Topic 7: Support for parallelism

Lars Karlsson

based on slides by
Mary Jane Irwin
and Stephen Hegner

2012-12-14

Parallelism at multiple levels

- **Job/process level parallelism** Inreased throughput by running multiple independent jobs/processes in parallel.
- **Thread level parallelism (aka TLP)** Threads cooperating to speed up a single program—*parallel processing*.
- **Instruction level parallelism (aka ILP)** Multiple instructions (in the same thread) executing in parallel (e.g., via pipelining).

Multicores now common

- **The power wall** forced a change in the design of microprocessors
  - Before 2002 the single-thread performance increased by approx **50%** per year
  - Today the increase is closer to **20%** per year
- **CMP – Chip Multicore microProcessor** Contains more than one **core** per integrated circuit.
  - Tens or fewer cores in high-end processors today.
  - Many tens or hundreds of cores in specialized accelerators/coprocessors (GPUs, AMD APUs, Intel MIC, etc).
  - Rapid scaling of the number of cores.
  - Simpler cores draw less power, take less die area.

Clusters (of multicores) are also common

- **Cluster** – a set of independent servers connected over a local area network (LAN) functioning as a single large multiprocessor
  - Search engines
  - Web servers
  - Online multiplayer games
  - Databases
  - Scientific simulations
  - *etc*
- Emphasis on **high-speed networks**
  - **Operating system bypass** Send and receive messages without the overhead of invoking the operating system.
  - **Independent progress** Enable network communication in parallel with computation.

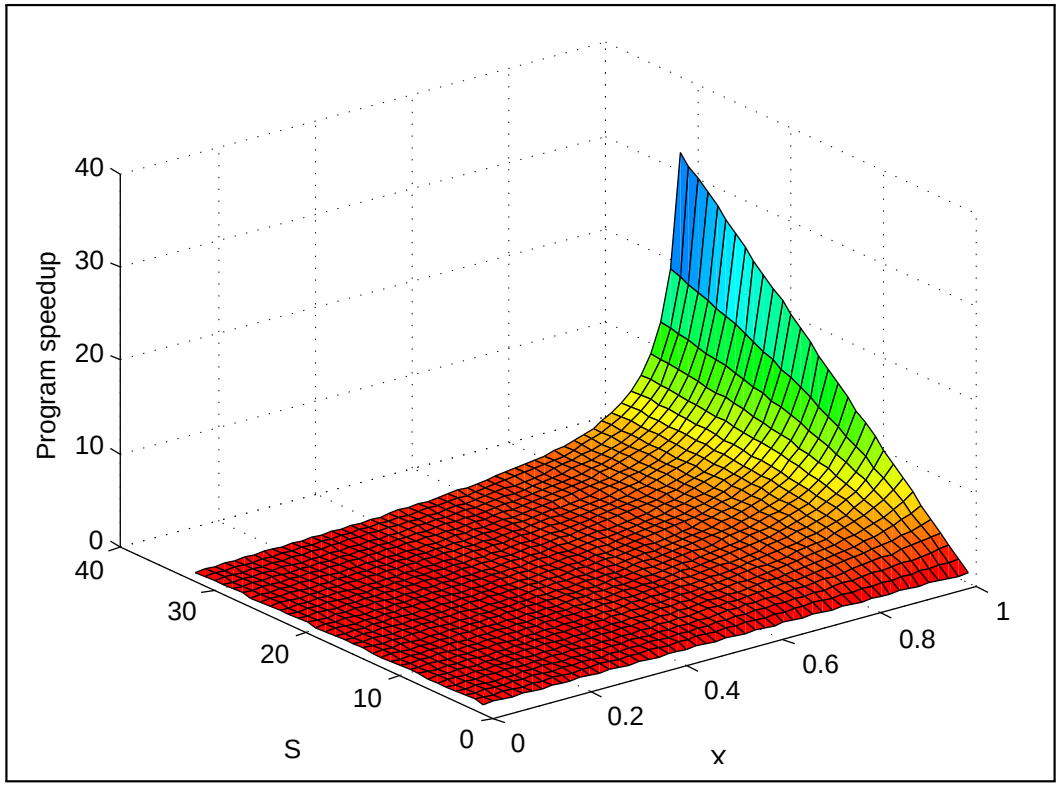
Scalability

- The main difficulty in using multiprocessors is to construct parallel programs that maintain high performance as the number of processors increases—i.e., writing **scalable** parallel programs
- **Challenges:**
  - Scheduling
  - Load balancing
  - Synchronization overhead
  - Communication overhead (network and memory)
  - *etc*

Amdahl's Law

- If the execution time of a fraction X of a program is reduced by a factor (speed up) of S, then the execution time of the entire program is only reduced by the factor

$$\frac{1}{(1 - X) + X/S}$$



Amdahl's Law: Examples

- A 20x speedup ($\boldsymbol{S = 20}$) usable $\boldsymbol{25\%}$ ($\boldsymbol{X = 0.25}$) of the time:

$$\text{speedup} = \frac{\mathbf{1}}{\mathbf{0.75 + 0.25/20}} = \mathbf{1.311}$$

- To get a speedup of $\mathbf{90}$ from $\mathbf{100}$ processors, the percentage of the original program that could be sequential would have to be $\mathbf{0.1\%}$ or less:

$$\text{speedup} = \frac{\mathbf{1}}{\mathbf{0.001 + 0.999/100}} = \mathbf{90.992}$$

Scalability and load balancing

- It is harder to get good speedup when the problem size is fixed than getting good speedup when the size of the problem is increased.
  - **Strong scaling** When speedup is achieved on fixed problem sizes.
  - **Weak scaling** When speedup is achieved on problem sizes increasing proportionally to the increase in the number of processors.
- **Load balancing** is important. Just a single processor with twice the load of the others cuts the speedup almost in half

Shared memory multiprocessor

- **Single address space** shared by all processors
  - **UMA (aka SMP)** Uniform Memory Access
  - **ccNUMA** Cache-Coherent Non-Uniform Memory Access. More scalable, but harder to program.
- Processors **communicate through shared variables** in memory (via loads, stores, and atomic read-modify-write operations)
- **Cache coherency** is a big deal and requires HW support

Example: Sum 100,000 numbers on 100 processors

```
1:  // Local reduction phase
2:  sum[Pn] = 0;
3:  for i = 1000*Pn; i < 1000*(Pn+1); ++i do
4:     sum[Pn] = sum[Pn] + A[i];
5:  end for
6:  // Global reduction phase
7:  half = Pn;
8:  while half > 1 do
9:     synch(); // Synchronize
10:    if half % 2 != 0 && Pn == 0 then
11:       sum[0] = sum[0] + sum[half-1];
12:    end if
13:    half = half / 2;
14:    if Pn < half then
15:       sum[Pn] = sum[Pn] + sum[Pn+half];
16:    end if
17: end while
```

Process synchronization

- Need to coordinate processes working together
- **Synchronization primitives** (mutexes/locks, semaphores, condition variables, barriers, etc) are used to coordinate or synchronize processes
- Need **architecture-supported atomic operations** (beyond load and store) to efficiently implement multiprocessor synchronization
- In MIPS, we have `ll` and `sc` that enable construction of arbitrary **atomic read-modify-write operations**. Common special cases:
  - **Compare-and-swap (CAS)**
  - **Test-and-set**

Spin lock implementation

Lock:

1: *// Shared* `lock` *variable initialized to* **0**
2: **repeat**
3: **repeat**
4: Read `lock` using `ll` (load-linked)
5: **until** read **0** (unlocked)
6: *// Other processors may acquire the lock before we do*
7: Try to set `lock` to **1** using `sc` (store-conditional). This step succeeds only if no other processor acquires the lock since our last `ll`.
8: **until** `sc` succeeds
9: *// We have now acquired the lock*

Unlock:

1: Set `lock` to **0**

Spin lock scalability

- The spin lock implementation has **poor scalability** when the lock is heavily **contended** (i.e., when many processors simultaneously try to acquire the lock)
- When the lock is released, all spinning processors simultaneously try to fetch the new lock value, which results in a lot of communication.
- Many sophisticated and scalable algorithms exist

Cache coherence

- Multiprocessors have **multiple memory hierarchies**
- **Multiple copies of the same variable**
  - OK as long as the variable is not modified
  - Problematic when modifications occur—need *cache coherence protocol* to maintain a coherent view of memory
- **Intuition:** A **read** to a location should return the value written by the **last write** to that location
  - OK for uniprocessors
  - But the concept of **last** is not well-defined on multiprocessors!

Necessary cache coherence features

- **Write propagation** Written value must *eventually* become visible to other processors (allowed to take some time)
- **Write serialization** Writes to a location are seen in the same order by all processors

Handling writes: Update versus invalidate

Two alternative strategies:

- **Write-update** A modification of a shared variable **updates** all copies of the variable directly without going through the memory.
  - Advantage—the new value is quickly propagated
  - Disadvantage—unnecessary coherence traffic if the new value is not used by the other processors (see also false sharing below)

- **Write-invalidate** A modification of a shared variabe **invalidates** all copies of the variable. Other processors obtain the new value via a cache miss on the next access of the shared variable.
  - Advantage—less coherence traffic
  - Disadvantage—longer latency when obtaining the new value

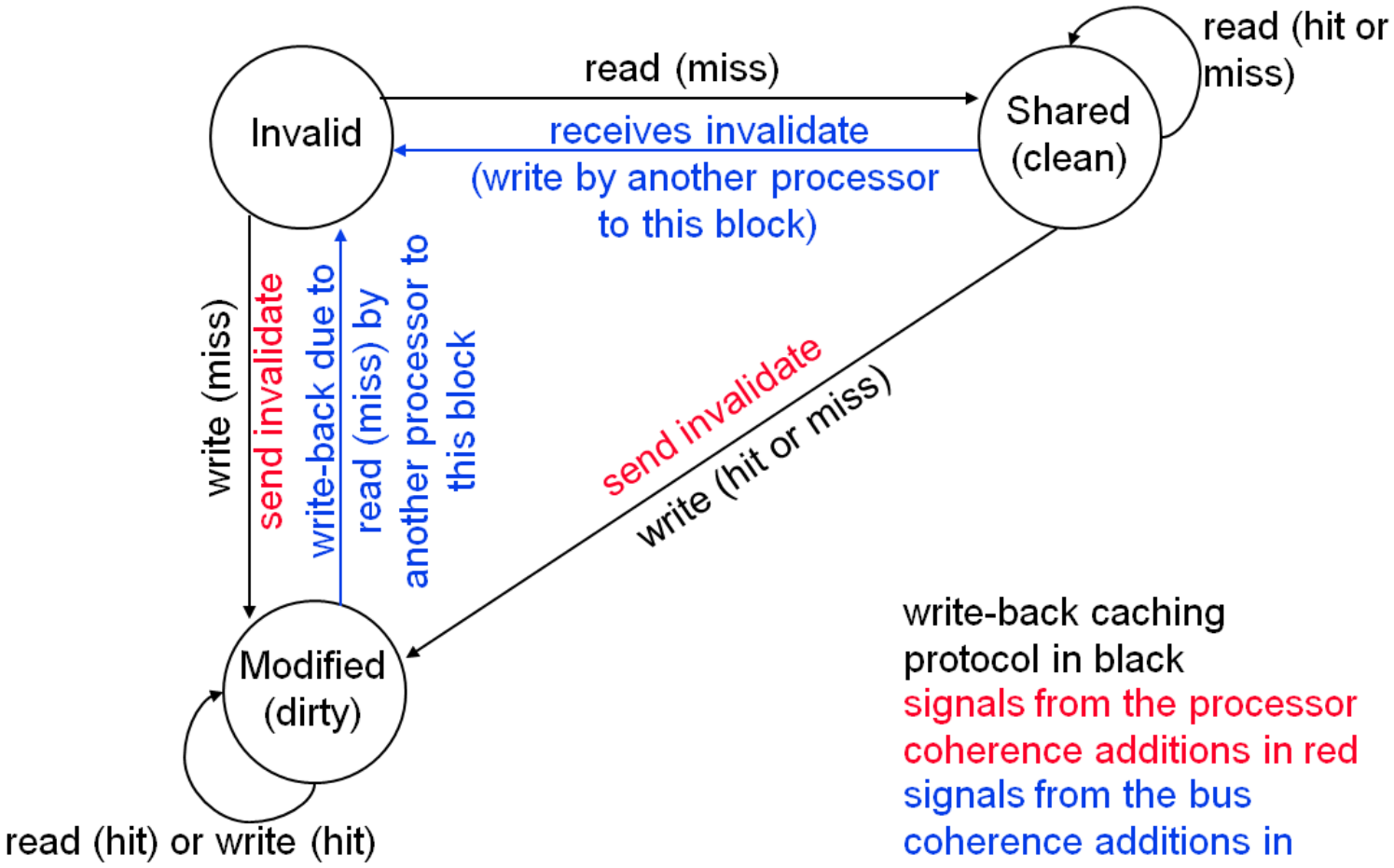
Cache coherence granularity and false sharing

- Cache coherence typically operates on the level of **cache blocks**
- **False sharing** When two processors write to **different** (non-shared) variables located in **the same** cache block. The cache coherence implementation do not make this distinction and must assume sharing.
  - See the earlier summation example, where false sharing does occur

Types of cache coherence protocols

- **Snooping-based cache coherence** Relies on a shared medium (bus). Processors listens to the bus traffic to detect reads and writes of shared data. Easier to implement, but less scalable.
- **Directory-based cache coherence** Each block has a *home node* (typically the node where the block is physically located) that keeps track of its sharing state. Harder to implement, but more scalable. Can use scalable interconnect instead of a bus.

A write-invalidate CC protocol (MSI)



More sophisticated CC protocols

- More sophisticated protocols are used in practice
- **MESI** splits the shared state into a new shared state and an *exclusive* state. Writes in the exclusive state do not trigger invalidations.
- **MOESI** adds an **owned** state similar to the shared state, but the memory version might be stale and the owner has the responsibility of updating the memory eventually

Memory models

- Cache coherence protocols enforce a consistency model for accesses **to the same location**
- But we also need models for accesses to **different locations**
- Such models are called **memory models** and are specified at different levels
  - **Processors** specify a memory model at the machine instruction level
  - **Programming languages** specify a memory model at the source code level
- Until recently, popular programming languages such as **Java**, **C**, and **C++** did *not* specify a memory model

Program order

In **uniprocessors**, the system guarantees that the statements of a program **appear to** execute in **program order**.

- In reality, program order execution is **an illusion**
- The truth is that both the compiler and the processor aggressively transform and **reorder** the statements and instructions of the program
- These reorderings are made for **performance reasons**

Sequential consistency

Generalization of program order to multiprocessors

In **multiprocessors**, a natural generalization of program order is called **sequential consistency**. A sequentially consistent system guarantees that a parallel program **appears to** execute as if the instructions of all the processors were executed in **some sequential order** that preserves the program order of each processor.

- In reality, sequential consistency is **much too slow**

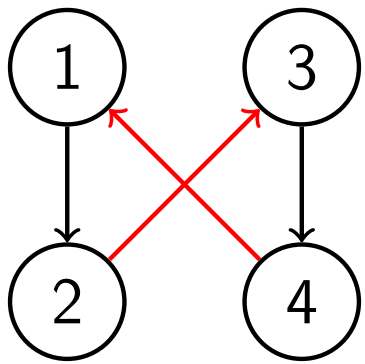
Reordering: Surprising results

```
Initialize global variables A = B = 0
```

Thread 1	**Thread 2**
`1: r2 = A`	`3: r1 = B`
`2: B = 1`	`4: A = 2`

Is it possible to get `r1 = 1` and `r2 = 2` after execution?

- To get `r1 = 1`, stmt 3 must execute after stmt 2
- To get `r2 = 2`, stmt 1 must execute after stmt 4
- Impossible (if sequentially consistent)!



The Java memory model

- In Java, the above program is said to be **incorrectly synchronized** since there is a **data race** in the sense that
  - There is a write to a variable (e.g., `A = 2`)
  - There is a read of that same variable (e.g., `r2 = A`)
  - And the write and the read are not ordered by synchronization
- **Synchronizations** via, e.g., **locks** or **reads/writes of volatile variables** are necessary to eliminate the race condition

Hardware multithreading

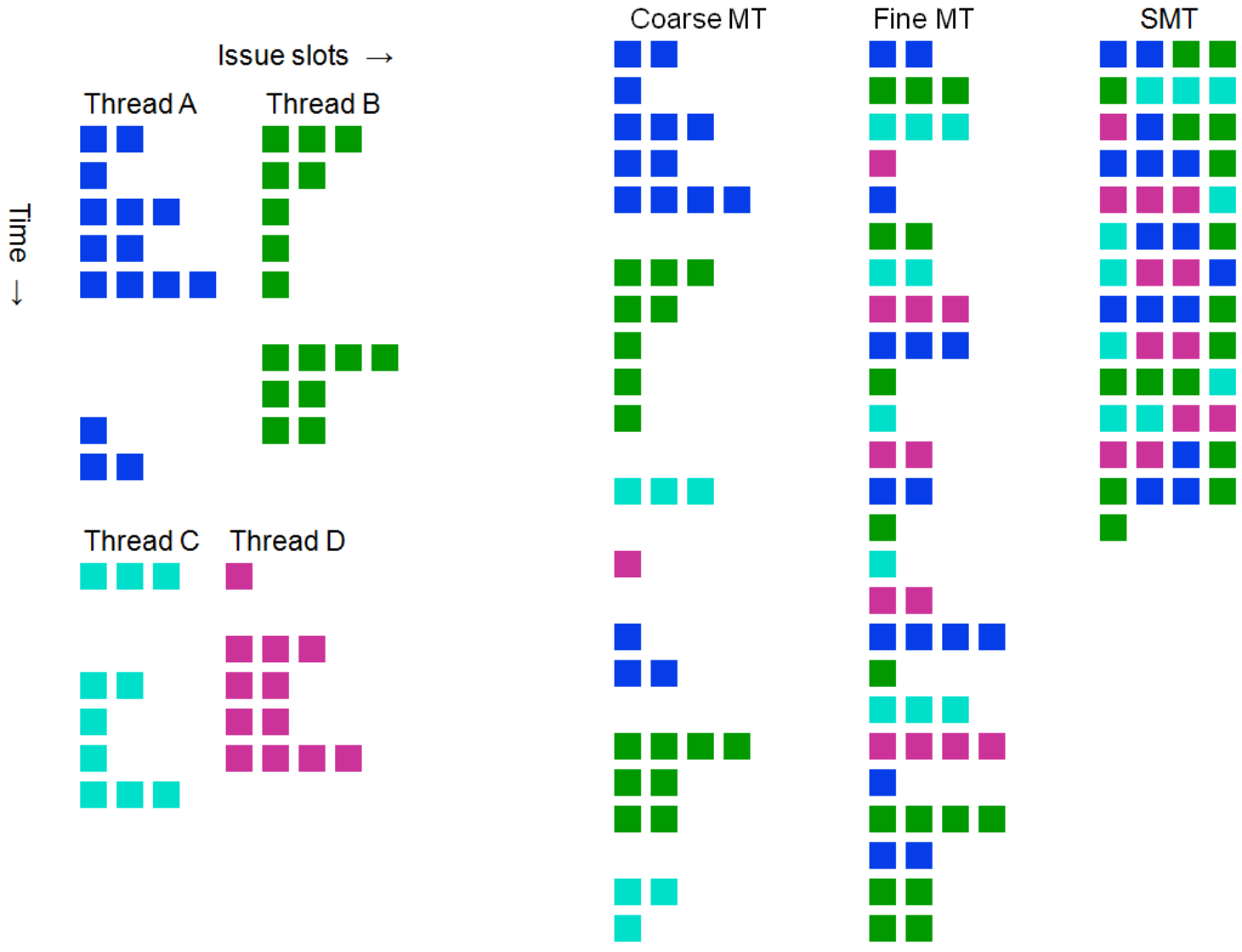
- Hides data dependency stalls, cache miss stalls, and branch stalls by finding and executing **instructions** independent of the stalling instructions
- Allows multiple processes (**threads**) to share the functional units of a single processor
  - Must duplicate state hardware for each thread:
    - Separate register file,
    - program counter (PC),
    - instruction buffer,
    - store buffer, *etc*
  - The caches, TLBs, *etc* can be shared although the **miss rates can increase** due to interference
  - Hardware must support **efficient thread context switching**

Types of multithreading

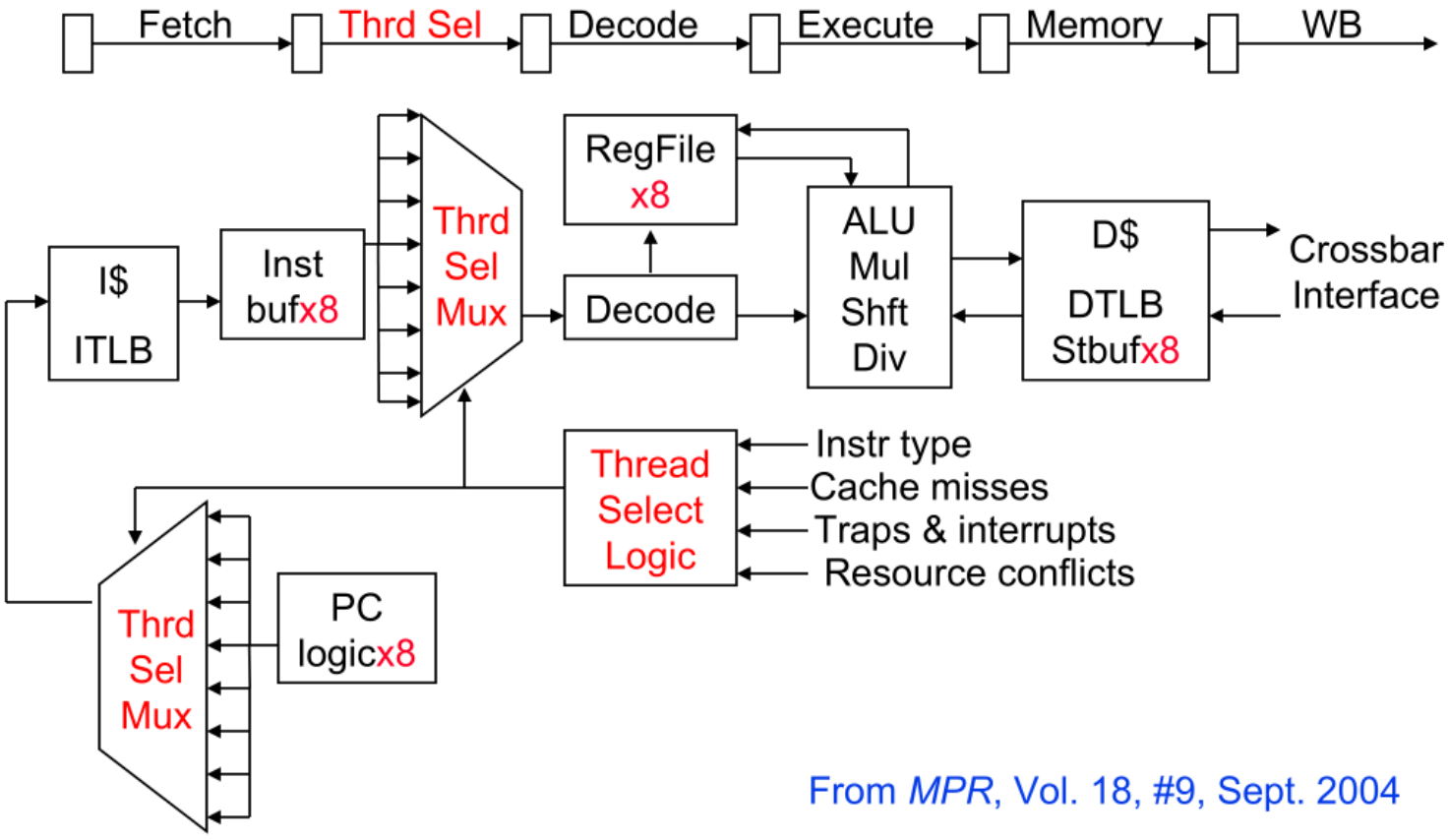
- **Fine-grain** Switch threads on every instruction issue
  - Round-robin thread interleaving (skipping stalled threads)
  - Processor must be able to switch threads every clock cycle
  - Advantage—can hide throughput losses from both short and long stalls
  - Disadvantage—slows down the execution of an individual thread since it may be delayed by other threads even if not currently stalled

- **Coarse-grain** Switch threads only on costly stalls
  - Advantages—thread switching doesn't have to be essentially free, and less likely to slow down an individual thread
  - Disadvantages—limited, due to pipeline start-up costs, in its ability to overcome throughput loss (pipeline must be flushed and refilled on thread switches)

Multithreading on a 4-way SS processor



Niagara integer pipeline



From *MPR*, Vol. 18, #9, Sept. 2004

Single Instruction Multiple Data (SIMD)

- SIMD computers have **multiple datapaths** sharing the same **control logic**
- SIMD is common today in the form of **short vector instructions**
  - MMX
  - SSE
  - AVX
  - AltiVec
  - *etc*
- AVX has 256-bit SIMD registers. Each SIMD instruction operates in parallel on one of
  - 4 double-words (`double`, `long`)
  - 8 words (`float`, `int`)
  - 16 half-words (`short`)
  - 32 bytes (`char`)

VMIPS daxpy code

```
l.d      $f0, a($sp)     ;load scalar a
lv       $v1, 0($s0)     ;load vector X
mulvs.d  $v2, $v1, $f0   ;vector-scalar multiply
lv       $v3, 0($s1)     ;load vector Y
addv.d   $v4, $v2, $v3   ;add Y to a * X
sv       $v4, 0($s1)     ;store vector result
```

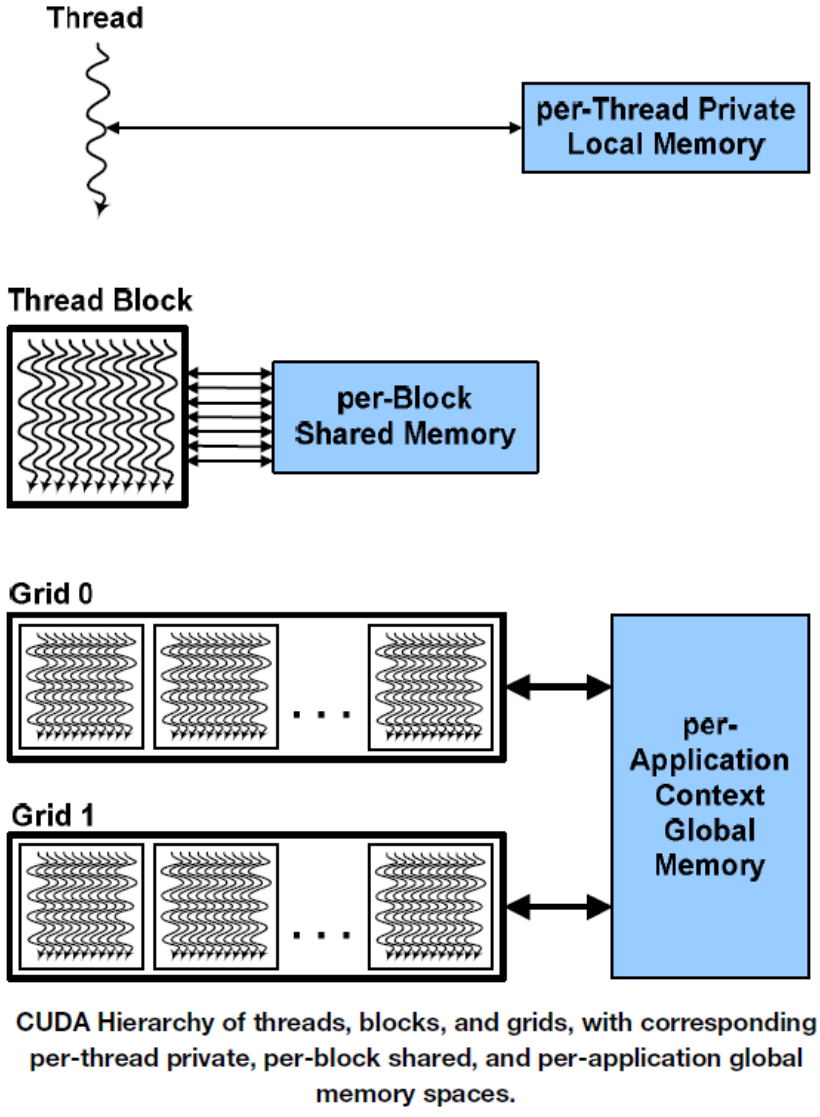
Graphics Processing Units (GPUs)

- Highly parallel
- Many simple cores
- Highly multithreaded
- Highly parallel memory system
- Has SIMD features
- Novel ISAs without backward compatibility woes

Brief history of GPUs)

- First introduced in 1999
- GPGPU programming via graphics APIs since 2003
- NVIDIA CUDA and GPU Computing since 2006
- Now hundreds of cores per die

CUDA Thread hierarchy



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Image source: "NVIDIA Next Generation CUDA Compute Architecture: Fermi", NVIDIA whitepaper

Fermi streaming multiprocessor (SM)

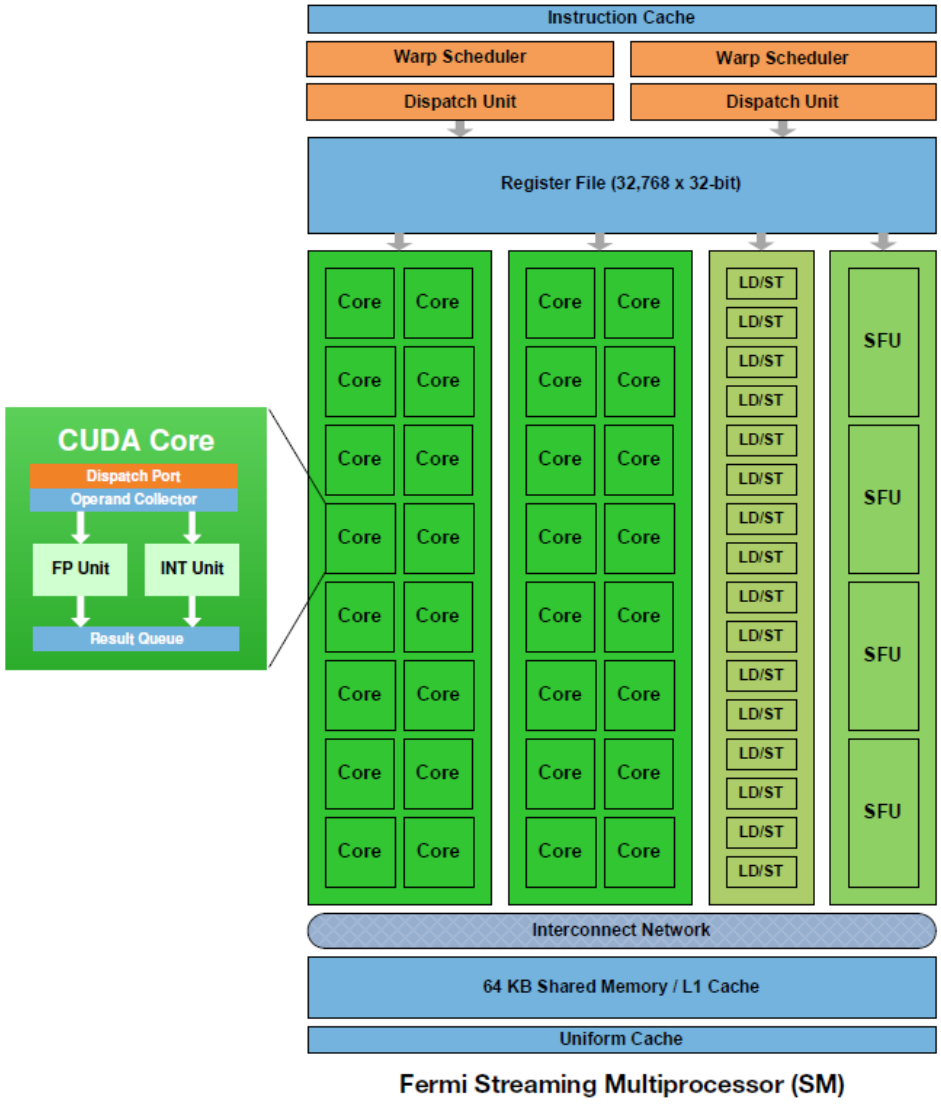


Image source: "NVIDIA Next Generation CUDA Compute Architecture: Fermi", NVIDIA whitepaper

Dual warp schedulers

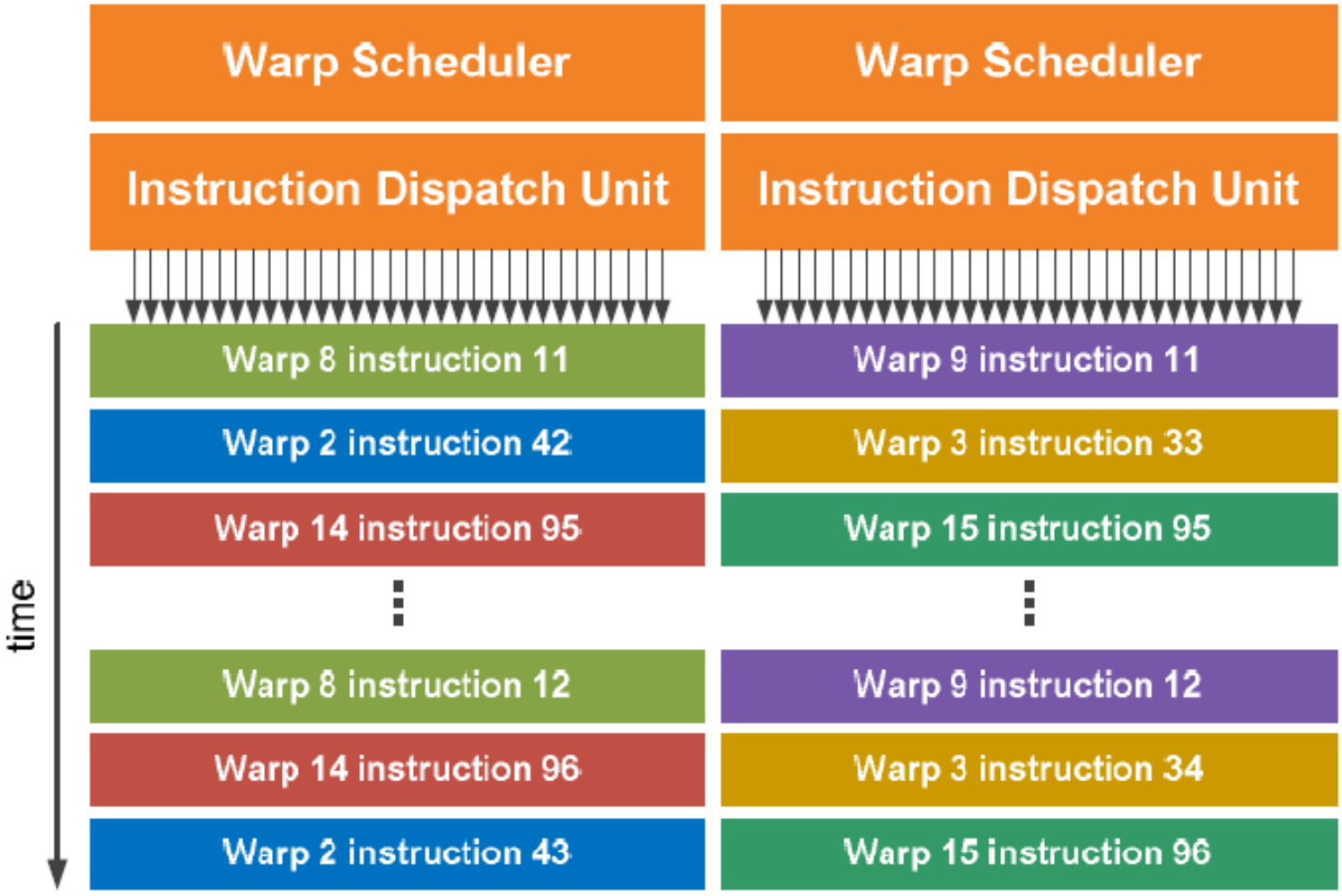


Image source: "NVIDIA Next Generation CUDA Compute Architecture: Fermi", NVIDIA whitepaper

Summary table

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Image source: "NVIDIA Next Generation CUDA Compute Architecture: Fermi", NVIDIA whitepaper