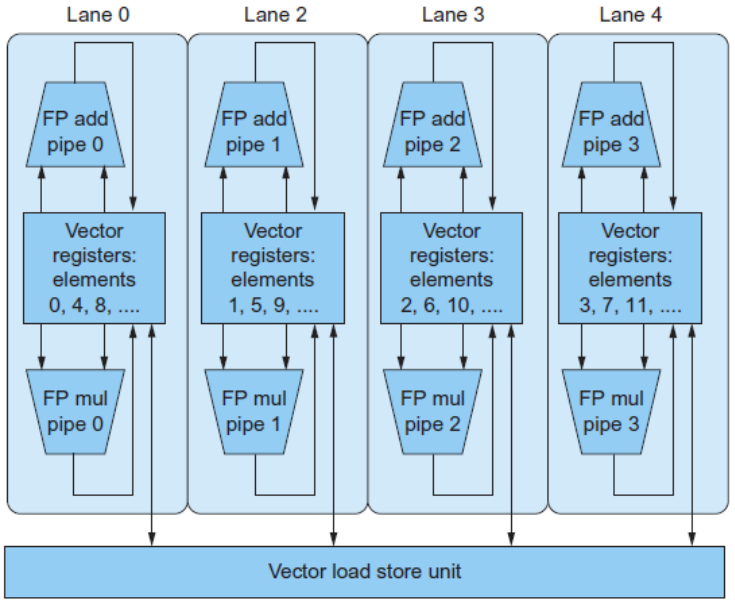
Vector vs. Scalar

- 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

Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width (length), multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:





Multithreading

One "thread of control" = one (traditional) sequential program.
Multiple threads = parallel program.

- 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)

mimic multiple cores, thus:

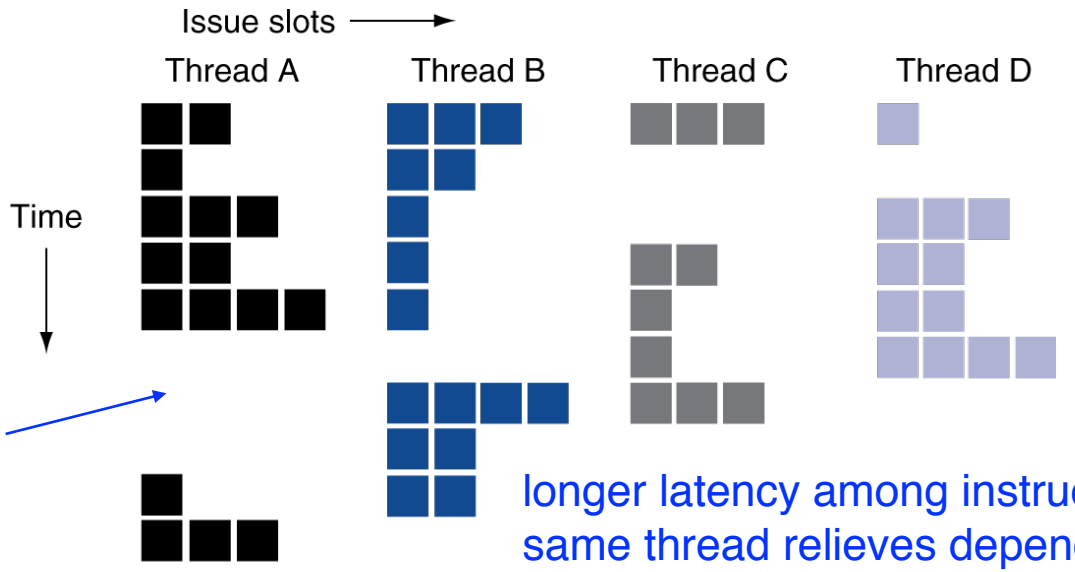
but Share the Functional Units and the Caches

as if hardware support for fast switching among processes

Simultaneous Multithreading

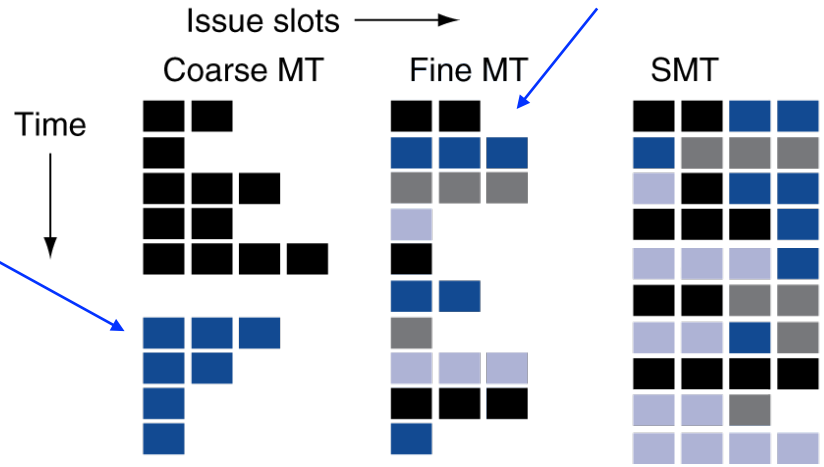
- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: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

Multithreading Example



Cache miss stalls are major concern; coarse MT targets them especially

longer latency among instructions of a same thread relieves dependencies, but slows down each individual thread



Future of Multithreading

- 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

Two different threads may have two different working sets of data/instructions; is it better to place them in a single cache, or in two different caches as two separate cores would do?

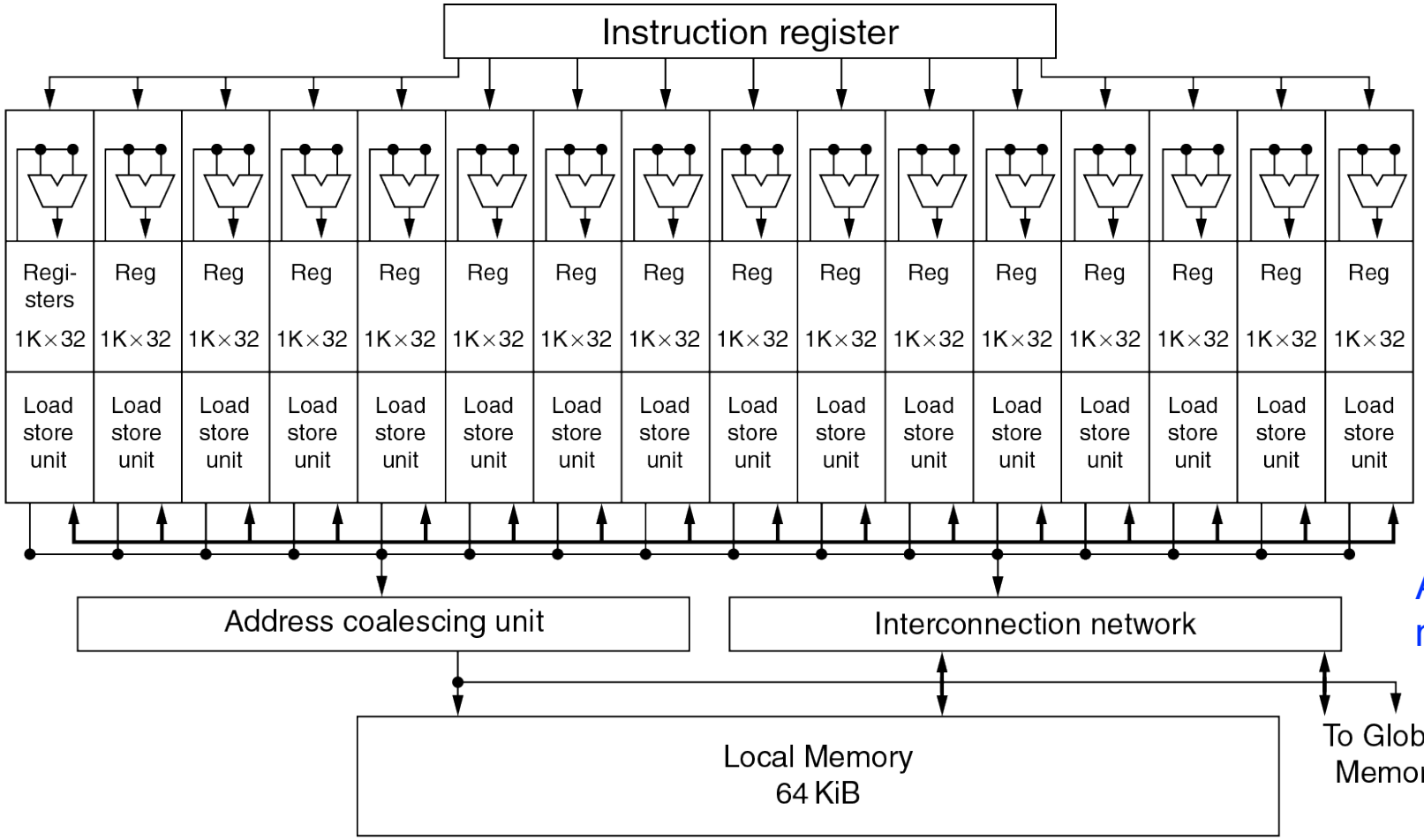
GPU Architectures

Graphics Processing Units

- 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)

Example: NVIDIA Fermi

- Multiple SIMD processors, each as shown:



16 lanes

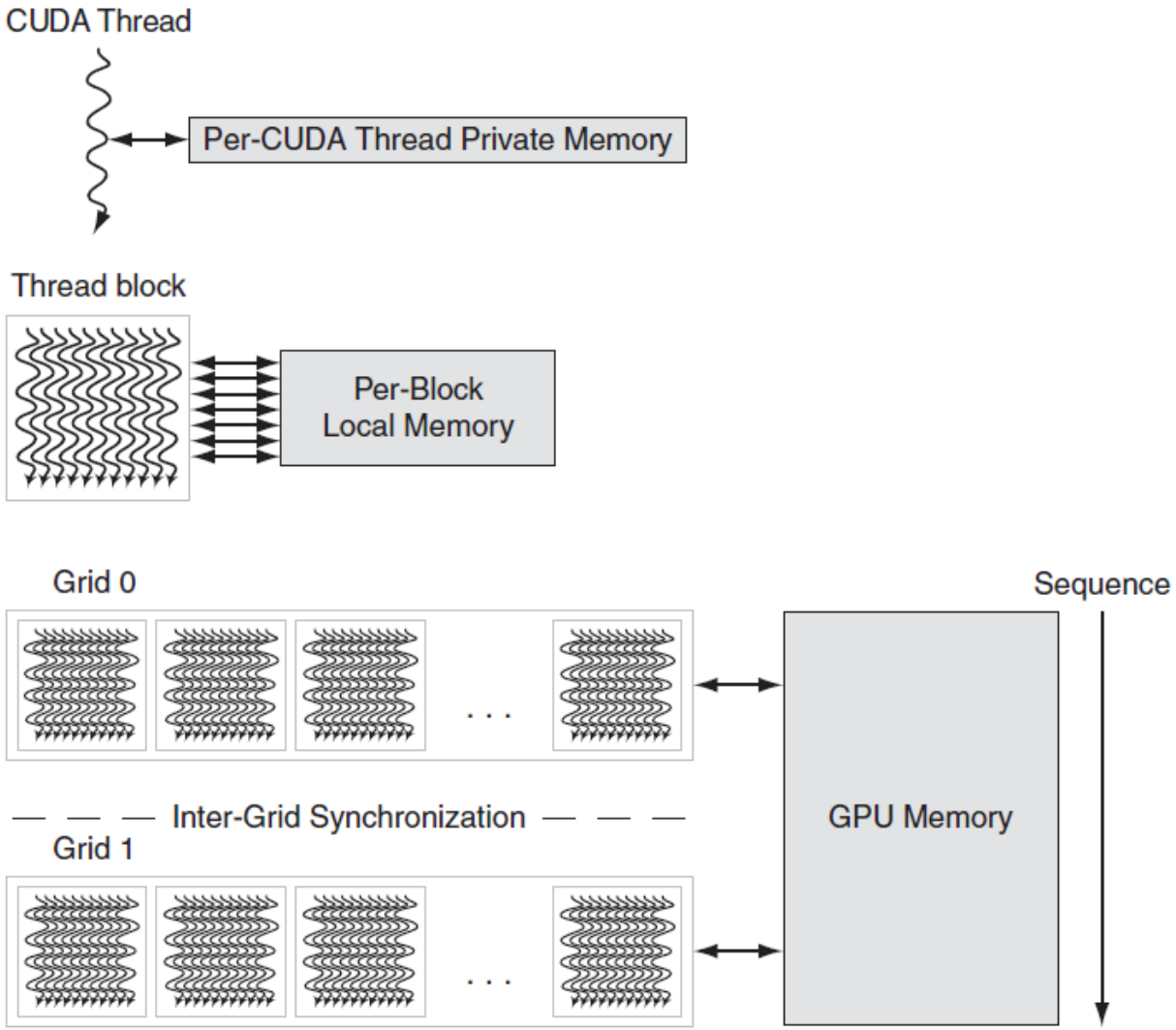
32-element SIMD instructions

Also, massively multithreaded

Example: NVIDIA Fermi

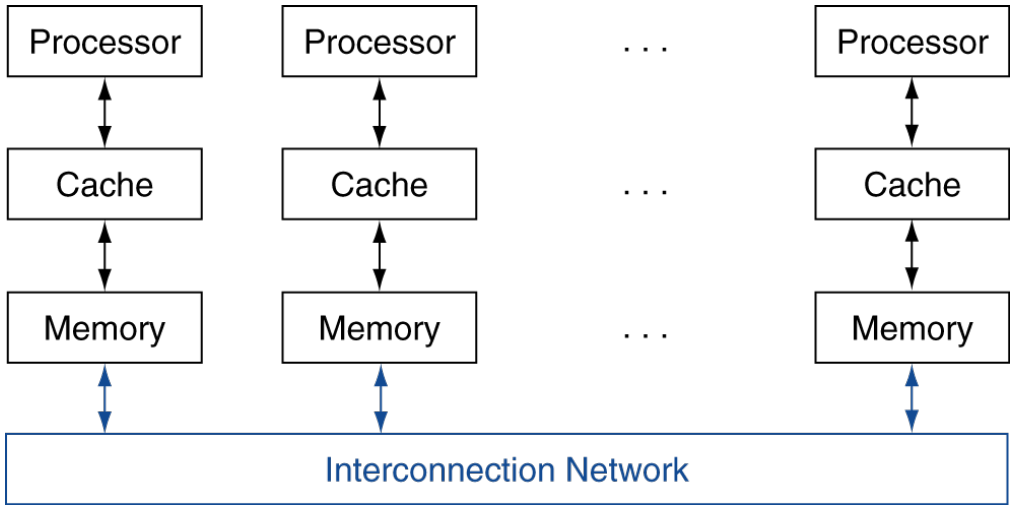
- SIMD Processor: 16 SIMD lanes
- SIMD instruction
  - Operates on 32 element wide threads
  - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
  - 64 registers per thread context

GPU Memory Structures



Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP