



Computer Hardware Engineering (IS1200)

Computer Organization and Components (IS1500)

Fall 2020

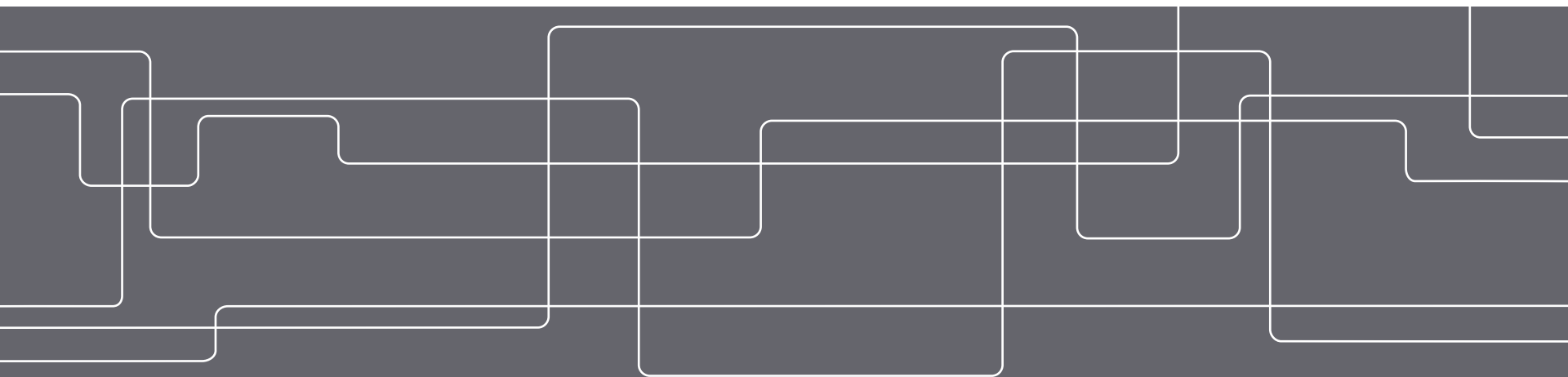
Lecture 13: SIMD, MIMD, and Parallel Programming

(We start 08:15)

Artur Podobas

Researcher, KTH Royal Institute of Technology

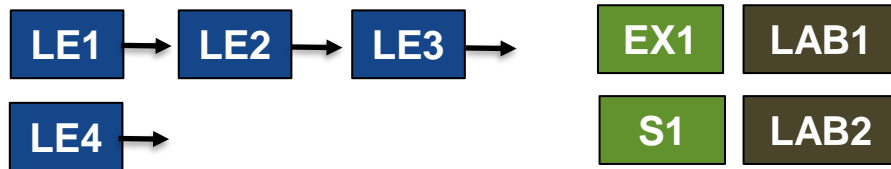
Slides by David Broman, KTH (Extensions by Artur Podobas)



Course Structure



Module 1: C and Assembly Programming

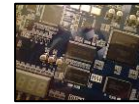


Module 2: I/O Systems



Module 3: Logic Design (IS1500 only)

**PROJ
START**



Module 4: Processor Design



Module 5: Memory Hierarchy



Module 6: Parallel Processors and Programs

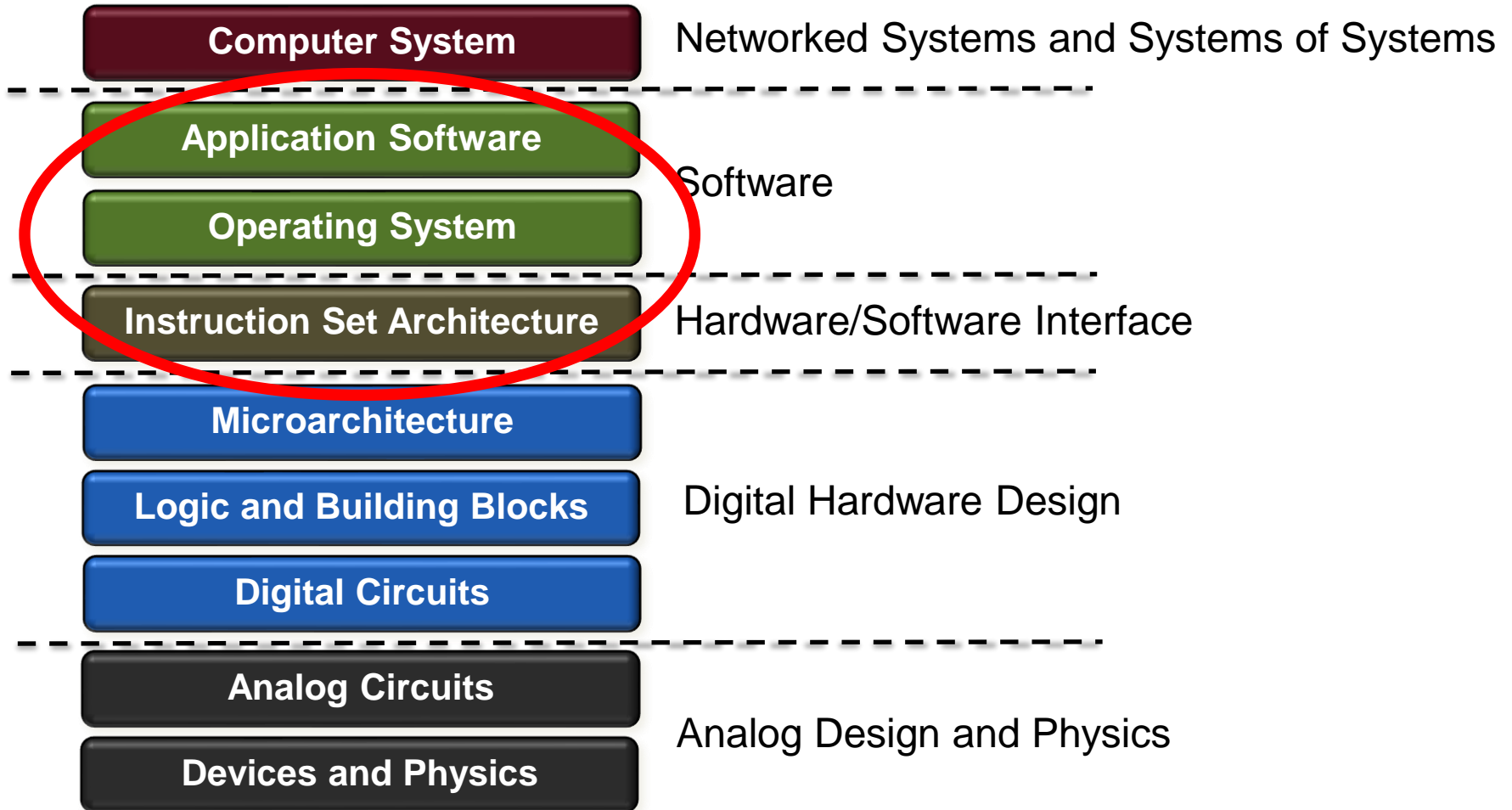


Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

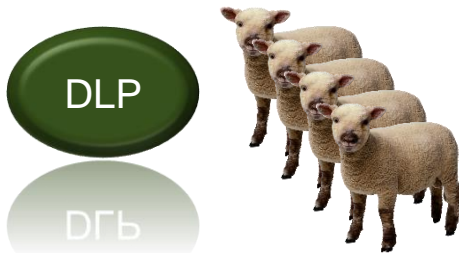
Abstractions in Computer Systems



Agenda

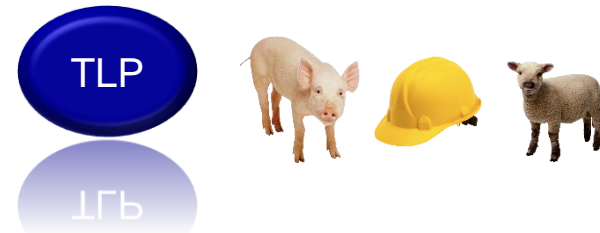
Part I

SIMD, Multithreading, and GPUs



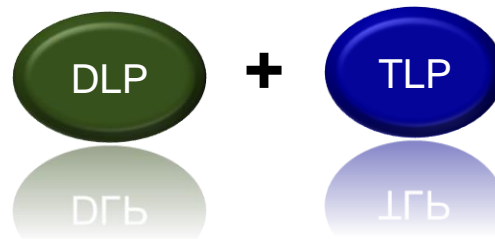
Part II

MIMD, Multicore, and Clusters



Part III

Parallelization in Practice



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Part I

SIMD, Multithreading, and GPUs



Acknowledgement: The structure and several of the good examples are derived from the book “Computer Organization and Design” (2014) by David A. Patterson and John L. Hennessy

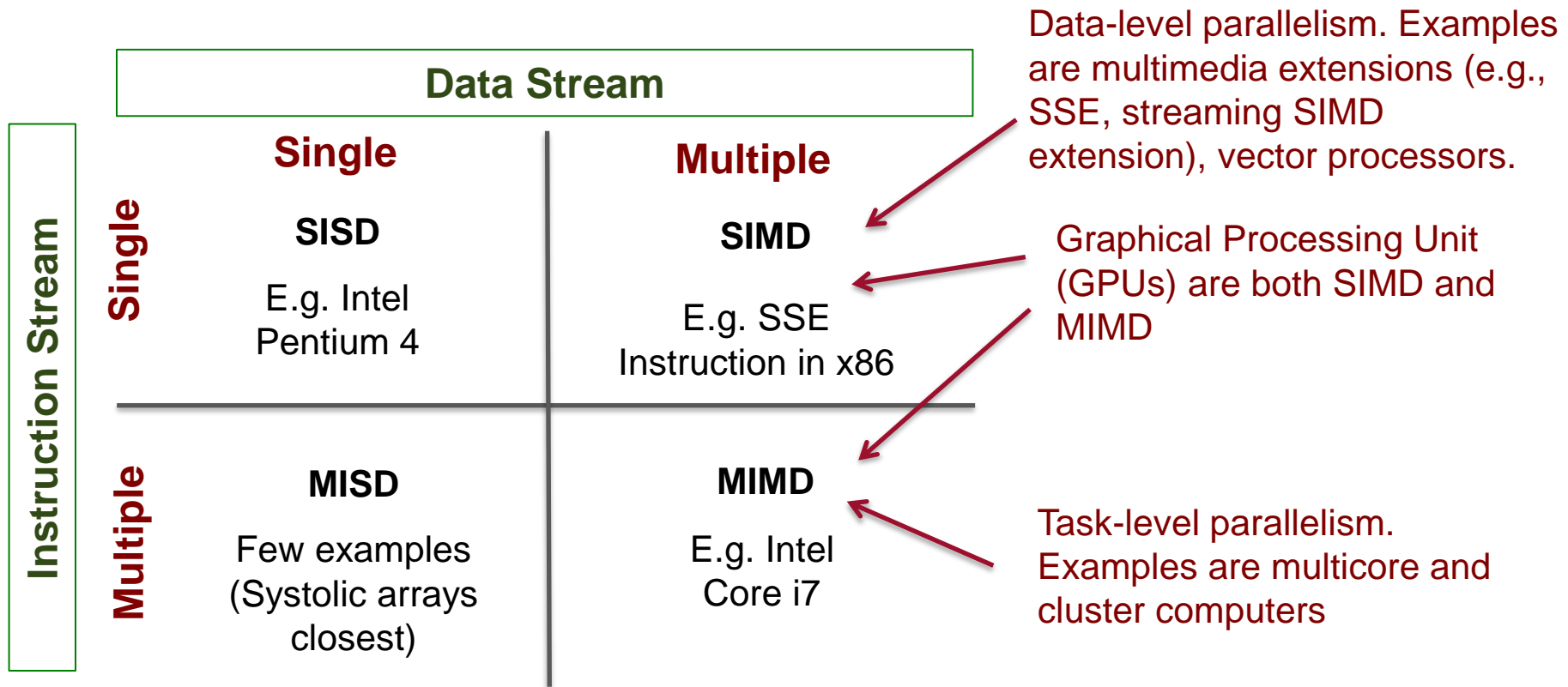


Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

SISD, SIMD, and MIMD (Revisited)



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

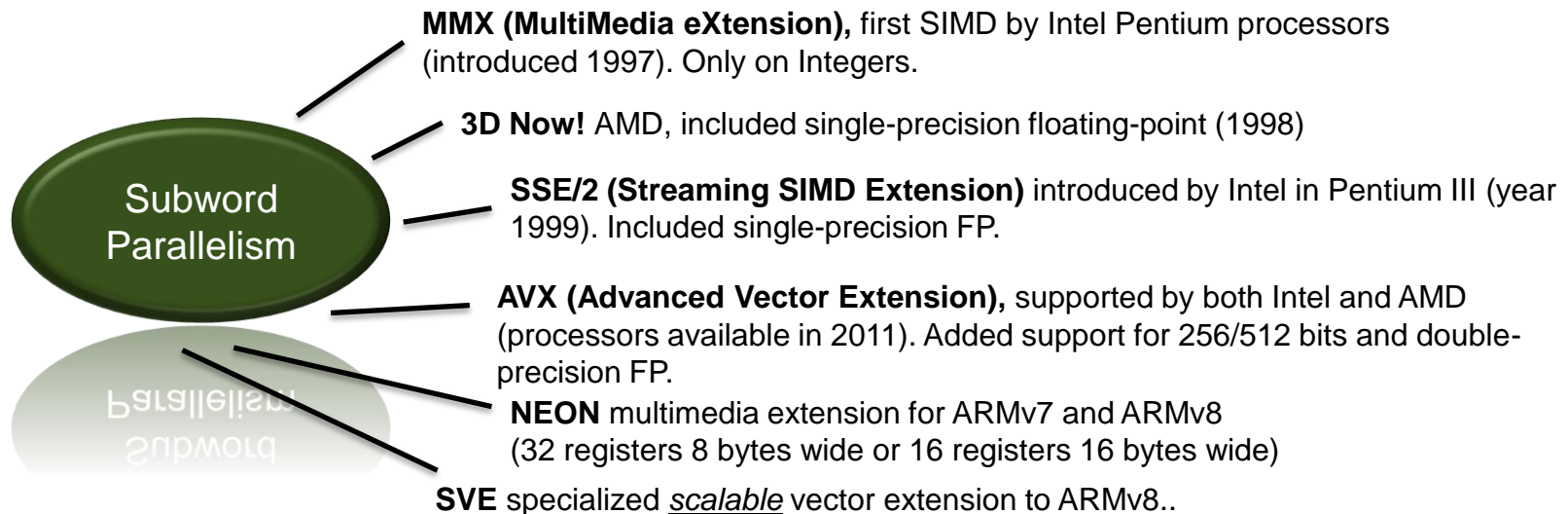
Part III
Parallelization
in Practice

Subword Parallelism and Multimedia Extensions

Subword parallelism is when a wide data word is operated on in parallel.

This is the same as SIMD or data-level parallelism.

One instruction operates on multiple data items.



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Streaming SIMD Extension (SSE) and Advanced Vector Extension (AVX)

In SSE (and the later version SSE2), assembly instructions are using two-operand format.

```
addpd %xmm0, %xmm4
```

meaning: $\%xmm4 = \%xmm4 + \%xmm0$
Note the reversed order.

Registers (e.g. $\%xmm4$) are 128-bits in SSE/SEE2.

Added the “v” for vector to distinguish AVX from SSE and renamed registers to $\%ymm$ that are now 256-bit

“pd” means Packed Double precision FP. It can operate on as many FP that fits in the register

```
vaddpd %ymm0, %ymm1, %ymm4  
vmovapd %ymm4, (%r11)
```

Question: How many FP additions does **vaddpd** perform in parallel?

Answer: 4



Moves the result to the memory address stored in $\%r11$ (a 64-bit register). Stores the four 64-bit FP in consecutive order in memory.

AVX introduced three-operand format
Meaning: $\%ymm4 = \%ymm0 + \%ymm1$



Part I

SIMD, Multithreading,
and GPUs

Part II

MIMD, Multicore,
and Clusters

Part III

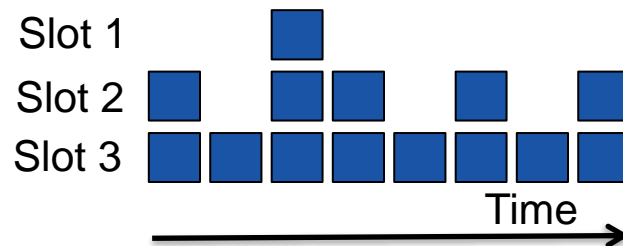
Parallelization
in Practice

Recall the idea of a multi-issue uniprocessor

Thread A

Thread B

Thread C



Typically, all functional units cannot be fully utilized in a single-threaded program (white space is unused slot/functional unit).



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Hardware Multithreading

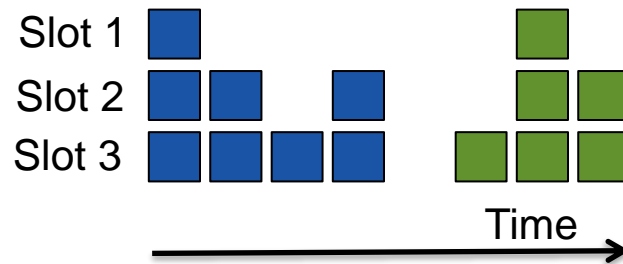
In a **multithreaded processor**, several hardware threads share the same functional units.

The purpose of multithreading is to hide latencies and avoid stalls due to cache misses etc.

Thread A

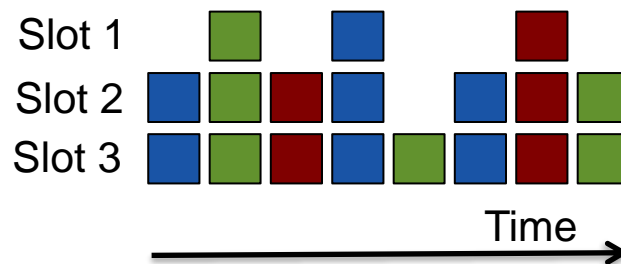
Thread B

Thread C



Coarse-grained multithreading, switches threads only at costly stalls, e.g., last-level cache misses.

Cannot overcome throughput losses in short stalls.



Fine-grained multithreading switches between hardware threads every cycle. Better utilization.



Part I

SIMD, Multithreading, and GPUs

Part II

MIMD, Multicore, and Clusters

Part III

Parallelization in Practice

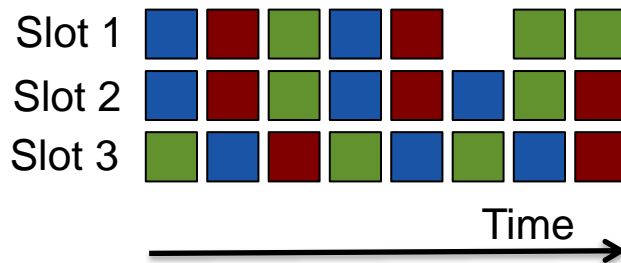
Simultaneous multithreading (SMT)

Simultaneous multithreading (SMT) combines multithreading with a multiple-issue, dynamically scheduled pipeline.

Thread A

Thread B

Thread C



Can fill in the holes that multiple-issue cannot utilize with cycles from other hardware threads. Thus, better utilization.

Example: **Hyper-threading** is Intel's name and implementation of SMT. That is why a processor can have 2 real cores, but the OS shows 4 cores (4 hardware threads).



Part I

SIMD, Multithreading,
and GPUs

Part II

MIMD, Multicore,
and Clusters

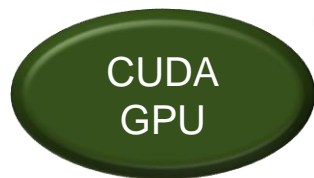
Part III

Parallelization
in Practice

Graphical Processing Units (GPUs)

A **Graphical Processing Unit (GPU)** utilizes multithreading, MIMD, SIMD, and ILP. The main form of parallelism that can be used is data-level parallelism.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model from NVIDIA.



The parallelism is expressed as CUDA threads. Therefore, the model is also called **Single Instruction Multiple Threads (SIMT)**.

A GPU consists of a set of **multithreaded SIMD processors** (called streaming multiprocessor using NVIDIA terms). For instance 16 processors.

The main idea is to execute a massive number of threads and to use **multithreading** to hide latency. However, the latest GPUs also include caches.



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice


Part II

MIMD, Multicore, and Clusters



Acknowledgement: The structure and several of the good examples are derived from the book “Computer Organization and Design” (2014) by David A. Patterson and John L. Hennessy

Part I
SIMD, Multithreading,
and GPUs

 **Part II**
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Shared Memory Multiprocessor (SMP)

A **Shared Memory Multiprocessor (SMP)** has a *single physical address space* across all processors.

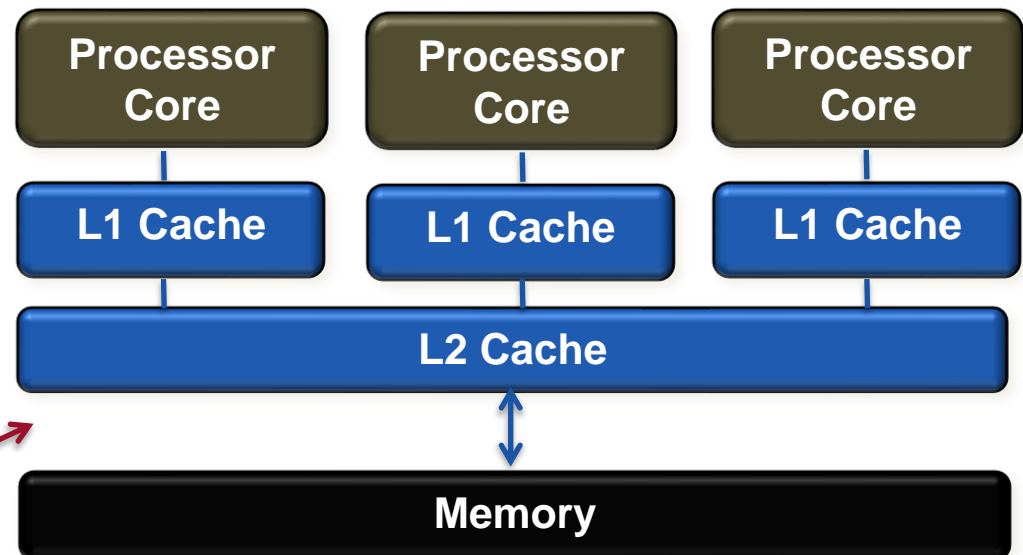
An SMP is almost always the same as a **multicore processor**.

In a **uniform memory access (UMA)** multiprocessor, the latency of accessing memory does not depend on the processor.

In a **nonuniform memory access (NUMA)** multiprocessor, memory can be divided between processor and result in different latencies.

Processors (cores) in a SMP communicate via **shared memory**.

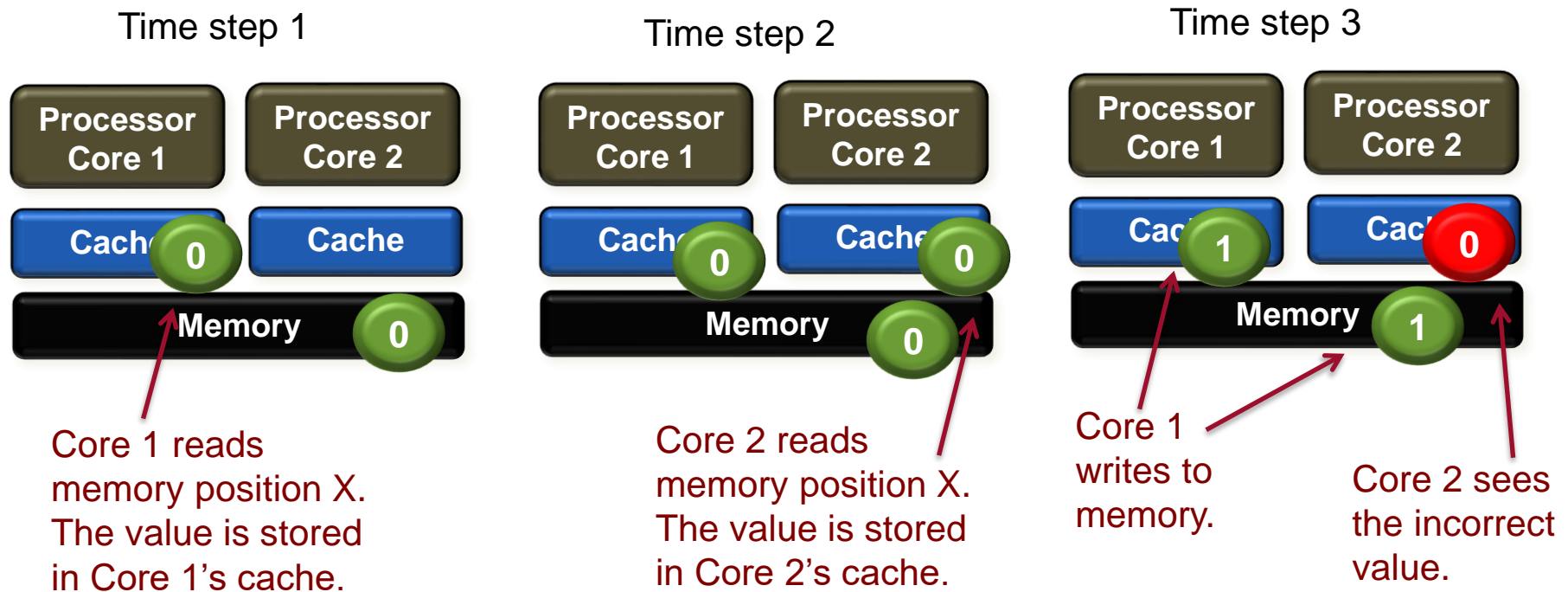
Alternative: Network on Chip (NoC)



Cache Coherence

Different cores' local caches could result in that different cores see different values for the same memory address.

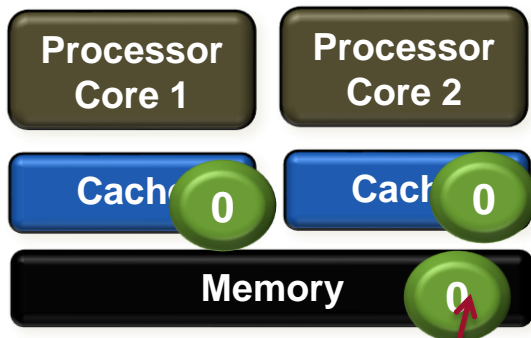
This is called the **cache coherency** problem.



Snooping Protocol

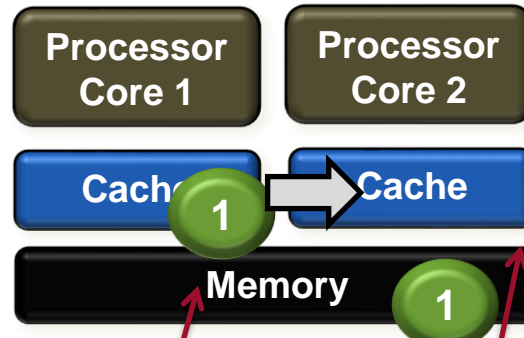
Cache coherence can be enforced using a cache coherence protocol. For instance a *write invalidate protocol*, such as the **snooping protocol**.

Time step 2



Core 2 reads memory position X. The value is stored in Core 2's cache.

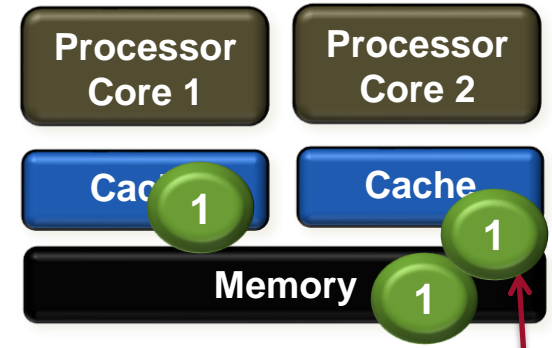
Time step 2



Core 1 writes to memory.

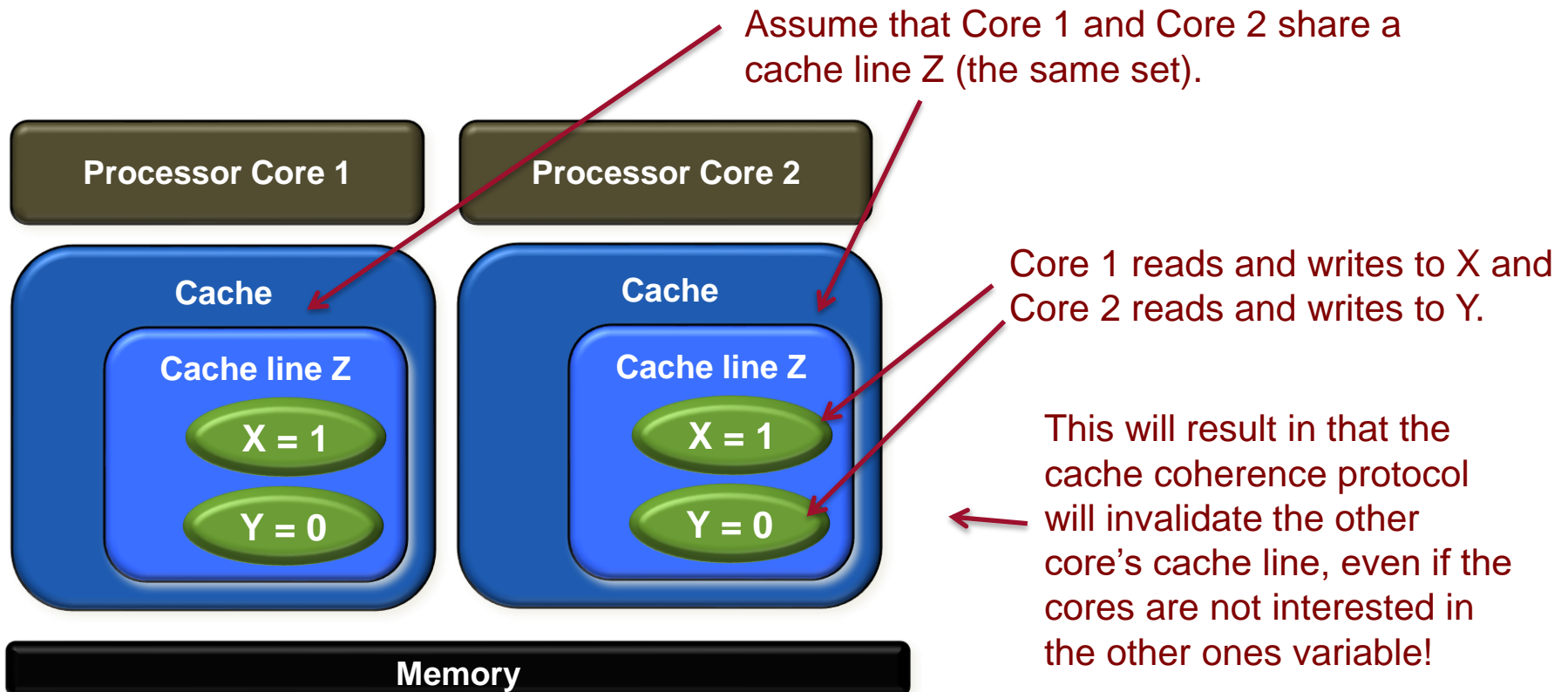
The write invalidates the cache line of other processors.

Time step 3



Core 2 now tries to read the variable, it gets a cache miss and loads the new value from memory (heavily simplified example)

False Sharing

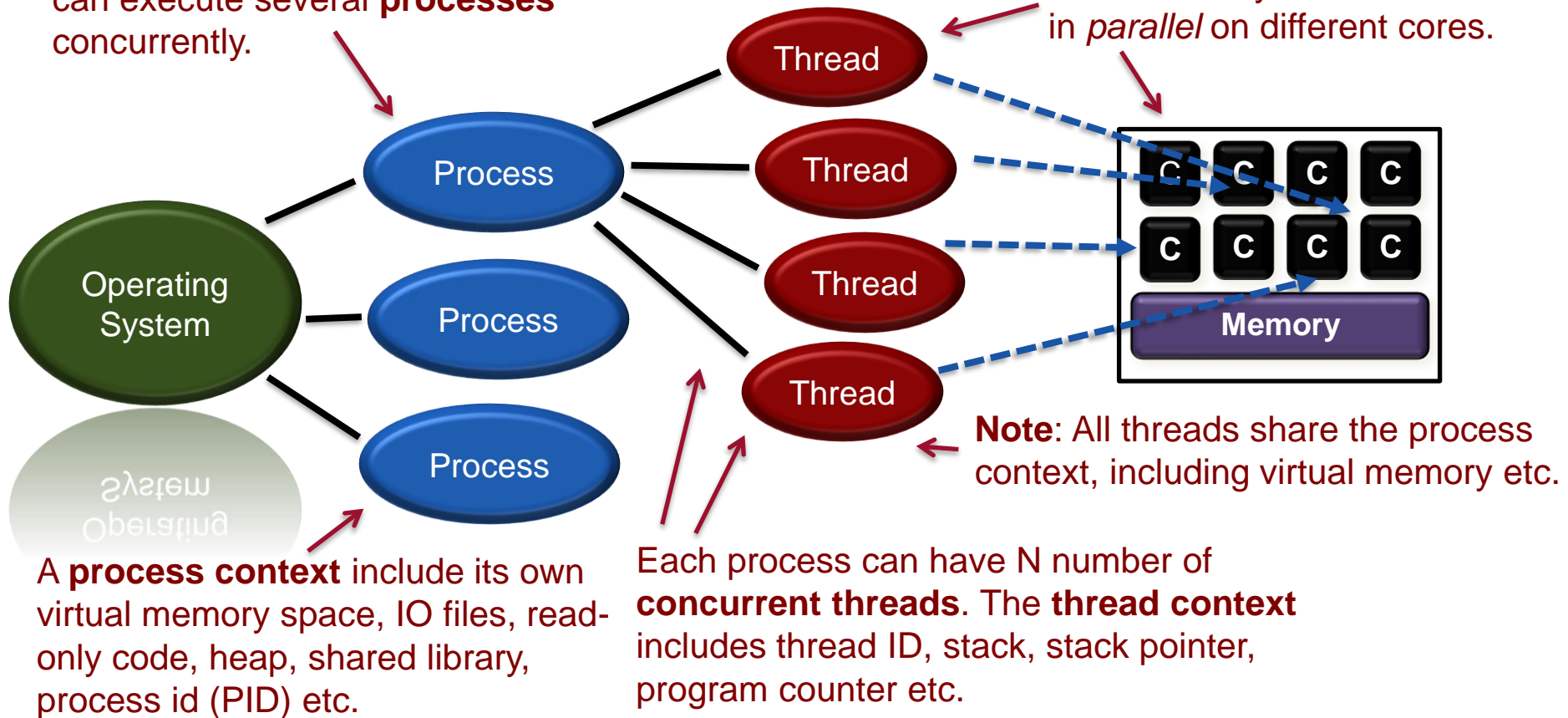


This is called **false sharing**.

Processes, Threads, and Cores

A modern **operating system (OS)** can execute several **processes** concurrently.

Concurrent threads are **scheduled** by the OS to execute in *parallel* on different cores.



```
int main() {
    pthread_t tid1, tid2;
    int max;
    max = 40000;
    pthread_create(&tid1, NULL, count, &max);

    max = 60000;
    pthread_create(&tid2, NULL, count, &max);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    pthread_exit(NULL);
}
```

Exercise: What is the output?

Answer: Possibly different values each time...



Semaphores

A **semaphore** is a global variable that can hold a nonnegative integer value. It can only be changed by the following two operations.



P(s): If $s > 0$, then decrement s and return.
If $s = 0$, then wait until $s > 0$, then decrement s and return.



V(s): Increment s .

Note that the check and return of $P(s)$ and increment of $V(s)$ must be **atomic**, meaning that appears to be “instantaneously”.

Semaphores were invented by Edsger Dijkstra, who was originally from the Nederland. P and V is supposed to stand for **Prolaag** (probeer te verlagen, “try to reduce”) and **Verhogen** (increase).

Mutex

A semaphore can be used for mutual exclusion, meaning that only one thread can access a particular resource at the same time. Such a **binary semaphore** is called a **mutex**.

A global binary semaphore is initiated to 1.

```
semaphore s = 1
```

One of more threads execute:

```
P(s);
```

```
Code to  
protected...
```

```
V(s);
```

One or more threads are executing code that needs to be protected.

P(s), also called wait(s), checks if the semaphore is nonzero. If so, **lock the mutex**, else wait.

In the critical section, it is ensured that not more than one thread can execute the code at the same time.

V(s), also called post, **unlocks the mutex** and increments the semaphore.

Programming with Threads and Shared Variables with Semaphores

```
volatile int counter = 0;
sem_t *mutex;

void *count(void *data){
    int i;
    int max = *((int*)data);
    for(i=0; i<max; i++){
        sem_wait(mutex); /* P() */
        counter++;
        sem_post(mutex); /* V(m) */
    }
    pthread_exit(NULL);
}
```



Exercise: Is it correct this time?

```
int main(){
    pthread_t tid1, tid2;
    int max;

    mutex = sem_open("/semaphore", O_CREAT,
                      O_RDWR, 1);
    sem_unlink("/semaphore");
    max = 40000;
    pthread_create(&tid1, NULL, count, &max);
    max = 60000;
    pthread_create(&tid2, NULL, count, &max);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    sem_close(mutex);
    pthread_exit(NULL);
}
```

Problem. We update the value max, that is also shared...

Programming with Threads and Shared Variables with Semaphores

Correct solution...

```
volatile int counter = 0;
sem_t *mutex;

void *count(void *data){
    int i;
    int max = *((int*)data);
    for(i=0; i<max; i++){
        sem_wait(mutex); /*P()*/
        counter++;
        sem_post(mutex); /*V(m)*/
    }
    pthread_exit(NULL);
}
```

Simple solution. Use different variables.

```
int main(){
    pthread_t tid1, tid2;
    int max1 = 40000;
    int max2 = 60000;

    mutex = sem_open("/semaphore", O_CREAT,
                    0777, 1);
    sem_unlink("/semaphore");
    pthread_create(&tid1, NULL, count, &max1);
    pthread_create(&tid2, NULL, count, &max2);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    sem_close(mutex);
    pthread_exit(NULL);
}
```

Clusters and Warehouse Scale Computers

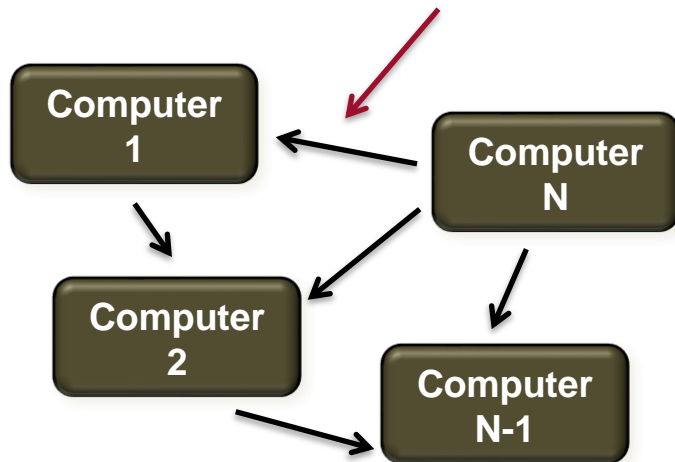


Photo by Robert Harker

A **cluster** is a set of computers that are connected over a local area network (LAN). May be viewed as one large multiprocessor.

Warehouse-Scale Computers are very large cluster that can include 100 000 servers that act as one giant computer (e.g., Facebook, Google, Apple).

Clusters do not communicate over shared memory (as for SMP) but are using **message passing**.



MapReduce is a programming model that is popular for batch processing.

1. **Map** applies a programmer defined function on all data items.
2. **Reduce** collects the output and collapse the data using another programmer defined function.

The map step is highly parallel. The reduce stage may be parallelized to some extent.



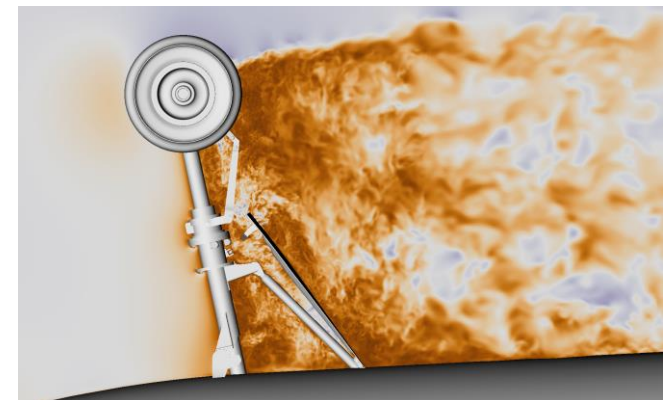
Supercomputers

Similar to a cluster but with focus on **high-performance**:

- Used to solve tough, real-life problems
 - Medicine, Weather, Fluid-Dynamics, AI, ...
- Fast inter-node communication
 - Infiniband, Tofu, Slingshot, etc.
- Large amount of memory bandwidth (HBM2, etc.)
- Non-volatile in-memory storage (Burst-buffers, think caches but for I/O)
- Performance measured in FLOP/s (double-precision)
- Programmed using different models
 - e.g., **OpenMP** for intra-node (shared memory)
 - Message Passing Interface (**MPI**) inter-node

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.26GHz, Tofu Interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SCoRidge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,094
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Tianhe-2A - TH-1V8-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Mainframe-2000, NJUT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

The Top500 list (www.top500.org)



Flow around landing gear of a private jet
(Photo: Niclas Jansson)



The KTH Beskow Supercomputer at PDC.

Part I
SIMD, Multithreading,
and GPUs

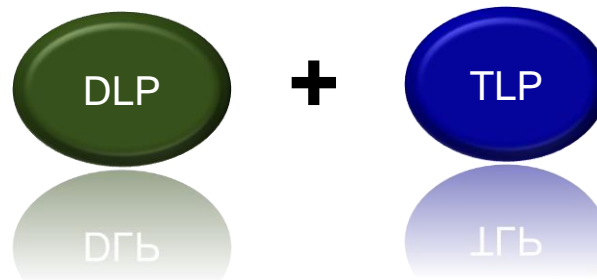


Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Part III


Parallelization in Practice



Acknowledgement: The structure and several of the good examples are derived from the book “Computer Organization and Design” (2014) by David A. Patterson and John L. Hennessy

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

 **Part III**
Parallelization
in Practice

General Matrix Multiplication (GEMM)

Simple matrix multiplication

Uses matrix size n as a parameter and single dimension for performance.

```
void dgemm(int n, double* A, double* B, double* C){  
    for(int i = 0; i < n; ++i)  
        for(int j = 0; j < n; ++j){  
            double cij = C[i+j*n];  
            for(int k = 0; k < n; k++)  
                cij += A[i+k*n] * B[k+j*n];  
            C[i+j*n] = cij;  
        }  
}
```

Parallelizing GEMM

Unoptimized

Unoptimized C version (previous page). Using one core.

1.7 GigaFLOPS (32x32)
0.8 GigaFLOPS (960x960)

SIMD

Use AVX instructions `vaddpd` and `vmulpd` to do 4 double precision floating-point operations in parallel.

6.4 GigaFLOPS (32x32)
2.5 GigaFLOPS (960x960)

ILP

AVX + unroll parts of the loop, so that the multiple-issue, out-of-order processor have more instructions to schedule.

14.6 GigaFLOPS (32x32)
5.1 GigaFLOPS (960x960)

Cache

AVX + unroll + blocking (dividing the problem into submatrices). This avoids cache misses.

13.6 GigaFLOPS (32x32)
12.0 GigaFLOPS (960x960)

Multi-core

AVX + unroll + blocking + multi core (multithreading using OpenMP)


23 GigaFLOPS (960x960, 2 cores)
44 GigaFLOPS (960x960, 4 cores)
174 GigaFLOPS (960x960, 16 cores)

Experiment by P&H on a 2.6GHz Intel Core i7 with Turbo mode turned off.

For details see P&H, 5th edition, sections 3.8, 4.12, 5.14, and 6.12

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

 **Part III**
Parallelization
in Practice

Future perspective (Part 1 of 3): MIMD, SIMD, ILP, and Caches

“For x86 computers, we expect to see two additional cores per chip every two years and the SIMD width to double every four years.”

Hennessy & Patterson, Computer Architecture – A Quantitative Approach, 5th edition, 2013 (page 263)



We must understand and utilize **both MIMD** and **SIMD** to gain maximal speedups in the future, although MIMD (multicore) has gained much more attention lately.

The previous example showed that **the way** we program for **ILP** and **caches**, also matters significantly.

Future perspective (Part 2 of 3): Heterogeneity and Accelerators

Heterogeneity

Different architectures good for different things:

❑ General-purpose systems (CPUs)

- ❑ Latency-critical applications
- ❑ Large/deep memory hierarchies
- ❑ Out-of-order execution
- ❑ Newer systems (e.g., ARM A64FX) offer high performance

❑ Accelerators

❑ Graphics Processing Units (GPUs)

- ❑ Throughput (not latency) oriented
- ❑ High amount of thread-level parallelism
- ❑ Programmed in special languages (CUDA/OpenCL/HIP/...)

❑ Field-Programmable Gate Arrays (FPGAs)

- ❑ Reconfigurable architectures (can take on many forms)
- ❑ Spatial computing (data-flow)
- ❑ Programmed using High-Level Synthesis (HLS) or HDLs
- ❑ Gaining in importance
 - ❑ Intel acquired Altera (June, 2015)
 - ❑ AMD to acquire Xilinx
 - ❑ Nvidia to acquire ARM (?)

} *Exciting (Heterogeneous) future ahead of us*

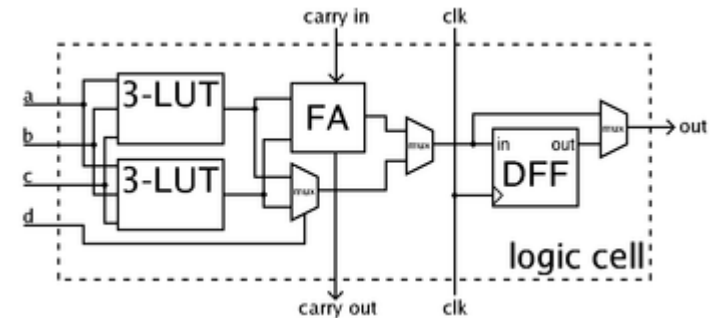


Fig. source: Wikipedia

Part I

SIMD, Multithreading,
and GPUs

Part II

MIMD, Multicore,
and Clusters

Part III

Parallelization
in Practice

Future perspective (Part 3 of 3): Specialization

Architectural Specialization

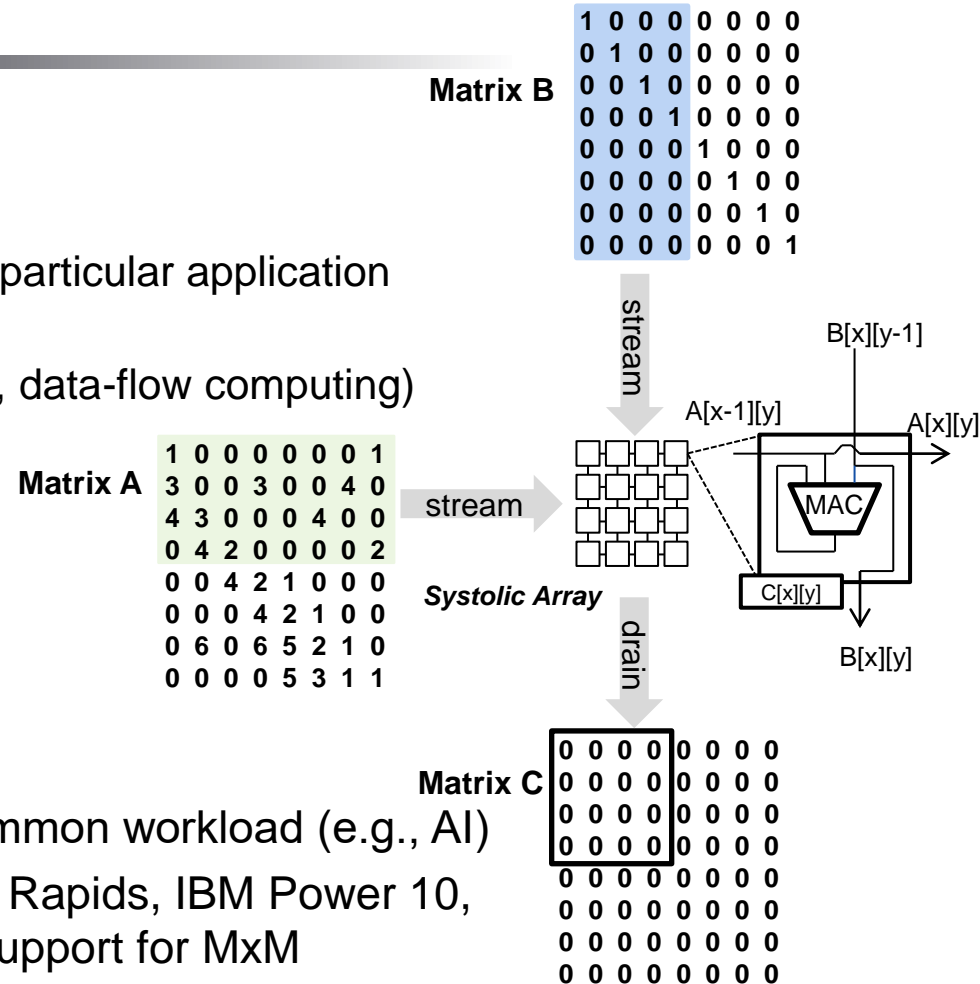
- ❑ Move away from "one size fits all"
- ❑ Specialize architecture (or silicon) toward a particular application domain
- ❑ Exploit alternative compute paradigms (e.g., data-flow computing)

Example 1: Cerebras CS-1

- ❑ "Wafer"-scale chip
- ❑ 46,225 mm² (56x larger than a GPU)
- ❑ 400,000 cores (tailored for AI training)

Example 2: Matrix Engines $C = A \times B$

- ❑ Matrix multiplication "claimed" to be a common workload (e.g., AI)
- ❑ Many modern processors (Intel Sapphire Rapids, IBM Power 10, Nvidia Volta/Ampere) include hardware support for MxM (implemented as systolic arrays)



It's about time to...

...relax



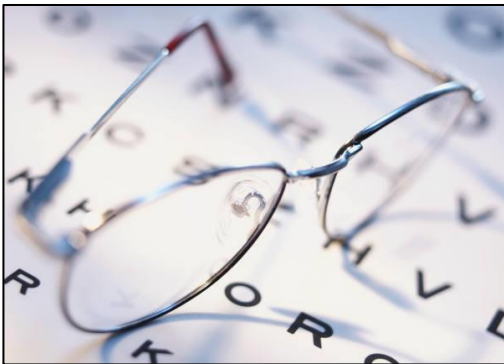
Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Reading Guidelines

Module 6: Parallel Processors and Programs



Lecture 12: Parallelism, Concurrency, Speedup, and ILP

- H&H Chapter 1.8, 3.6, 7.8.3-7.8.5
- P&H5 Chapters 1.7-1.8, 1.10, 4.10, 6.1-6.2
or P&H4 Chapters 1.5-1.6, 1.8, 4.10, 7.1-7.2

Lecture 13: SIMD, MIMD, and Parallel Programming

- H&H Chapter 7.8.6-7.8.9
- P&H5 Chapters 2.11, 3.6, 5.10, 6.3-6.7
or P&H4 Chapters 2.11, 3.6, 5.8, 7.3-7.7

Reading Guidelines

See the course webpage
for more information.

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Summary

Some key take away points:

- **SIMD and GPUs** can efficiently parallelize problems that have data-level parallelism
- **MIMD, Multicores, and Clusters** can be used to parallelize problems that have task-level parallelism.
- In the future, we should try to **combine** and use both **SIMD** and **MIMD**!



Thanks for listening!

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice