Contents

1. Introduction

High-performance computing is the practice of **aggregating computing power** to get very high performance to solve complex problems.

Historically, HPC was only related to **supercomputers** and was used only in niche contexts (scientific calculations, etc.).

Embedded computing system (ES): An electronic device that is not a computer per se, but includes a programmable computer.

Embedded systems typically use a **battery**. HPC helps obtain a good balance between performance and **energy efficiency**.

Embedded systems evolved a lot over time (more transistors, smaller size, etc.).

Bell's law: Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry.

Definition 1: Bell's law summarizes embedded systems' evolution over time

With an hardware that is increasingly **parallel** and **heterogeneous** the challenge is on the application developer, who must learn how to extract the maximum performance from these systems.

2. The multicore revolution

Architecture of modern high-performance computers is highly **heterogeneous**: there is no longer a single ISA[1], but there are different sets of processors, each with its own ISA.

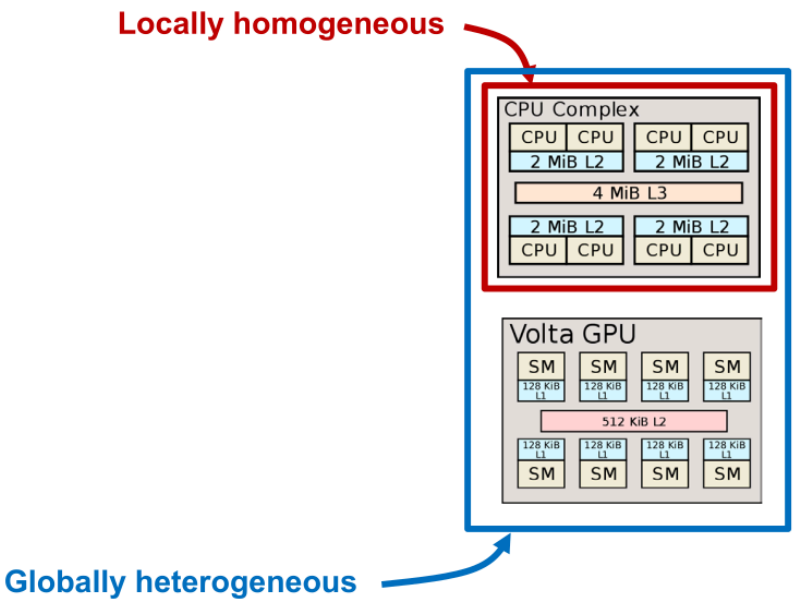


Figure 1: Modern architectures are locally omogeneus, but globally heterogeneous

Moore's law: The number of transistors in an integrated circuit doubles every two years.

[1] Instruction Set Architecture

Until 2004-2005, manufacturers have been increasing CPUs performance by simply increasing their **clock rate**. This changed after the **multicore revolution**: performance improvement is not about increasing the clock rate anymore, but it's about adding more **parallelism**.

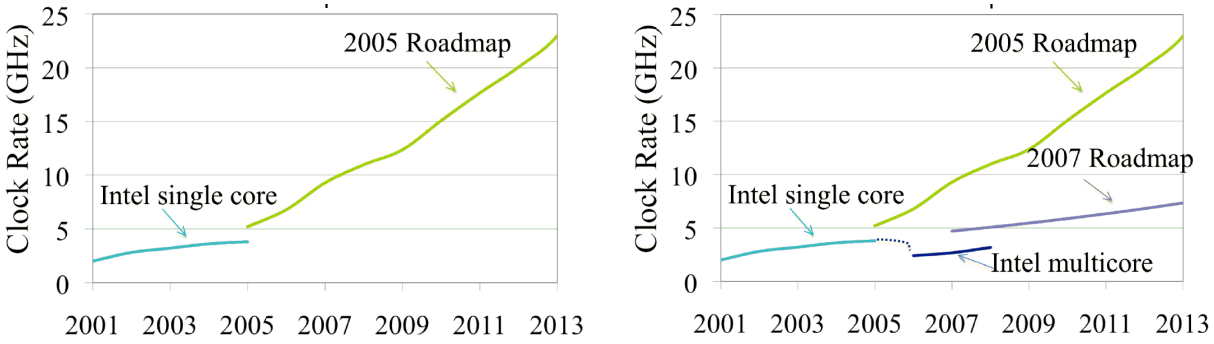


Figure 2: Intel CPU clock rate roadmaps before and after the multicore revolution.

The reason behind this change in CPU design is the **power wall**. Clock rate cannot increase without limits because at some point the amount of **heat** to dissipate and the **energy consumption** would be too high to handle.

Circuit delay: The time the current takes to traverse a circuit.

CPU frequency is circuit delay's inverse:

$$f = \frac{1}{\text{CD}}$$

Every technology generation (about 1.5 years), circuit delay is reduced by a 0.7x factor, which implies an increase of the frequency by a 1.4x factor.

An increase of CPU frequency implies an increase of **dynamic**[2] power consumed by a digital circuit:

$$P_{\text{DYN}} = \text{capacity load} \times \text{voltage}^2 \times \text{frequency}$$

Manufacturers have managed to keep dynamic power low by adding **power managers** to computer architecture. These devices reduce frequency when the CPU is idle.

But circuits also have a **static** power component, which does not depend on the frequency:

$$P_S = k_d \times \text{voltage} \times \text{current}_{\text{leaked}}$$

The only way to low static power consumption is by turning off the circuit.

Power density is another big problem: having a lot of power in a **small area** generates heat.

2. 1. Instruction-level parallelism

CPU pipeline allows to exploit parallelization on **single core** systems.

At the end of each stage of the pipeline there is a **register**. This register helps reduce the length of the **critical path**: its length is limited to the single pipeline stage and it's not the entire pipeline.

Multiple issue is another way to improve ILP: by replicating some pipeline stages (or adding other pipelines altogether) CPU is able to start multiple instructions in the same clock cycle.

[2]The power consumed while transistors are working.

Multiple issue can be either **static** or **dynamic**:

- static multiple issue is performed by the **compiler**, which schedules instructions appropriately;
- dynamic multiple issue (or **out-of-order execution**) is performed by the hardware itself

But also ILP cannot be increased without limits:

- **dependencies** reduce its applicability in practice;
- out-of-order CPUs require much more power, so this does not help in avoiding the power wall

2. 2. Memory wall

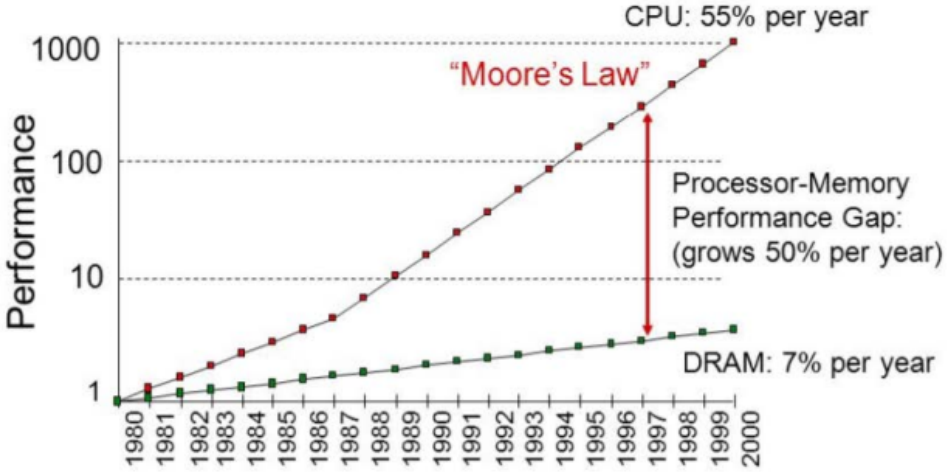


Figure 3: The gap between CPU and memory performance has increased a lot over time

It's useless to invest lots of resources in improving CPU performance, because now the bottleneck is the memory.

2. 3. Multicore

Multicore is another strategy (along with ILP) manufacturers explored to increase CPU performance.

Having multiple cores on the **same chip** has big advantages on CPU design, because each CPU can be simpler and slower (and thus less power demanding).

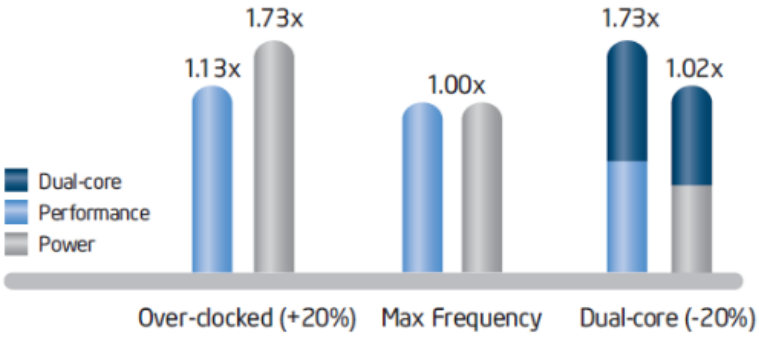


Figure 4: Multicore performance and energy demand

Performance improvement of multicore is only theoretical. The problem now is on the software, which must be written to exploit parallelism given by multiple cores.

2. 4. Dark silicon

Power consumption is still a problem also after the multicore revolution. Cores cannot be all kept on at the maximum power.

Dark silicon: A portion of hardware that cannot be used at full power, or cannot be used at all, to avoid increasing the power consumption of the circuit.

Every modern hardware design has a lot of dark silicon.

Dark silicon is the consequence of the **utilization wall**: the portion of a chip that can work at **full frequency** drops **exponentially** after each processor generation because of power constraints.

"**Four horsemen**" is a way to call 4 different proposed solutions to handle dark silicon. None of these solutions is optimal, but each one has its benefit. Probably the optimal solution is a mixture of those.

1. **shrinking** horseman: instead of having dark silicon, manufacturer should simply make smaller processors. Not ideal because dark silicon does not mean **useless silicon**, it just means that it's under-clocked or not used all of the time;
2. **dim** horseman: can be a **spacial** dimming (more cores, but each one with a lower frequency) or a **temporal dimming** (each core has a higher frequency, but they are not used all together). Temporal dimming is widely used in battery-limited sistems (e.g. smartphones);
3. **specialized** horseman: dark silicon is used only to perform **specialized** tasks (e.g. video decoding/encoding, crypto stuff, etc.);
4. **deus ex machina** horsemen: move from the current CMOS technology to another one (e.g. nanotubes). None of these newer technology is sustainable across processor generations.

3. Parallel systems

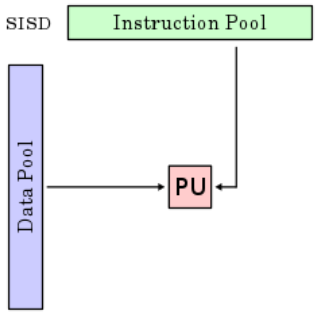
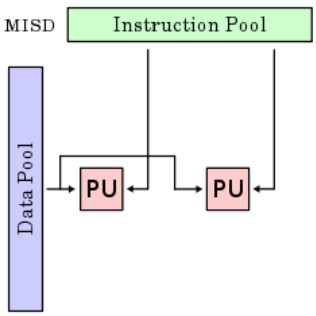
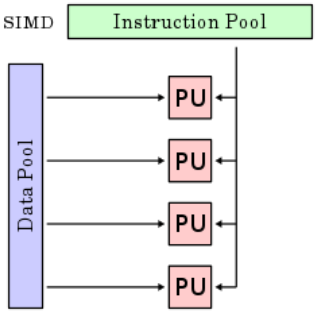
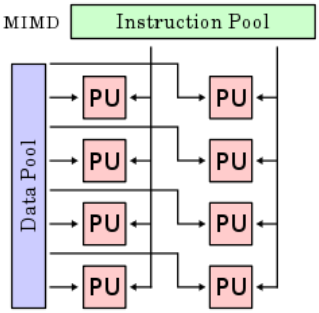
advantages	disadvantages
• effective use of all (or most of) transistors in a chip; • **scaling** (i.e. adding a core) is very easy; • each processor can be **less powerful**, which optimizes costs and power efficiency; • designing multi-processor systems is easier because once a processor has been tested it can simply be **replicated**	• performance improvement is only **theoretical**. Not all problems can be efficiently parallelized; • more processors needs to be **synchronized**; • software developers should write code that uses these multiple processors

Table 1: Pro and cons of multicore

3. 1. Taxonomy of parallel computers

		Data streams	
		Single	Multiple
Instruction Streams	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Figure 5: Flynn taxonomy of parallel computers

SISD (single instruction, single data)		A single instruction is executed on a single data stream. This is the architecture of single-core systems. These single-core systems can still use some forms of parallelization, e.g. through ILP
MISD (multiple instruction, single data)		Multiple instruction streams that can be naturally parallelized are executed on a single data stream. This architecture is quite rare today, but it's widely used in AI computation.
SIMD (single instruction, multiple data)		A single instruction is executed on multiple data streams. This architecture is widely used in GPUs.
MIMD (multiple instruction, multiple data)		Multiple instructions are executed on multiple data streams. Since different program counters are used, this architecture is able to execute **multiple programs** (or diffrent parts of the same program) at the same time.

SIMD and MIMD are the most widespread architectures today.

The most common programming style is SPMD (single program, multiple data). This is not related to the architecture, but to the **execution model** (a single program is executed on all processors of a MIMD).

3. 1. 1. SIMD architectures

In this architecture the hardware itself is able to execute the same instruction on a large dataset (e.g. scalar-vector sum or multiplication).

This architecture exploits the fact that lot of work is done inside **loops** for a large data structure (vectors, matrixes):

```
for (int i = 0; i < N; i++)
    x[i] += s
```

Listing 1: Example of scalar-vector sum

In this context there is no need to stick to the Von Neumann instruction processing (fetch, decode, execute, etc.), because when the instruction has been decoded once, it can be executed **multiple times** independently without the need to fetch/decode it again.

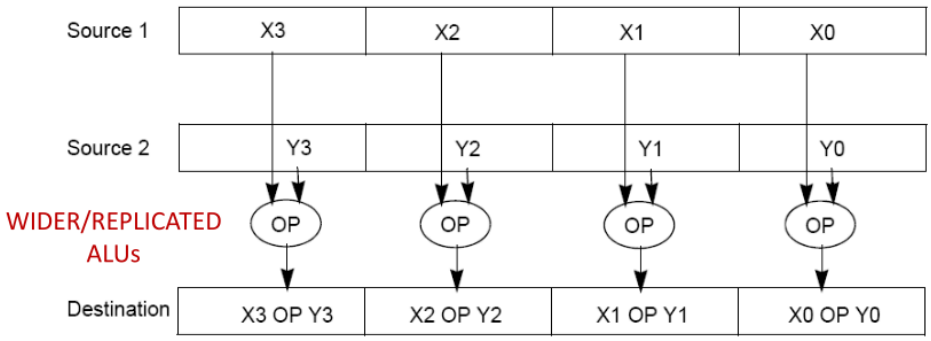


Figure 6: Schema of a SIMD architecture

To exploit data parallelism, CPU pipeline must be modified:

- EXE stage must be replicated to execute the instruction multiple times;
- a larger memory unit must be used to read/write **multiple** vector values at the same time

```
for each i in array
{
    load x[i] to a register
    add scalar coefficient s
    write the result from the register to memory
}
```

SISD

```
for each 4 members in array
{
    load 4 members of x to the SIMD register
    calculate 4 additions in one operation
    write the result from the register to memory
}
```

SIMD

Figure 7: SISD vs SIMD (pseudocode)

SIMD architecture needs **adjacent** values in memory.

Loop unrolling can help revealing more data-level parallelism than available in a single loop iteration:

```
L1:  ld   s0, 0(a0)
     ld   s1, 8(a0)
     ld   s2, 16(a0)
     ld   s3, 24(a0)
     add  s1, s1, a1
     add  s2, s1, a1
     add  s0, s0, a1
     add  s3, s1, a1
     sd   s0, 0(a0)
     sd   s1, 8(a0)
     sd   s2, 16(a0)
     sd   s3, 24(a0)
     addi a0, a0, 32
     bne  a0, t0, L1
```

Figure 8: RISC-V ASM example of an unrolled loop. Every block of 4 similar instructions can be replaced by a single SIMD instruction.

Unrolling is done automatically by the compiler.

Loops are unrolled by a factor equal to the width of the SIMD unit of the architecture.

3. 1. 2. MIMD architectures

This architecture is more complex than SIMD, because here there are multiple **independent** pipelines, each one with its own **program counter**.

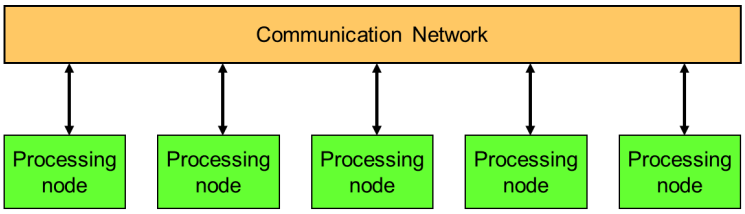


Figure 9: In MIMD each processing node is interconnected through a communication network

MIMD uses **thread-level parallelism**. Each thread can use data-level parallelism (e.g. SIMD) on its own.

MIMD architecture allows two different patterns of execution:

1. multiple program, multiple data (MPMD): each thread executes a **different** program;
2. multiple threads, multiple data (MTMD): each thread executes a different part of the **same program**

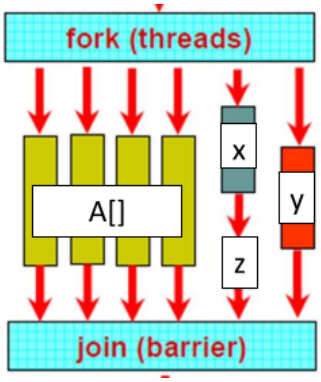


Figure 10: Different types of parallelism in action (SIMD/SPMD and MPMD)

3. 2. Memory architecture in multiprocessor systems

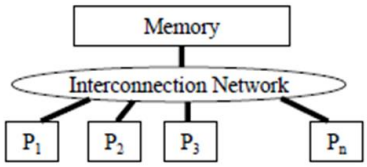
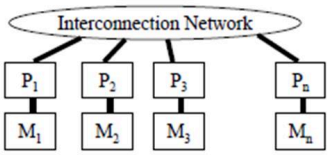
shared		• one **copy** of data shared among many cores; • synchronization must be handled explicitly; • communication between processors is implicit by modifying shared data; • scales poorly in terms of number of processors; • main issue: **correctness**
distributed		• each core has its own, private, memory, inaccessible to other cores; • each core is interconnected; • communication is explicit through a **messaging system**; • synchronization is handled implicitly during message send/receive; • better scalability; • main issue: **performance**

Table 2: Primary patterns of multicore architecture design

3. 2. 1. Shared memory

In a shared memory address model:

- each processor can address every physical location in the machine;
- each process can address all data it shares with others;
- communication happens through load and stores
- **memory hierarchy** applies, with multiple levels of cache

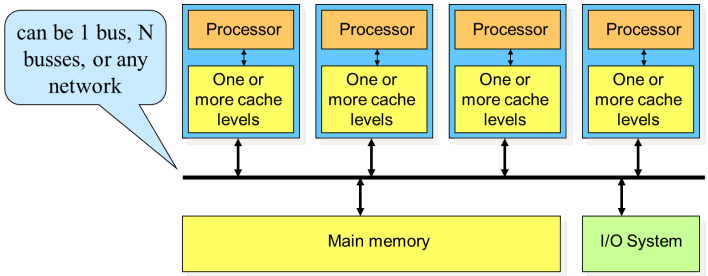
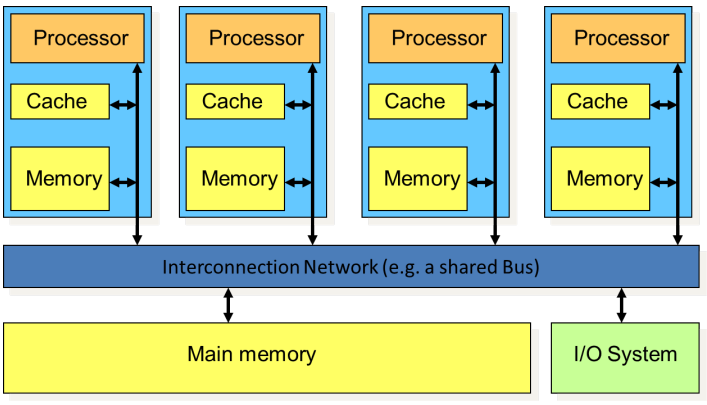
physically shared (typical of on-chip multiprocessors)	
distributed shared memory (typical of large-scale computers)	

Table 3: Different physical implementations for shared memory

Physically shared memory is more performant, because plain load/stores can be used to access data. In distributed shared memory instead a message-passing protocol must be used.

NUMA architecture: A type of multiprocessor architecture where access to memory is **non-uniform**. Different processors take different time to access the memory, because of the **physical distance** between the processor and its own memory.

3. 2. 2. Distributed memory

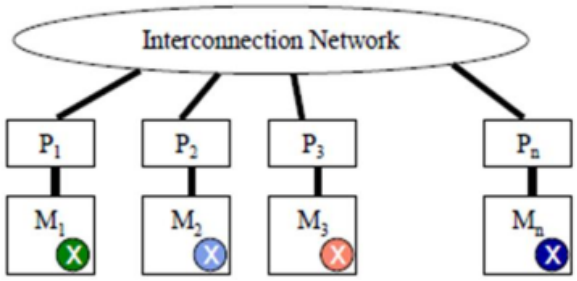


Figure 11: Each processor has its own copy of `X`

Communication happens through **message exchange**. If P1 needs to access P2's value of `X`:
1. P1 sends a read request for `X` to P2;
2. P2 sends a copy of its `X` to P1;
3. P1 stores the received value in its own memory

Data exchange requires **synchronization** between sender and receiver.

Messages are processed through a **FIFO queue**:

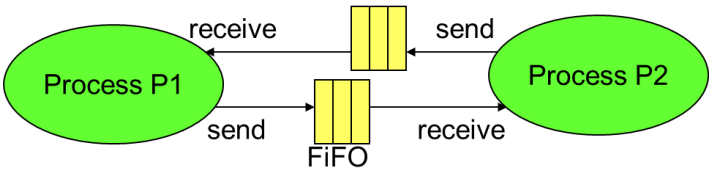


Figure 12: Message passing communication model

Message send/receive can either be blocking (synchronous) or asynchronous.

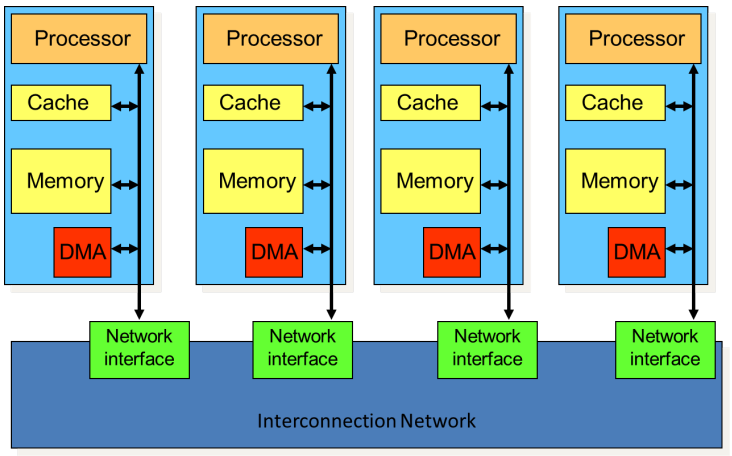


Figure 13: Physical implementation of message passing

processor 1

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

sequential

parallel with messages

processor 1

```
A[n] = {…}
B[n] = {…}

Send (A[n/2+1..n], B[1..n])

for (i = 1 to n/2)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Receive(C[n/2+1..n][1..n])
```

processor 2

```
A[n] = {…}
B[n] = {…}

Receive(A[n/2+1..n], B[1..n])

for (i = n/2+1 to n)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Send (C[n/2+1..n][1..n])
```

Figure 14: Example of a program that uses distributed memory to calculate the distance between two given points on the plane. P1 holds `A` and `B`, while P2 holds `C`.

4. Shared memory issues

Main issues of shared memory:

- **coherence**: each processor holds a different **copy** of the same data. It's crucial that all processors see the **most recent version** of the data;
- **synchronization**: multiple processors accessing the memory must not mess up the computation. Memory **locks** should be handled;
- **consistency**: the time it takes to propagate changes of the data to all memory locations (e.g. internal cache of each CPU)

4. 1. Coherence

Coherence deals with mantaining a **global order** in which writes to a **single location** are seen by all processors.

Coherence is relevant anywhere there are multiple actors who may write the memory. This may also happen in single core systems because both the CPU and the I/O system may want to access the memory.

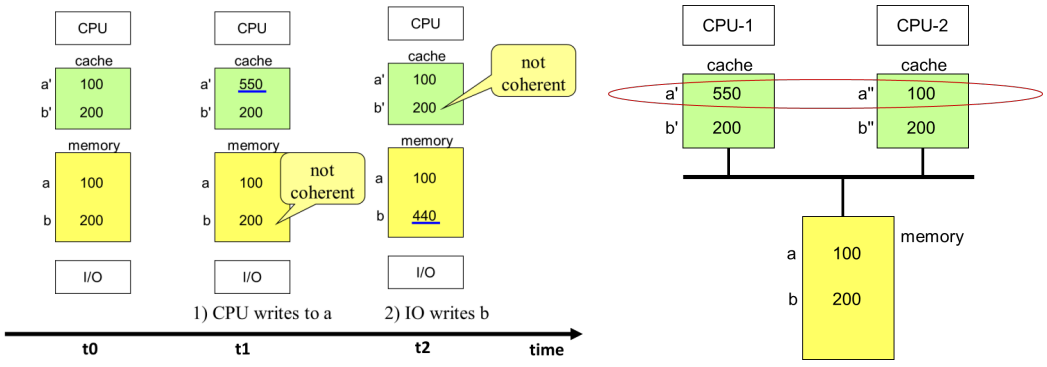


Figure 15: Coherence in single and multiprocessor systems

Since writes to DRAM are expensive, single-core systems adopt a **write-back logic**: writes to DRAM are not performed until the very last moment, which is a **cache eviction**.

In a multiprocessor system, the interconnection network intercepts a write request of a processor and broadcasts it to all other processors. Each processor can then proceed with cache invalidation to update their local copy of the data.

The coherence system must be completely **transparent** to the user and to the programmer.

Key ideas for cache coherence:
- each processor stores a **copy** of the data it needs in its own cache;
- if processors are only **reading** data, nothing happens;
- if even a single processor is **writing** a value, all copies of that data must be invalidated

4. 1. 1. Snoopy protocol

Each processor has a **snoopy cache** which listens for memory updates on the bus in order to keep the memory view of each processor coherent.

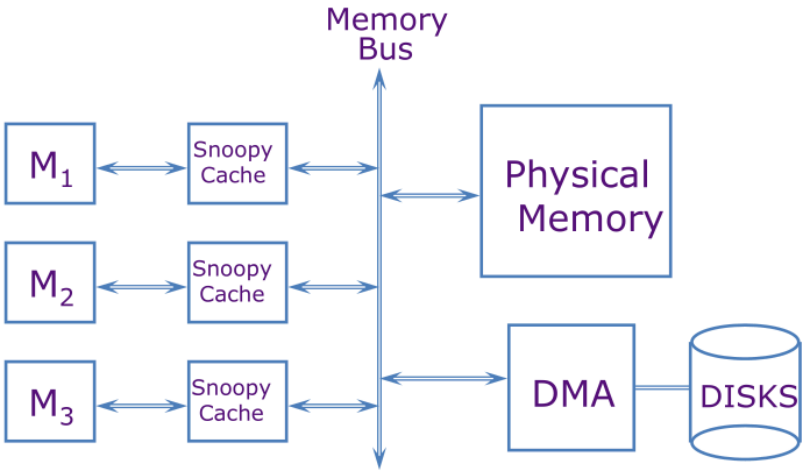


Figure 16: Snoopy caches

This protocol does not scale well because it operates over a **single bus**.

On a **write miss**, the address is invalidated in all other caches before the write is performed.

On a **read miss**, if a **dirty copy** is found in some cache, a write-back is performed before the memory is read.

4. 1. 2. MSI protocol (Modified, Shared, Invalid)

Each cache line has some **state bits**. Each one of them can be either:
- **shared**: the processor read a value from the memory because of a read miss. If other processors read the same value, the state does not change;
- **invalid**: some **other** processor has written the data;
- **modified**: **this** processor has written the data

The processor with the cache line in state M is the owner of the most recent copy of the data. This may also happen when the cache line is in state S, but the **dirty bit** is different (on when in state M, off when in state S).

Only one processor in the system can have a cache line in state M for a specific data, because there could be only one owner of the most recent copy of the data.

action	P1 cache line state	P2 cache line state
P1 reads	S	-
P1 writes	M	-
P2 reads	S (write-back)	S
P2 writes	I	M
P1 reads	S (read miss)	S (write-back)
P1 writes	M	I
P2 writes	I	M

Table 4: MSI protocol example

4. 1. 2. 1. MSI protocol performance

I state is a very cheap way to mark the data as invalid. The alternative would be to read the value **immediately** from the memory, which is much more costly.

MSI protocol treats every data item as **shared**. This is its major performance drawback, because when a processor write a value into the memory, all other processors must be notified and this generates a lot of bus traffic which may be useless if the data is actually **private** to that processor.

4. 1. 3. MESI protocol (Modified exclusive, Exclusive but unmodified, Shared, Invalid)

This protocol adds to the MSI protol the state E: **exclusive but unmodified**. This state marks a **private** value that has not been modified (yet) by the processor.

This protocol treats all data items as **private** by default. The state of a cache line keeps bouncing between states E and M until some other processor needs that data, which causes the line to jump to S state.

action	P1 cache line state	P2 cache line state
P1 reads	E	-
P1 writes	M	-
P1 reads/writes	M	-
P2 reads	S (write-back)	S
P2 writes	I	I
P1 reads	S	S

Table 5: MESI protocol example

4. 1. 3. 1. MESI protocol performance

MESI performs better than MSI because it generates much less bus traffic.

4. 1. 4. Coherence misses

Cache lines are organized in **blocks**. A single cache block can hold more than one word of data. Processors access a single word, but cache coherence works at block level.

If P_1 writes w_1, P_2 writes w_2 and both w_1 and w_2 share the same block, P_1's write will generate a **false sharing miss** on P_2. P_2 needs to evict the whole cache block even though it's working only with w_2.

time	P1	P2	outcome
1	write `x1`	read `x2`	false miss: P1′s write to `x1` invalidates also `x2` for P2
2	write `x1`	write `x2`	false miss: the cache line is invalidated for both P1 and P2, even though P2 does not care about `x1` and P1 does not care about `x2`
3	read `x2`	-	true miss: the cache line is dirty because P2 has written `x2` previously

Table 6: True vs false sharing misses (assuming `x1` and `x2` are in the same cache block)

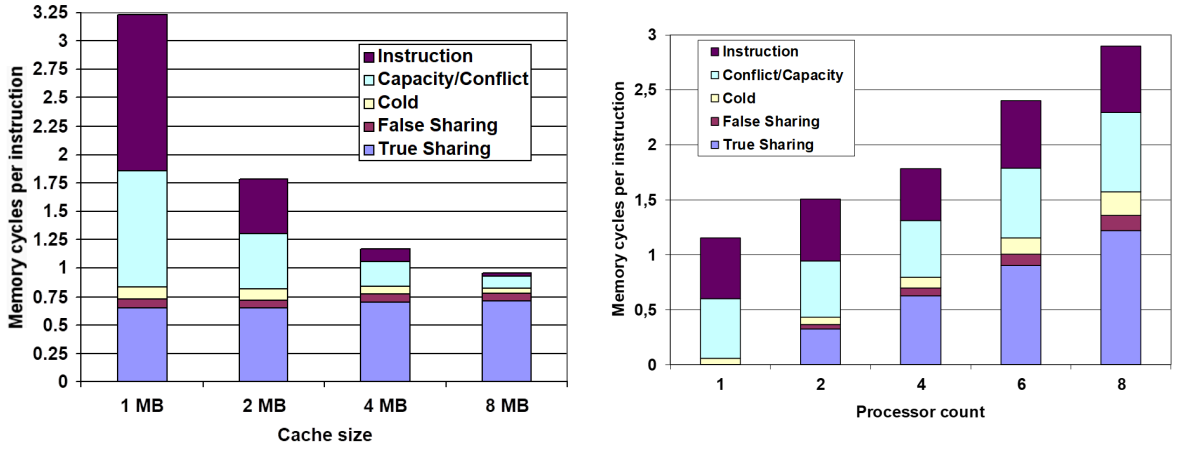


Figure 17: Cache misses over cache size and processor count increments

Bigger cache size reduces the impact of cache misses, but a bigger number of processors increases sharing misses. It's important to find a balance between these two factors.

4. 1. 5. Implementing cache coherence

A cache coherent system must:

- provide a set of **states**, with a transition diagram and a set of actions for each state;
- manage the coherence protocol, which means:
  - determine **when** to invoke the protocol (e.g. distinguishing between shared and private data);
  - find the state of the address in other processors' caches;
  - locate other copies of the data;
  - communicate with each processor

Different systems implement cache coherence differently, but all of them invoke the protocol only when an "access fault" occurs on the cache line.

4. 1. 5. 1. Snoopy protocol

Bus-based protocol that **broadcasts** a write request from a processor to **all** other processors in the system.

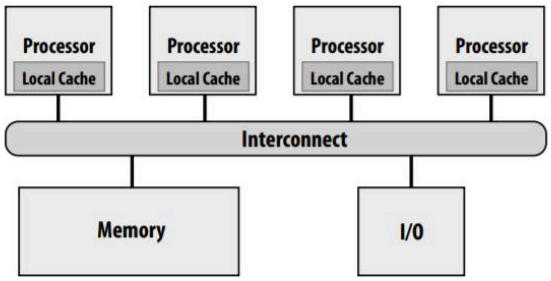


Figure 18: Architecture of a snoopy protocol

This protocol is very simple, but it does not scale well because of the high bus traffic it generates.

NUMA systems help reducing the latency (especially when there is a lot of **locality** in the application), but still they are not so useful if the cache coherence protocol cannot scale.

Scalability can be improved by applying different layes of interconnection between processors:

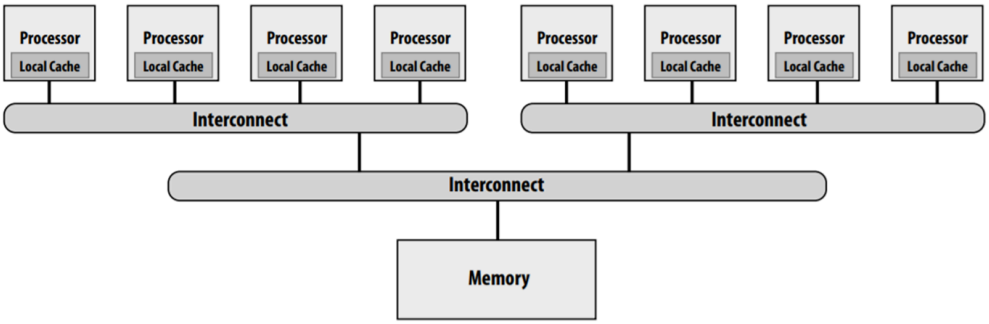


Figure 19: Hierarchy of snooping

This hierarchy is simple to build, but it has disadvantages too:

- the root becomes a bottleneck;
- latency is higher, because processors are no longer directly interconnected;

Another option to improve snoopy protocol performance is **decentralizing memory** (each group of processors has its own memory):

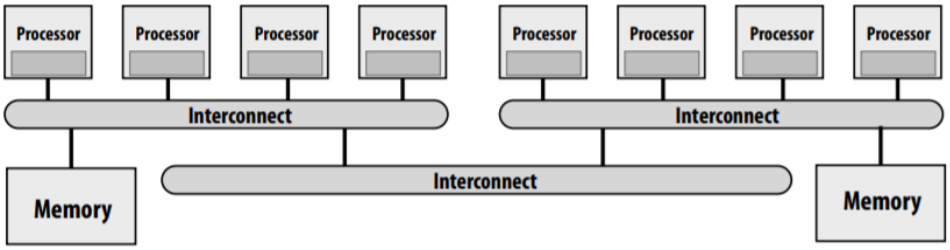


Figure 20: Decentralized memory

4. 1. 6. Directories

This approach relies on **point-to-point** messages instead of broadcasts.

Information about the cache block is stored in a **directory**:

- every memory block has associated directory information, which keeps track of copies of cached blocks (and their states);
- on a miss, the cache coherence protocol looks for the corresponding directory entry and communicates only with nodes that have copies of that data;

Directory information can be organized in different ways. A very simple directory structure is based on **presence bits**:

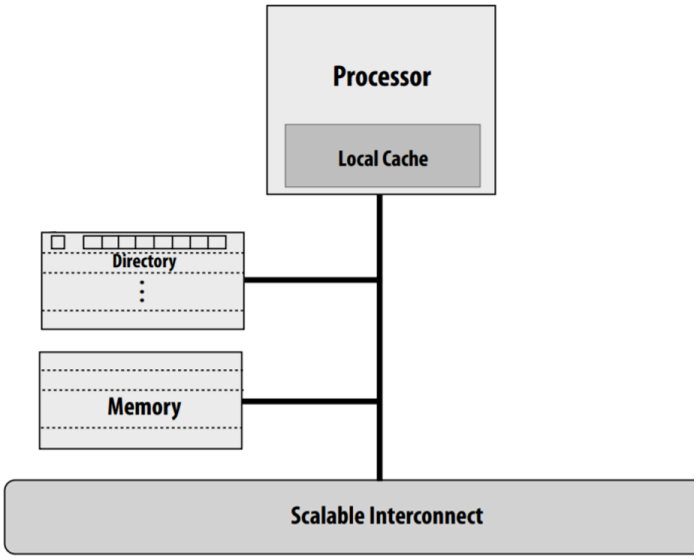


Figure 21: Directory structure with presence bits

Each directory entry stores n bits, one for each processor. The i-th bit is 1 if the i-th processor holds that memory block in its cache. The directory entry also stores the **dirty bit**, which is set to 1 if one processor has written into the memory block.

Each processor has a separate directory:

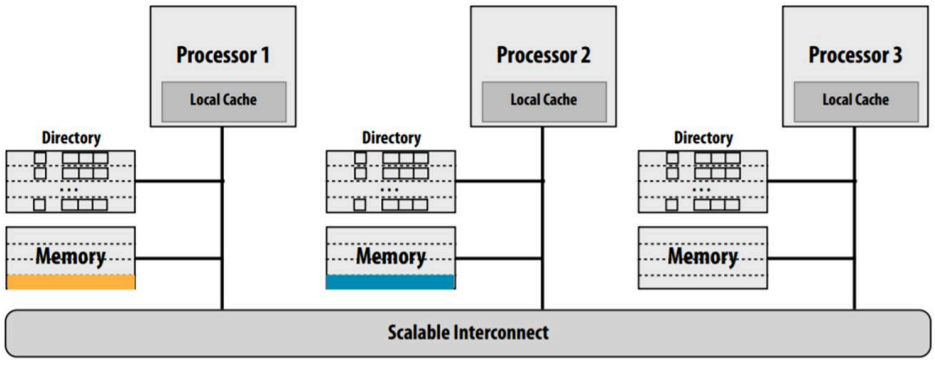


Figure 22: Directory for each node

- the **home node** of a memory block is the node whose memory is allocated for that data;
  - being the home node does not necessarily mean to be the owner of the most recent copy of the data, it just means that memory block is allocated on that node's memory;
- the **requesting node** is the node who asks the home node for a memory block

4. 1. 6. 1. Example 1: clean block read miss

1. P1 has a cache miss;
2. P1 asks P2 for that data, since P2 is the home node of that memory block;
3. P2 looks up the directory entry for that memory block, finds that the block is clean, and sends the data to P1;
4. P2 updates the directory entry by adding a presence bit for P1

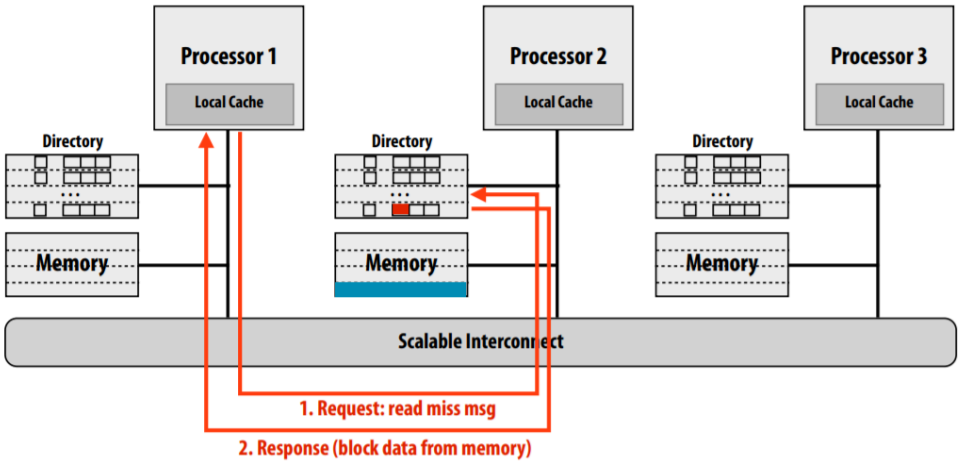


Figure 23: Clean block read miss

4. 1. 6. 2. Example 2: dirty block read miss

1. P1 has a cache miss;
2. P1 asks P2 for that data, since P2 is the home node of that memory block;
3. P2 looks up the directory entry, finds that the block is dirty, and tells P1 to get the data from P3 instead, which is the owner of the most recent copy of the data;
4. P1 asks P3 for the data;
5. P3 sends the data to P1;
6. P2 updates the directory entry by removing the dirty bit and by adding a presence bit for P1

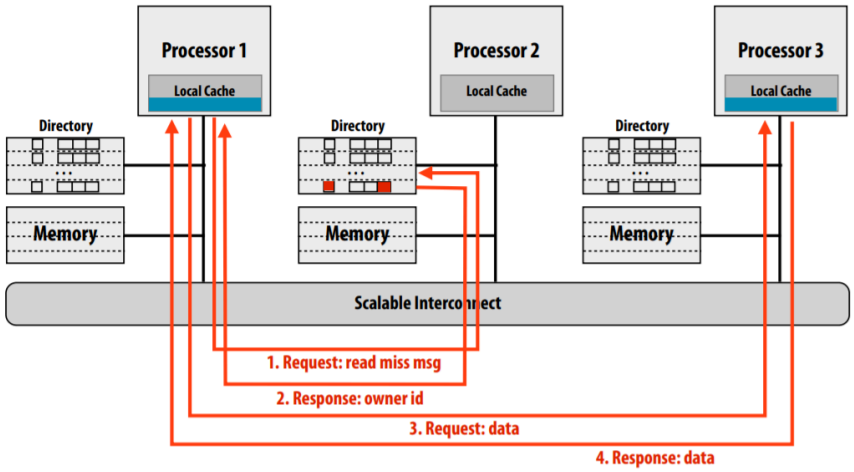


Figure 24: Dirty block read miss

4. 1. 6. 3. Example 3: write miss

1. P1 has a write miss;
2. home node of that memory block is P2, so P1 asks P2 to update the value in its memory;
3. P2 looks up the directory entry, finds that P2 itself and P3 hold a copy of the data, and tells P1 to ask those nodes to invalidate their copies;
4. P1 asks P2 and P3 to invalidate their copies and waits for an acknowledge;
5. after receiving both invalidation acks, P1 can perform the write

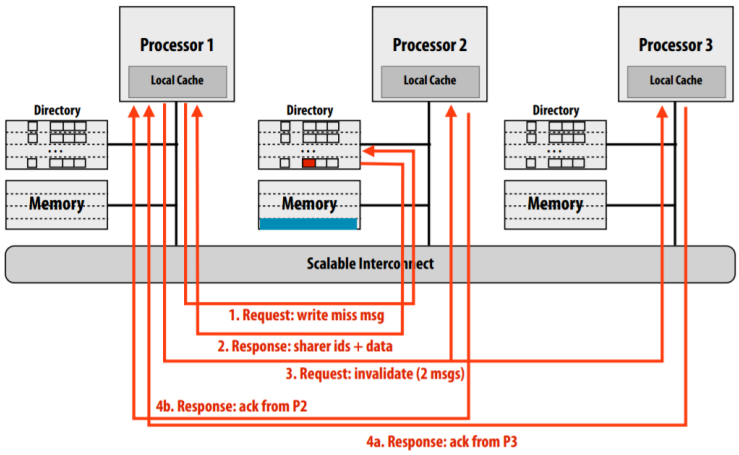


Figure 25: Write miss

4. 1. 6. 4. Summary: advantages of directories

On reads, directory tells the requesting node where to get the memory block from (either from the home node or from the owning node). In both cases it's point-to-point communication, which is much more efficient than broadcast messaging.

On writes, the advantage of directories depends on the number of sharers[3]. In the worst case this falls back to broadcast messaging.

4. 1. 7. Cache access patterns

In general there are two main cache access patterns:

- mostly-read objects: the same data is shared among a large number of processors, but writes are infrequent, so their impact on performance is negligible;
- migratory objects: the number of sharers is very low and it does not scale with the number of processors. Writes do not affect performance in this case too, because sharers count is low

A few observations can be done based on these access patterns:

- directories are very useful for limiting traffic;
- since it's quite rare that **all** nodes are working on the same data, directory structure can be optimized to reduce storage overhead

4. 1. 8. Full bit vector approach

Directory entries holds one presence bit for each processor.

This approach has a storage overhead of $P \times M$, where P is the number of processors and M is the number of memory blocks. This overhead can be diminished by lowering one of these factors:

- M can be reduced by increasing cache block size, but this will affect the performance impact of **sharing misses**;

[3] A sharer is a node which holds a copy of the data.

- P can be reduced by **grouping** processors together. The presence bit would not be for a single processor but for a group of them. Within each group a (simpler) snoopy protocol can be used.

4. 1. 8. 1. Limited pointer schemes (LPS)

Optimization to the full bit vector approach to reduce factor P.

This scheme exploit the fact that, in most cases, the number of sharers is very low (about 10):

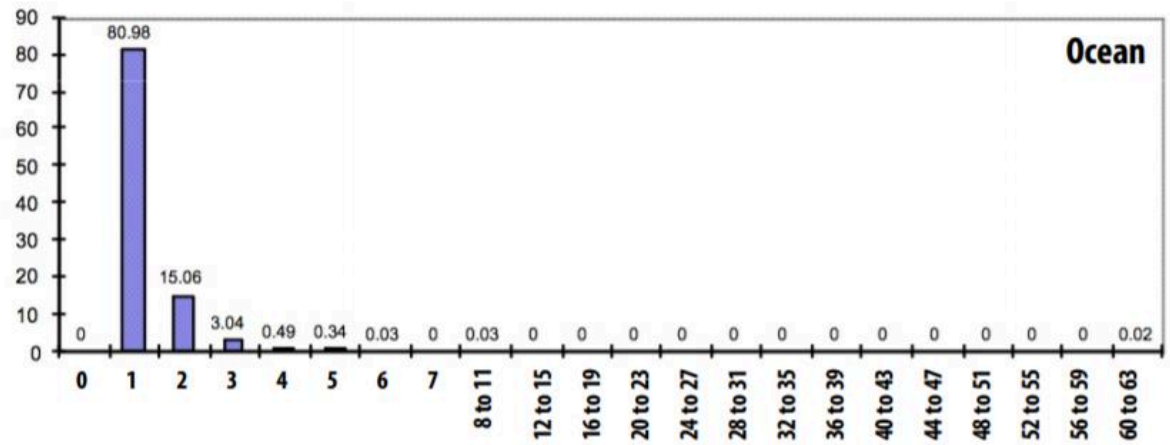


Figure 26: Number of sharers in a 64-core system

A full bit fector approach is therefore useless, because it would have a lot of 0s. A better solution is to use just 10 pointers to the nodes which are holding the block.

Overflows can be managed in different strategies:
- broadcasting;
- processor grouping;
- sharers invalidation: when there is no more room in the directory for a new sharer, the least recent sharer is removed and the newer takes its place;

4. 1. 8. 2. Sparse directories

Optimization to the full bit vector approach to reduce the factor M.

This solutions aims to reduce the size of each directory entry by considering that the majority of the data does not resides in cache, but in the DRAM, and directories need only to care about data in cache.

A single directory entry can hold a **linked list** of cache lines:

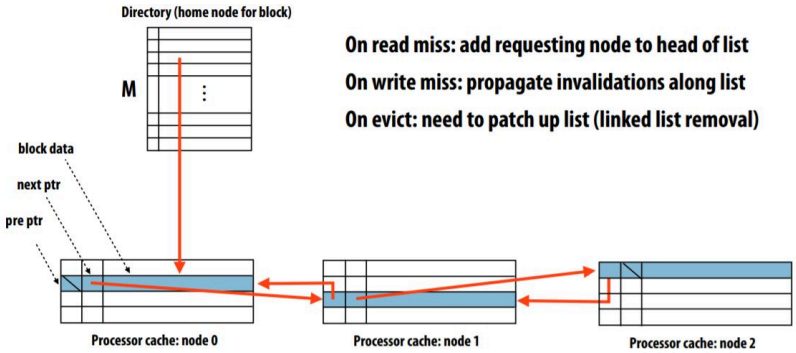


Figure 27: Sparse directories

This solution increases the overhead of writes, since a full list traversal is required. Also evictions are more compliated because a node must be removed from the list.

4. 2. Summary

Cache coherence is mandatory to make parallel programming easier.

Modern computers implement the snoopy protocol. This is also the reason why the number of cores is limited (4 to 12).

Cache in GPUs instead is **manually managed** by the programmer. This allow to scale to a very high number of processors.

Distributed machines have different needs. Broadcasting is practically impossible, so there is a directory-based approach.

4. 3. Synchronization

Synchronization is about **protecting access** to shared data in a context where there are multiple processors/threads that may access the same memory area concurrently.

Each thread has a set of both **private variables**, which do not need synchronization, and **shared variables**. Threads communicate implicitly by writing and reading shared variables and coordinate by synchronizing on shared variable.

Synchronization is a must to avoid **race conditions**.

When writing shared memory programs it's important to reason about **atomicity**. Only reads and writes are atomic by default. When an operation is atomic, all computations happen in **private** registers.

Atomic operations are a fundamental building block for multi-threaded applications.

Synchronization is the process of using atomic operations to ensure cooperation between threads.

Synchronization implements **mutual exclusion**: only one thread is performing a specific work at a given time, while other threads **wait** for their turn.

A **critical section** is a portion of code that only a thread at a time can execute, thus access to critical sections must be synchronized between threads.

Synchronization can be of different types:

- mutual exclusion: a thread performs work, the others wait;
- **event synchronization**: an event is raised when the execution of some threads reaches a given point:
  - **global** synchronization: all threads must wait for the others to finish before resuming their computations. This is also known as a **join point** and it's implemented using **barriers**;
  - **group** synchronization: join operation is restricted to a limited number of threads;
  - **point-to-point**: synchronization is only between 2 threads (or very few of them)

A **lock** is a mechanism that prevents other threads from doing something. A lock is applied before entering in a critical section and it's released when leaving it.
If other threads reach a portion of code with an applied lock, they wait for their turn.

Locks imply **waiting**, so it's very important to keep the **granularity** of critical sections small.

Correctness is another aspect that should be checked througly when writing multithreaded programs.

Synchronization requires **hardware support**. It's not possible to write multithreaded programs with only synchronized reads and writes:

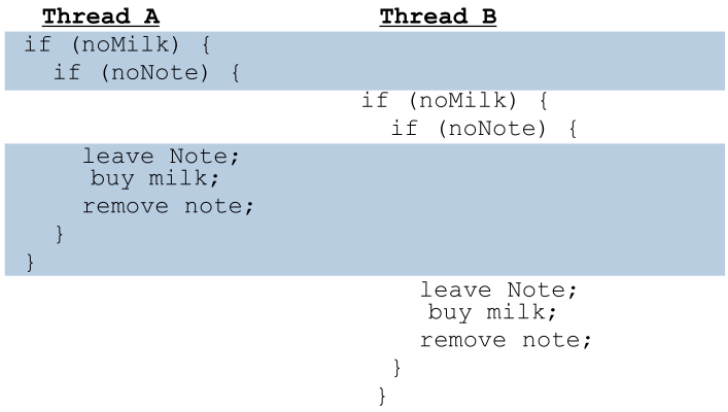


Figure 28: Non-correct multithreaded program. The check `if (noNote)` must be atomic.

`test&set(&address) { result = M[address]; M[address] = 1; return result; }`	checks if a memory address has value 0 and, if so, sets the value to 1. Does nothing if the value is already 1.
`swap(&address, register) { temp = M[address]; M[address] = register; register = temp; }`	swaps atomically the content of two memory addresses
`compare&swap(&address, reg1, reg2) { if (reg1 == M[address]) { M[address] = reg2; return success; } return failure; }`	performs the atomic swap only if the content of the memory address has a specific value

Table 7: Hardware atomic instructions

4. 3. 1. Implementing locks with `test&set`

```
int value = 0;
Acquire() {
  while (test&set(value) == 1);
}
Release() {
  value = 0;
}
```

Listing 2: Lock implementation using `test&set`

If the lock is free, `Acquire()` will set `value` to `1` and will get the lock, otherwise it will wait until `value` is 0.

```
milkLock.Acquire()
if (nomilk)
  buy milk
milklock.Release()
```

Listing 3: Usage of `Acquire` and `Release` primitives. The code between them is the critical section.

4. 3. 2. Synchronization performance impact

Different aspects to keep in mind:

- **latency**: how long does it take to execute a critical section;
- **bandwidth**: how long does it take when many other threads are waiting to execute;
- **traffic** on the bus;
- **storage** requirements;
- **fairness**: all threads should be able to perform work

Locking though `test&set` implies **cache invalidation**, because a value is written into memory. This also implies a lot of **cache coherence** traffic, which kills performance when the number of processors is high:

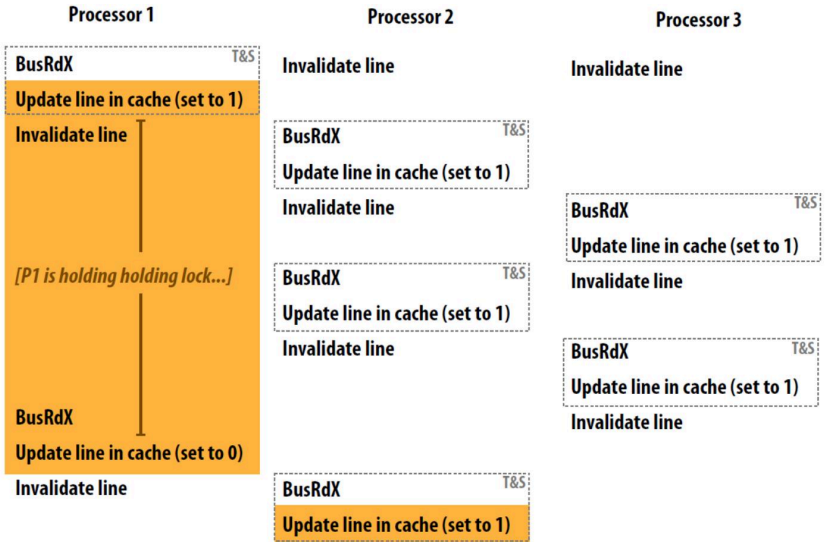


Figure 29: `test&set` coherence traffic

Coherence traffic can be reduced a lot by using private variables and pointers.

```
void lock(volatile int* l) {
  while (1) {
    while (*l != 0);
    if (testandset(*l) == 0) {
      return;
    }
  }
}

void unlock(volatile int* l) {
  *l = 0;
}
```

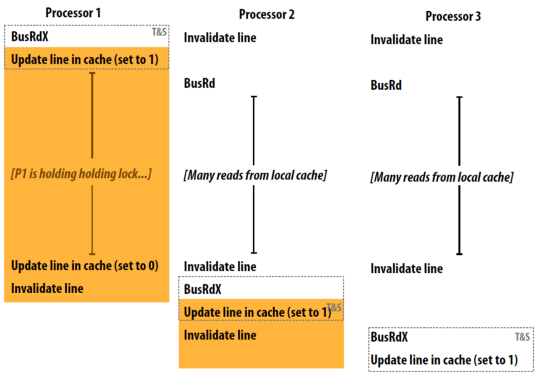


Figure 30: Implementation and coherence traffic of `test&test&set` lock

`test&test&set` lock has a higher latency than plain `test&set`, but it generates much less coherence traffic and thus it has better performance.

Neither `test&test&set` nor `test&set` provide any kind of **fairness**.

`test&test&set` still keeps the bus pressure high because each thread is costantly checking if the lock has been released. Bus traffic can be reduced by implementing an **exponential backoff**:

```
void lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (testandset(*l) == 0) {
      return;
    }
    delay(amount);
    amount *= 2;
  }
}
```

Listing 4: `test&test&set` with exponential backoff

If a resource is busy, it does not make sense to insist on getting the lock. If a resource is still busy after some time, it means that the lock owner is taking much longer than expected to finish its work, so other processors should wait more.

Exponential backoff can cause severe unfairness, because processors that come to the critical section later wait much less thant the first ones.

A **ticket lock** can be used to improve fairness:

```
struct lock {
  volatile int next_ticket;
  volatile int now_serving;
};

void lock(lock* l) {
  int my_ticket = atomicIncrement(l->next_ticket);
  while (my_ticket != l->now_serving); // wait
}

void unlock(lock* l) {
  l->now_serving++;
}
```

Listing 5: Ticket lock example (first-come first-served logic)

In this example each thread is using the same **shared** variable, so they must be synchronized. Typically an **array-based** lock is preferred:

```
struct lock {
  volatile int status[P]; // P = number of processors
  volatile int head;

  int my_element;

  void lock(lock* l) {
    my_element = atomicIncrement(l->head);
    while (l->status[my_element] == 1);
  }

  void unlock(lock* l) {
    l->status[next(my_element)] = 0;
  }
}
```

Listing 6: Array-based lock

The polling loop checks only a specific flag inside the shared `status` array.

This adds a memory overhead to the locking mechanism.

4. 4. Shared memory issues: consistency

Consistency ensures that the ordering of memory operations to **multiple locations** is maintained.

Coherence is not enough, because it ensures ordering only for memory operations that involves the **single** location. In general programs use **many** variables to control their behaviour, therefore cache consistency is needed.

Instruction reordering, which is very common in modern processors (e.g. out-of-order execution), must ensure memory consistency.

Single thread		**Multithread**	
Ordering 1	**Ordering 2**	**Thread 1**	**Thread 2**
`A = 5;` `flag = 1`	`flag = 1;` `A = 5;`	`A = 5;` `flag = 1;`	`while (flag == 0);` `printf(A);`

Instruction reordering in single thread programs is not a big deal if there are no dependencies between variables. In multithreaded programs, however, reordering must be done with care because it's common to implement **inter-processor** control schemes: instructions may appear to be independent when looked at single thread level, but when looked at multithreaded level they are not independent.

4. 4. 1. Memory operation ordering

1. $W \to R$: the write must complete before the read;
2. $R \to R$: the first read must complete before the second;
3. $R \to W$: the read must complete before the write;
4. $W \to W$: the first write must complete before the second

4. 4. 2. Sequential consistency

A sequential consistent memory model preserves all those orderings when code is executed, but only for the **same thread**. It cannot guarantee order across multiple threads.

Thread 1

```
B = 0;
A = 1;
if (B == 0)
  printf("Hello");
```

Thread 2

```
A = 0;
B = 1;
if (A == 0)
  printf("World");
```

If this code is executed on a sequential consistent memory model, the output can either be nothing, `Hello` or `World`, but it can never be `Hello World`.

Sequential consistency makes programs easier to understand for programmers: what they see in the code is also what the machine sees when executing the program. However sequential consistency is very expensive, because all memory accesses should be delayed until all invalidations complete. For this reason sequential consistency has never been used in practice.

4. 4. 3. Relaxed memory consistent

Since sequential consistency is expensive, some orderings are **relaxed** if they are acting on different memory addresses. This allows to implement **memory latency hiding**: while a processor is writing `A`, the other can read `B` (as long as these variables do not depend on each other).

Latency hiding on writes is implemented by adding a **write buffer** to the load/store unit. As long as the processor has put a "write request" in this buffer, it can forget about it (it will be carried on asynchronously) and it can proceed with its work. Writes in this buffer are applied sequentially, so the $W \to W$ ordering is preserved, but the $W \to R$ ordering is relaxed.

In **total store ordering** (TSO) a processor can move its own reads before its own writes: P1 can read `B` before its write to `A` is seen by other processors, but other processors must wait to see the new value of `A` before reading `B`.

In **processor consistency** (PC) instead this is not required: every processor can read `B` before the write to `A` is seen by all the others.

Both TSO and PC only relax $W \to R$ ordering. Other orderings are still preserved.

Partial store ordering (PSO) allows $W \to W$ reordering too:

Thread 1

```
A = 5;
flag = 1;
```

Thread 2

```
while (flag == 0);
printf(A);
```

Listing 7: With PSO, T2 may observe changes on `flag` before changes on `A`.

Weak odering and **release consistency** allow **all** orderings to be violated. Each processor must support special synchronization operations, called **memory fences**:

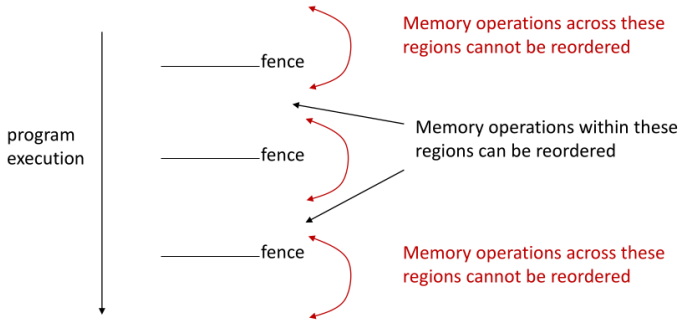


Figure 31: Reordering with memory fences

- memory accesses before the fence must complete before the fence issues;
- memory accesses after the fence cannot begin until fence instruction is completed

A fence may look like a barrier, but they are very different things:
- a barrier synchronizes execution of multiple threads;
- a fence only blocks memory operations

A fence can be placed after a write to force other threads to wait until the result of that operation is propagated to all threads.

PSO is so widespreaded, even though it further complicates programmer's life, because programmers already need to take care of synchronization anyhow, so they are able to handle memory ordering too.

Fences are typically implicit upon unlocks and barriers.

5. Performance of parallel programming

3 factors affect performance of a parallel program:
- **coverage**: the amount of parallelism in an algorithm;
- **granularity**: how the work is partitioned between processors;
- **locality**: how often a value is found in cache

Programs are composed of sequential parts and parallel parts. The whole point is how big those parallel parts are.

Theoretically, speedup due to parallelization should increase linearly with the number of processors. However, in practice this almost never happens due to the **Amdahl's law**.

Amdahl's law: Performance improvement gained from using some faster mode of execution is limited by the fraction of time that execution model can be used.

This means that if there are few parallel sections, or each parallel section is very small, the speedup obtained by parallelization is very limited.

If p is the fraction of work that can be parallized and n is the number of processors, the speedup can be obtained by:

$$\text{speedup} = \frac{\text{old execution time}}{\text{new execution time}} = \frac{1}{(1-p) + \frac{p}{n}}$$

Linear speedup can be achieved only for $p = 1$, i.e. the **whole program** is parallelizable, which is quite rare in practice. However, as p decreases, the obtained speedup drops dramatically too.

Amdahl's law explains why performance improvement across multicore CPU generations stays around 22%, which is much less than the 52% of the single core era.

Amdahl's law is also another reason why the number of CPU cores has remained low over time.

Performance of parallel applications are not bounded by Amdahl's law only, but also by the intrinsic overhead of parallelism:
- cost to start a thread;
- communication cost between threads;
- extra code required to make the program parallel

Parallism is a **tradeoff**. An algorithm needs sufficiently large units of parallel work, but not so large that not all processors can work together.

5. 1. Granularity

Granularity is a measure of the ration between useful work and additional work implied by the parallelism.

Granularity affects **load balancing**.

Fine-grain parallelism	**Coarse-grain parallelism**
• higher overhead, because there are more synchronization instruction; • better load balancing	• overhead is better amortized • difficult to load balance efficiently (not all processors have the same amount of work)

Table 9: Fine grained vs coarse grained parallelism

Load balancing is extremely important when using barriers, because the slowest processor dictates the overall execution time.

Static load balancing	**Dynamic load balancing**
• the amout of work per processor is defined at **complie time**; • works well if all processors are homogeneus, which is not always the case (e.g. NUMA machines); • it's not always possible to distribute the work equally between all processors, e.g. due to **data dependencies**	• there is **task queue** from which a processor takes a task when it's idle; • helps in reducing imbalances in work distributions; • it's the **processor itself** that asks the runtime system for work to do

Table 10: Static vs dynamic load balancing

5. 2. Communication and synchronization

When executing an application in parallel, processors need to **communicate** partial results and to **synchronize**.

Performance impact of communication depends on the available **bandwidth** and **latency**.

Interconnection can either be implemented using a **shared bus** or a **network-on-chip**:

- shared bus provides lower latency, but also lower bandwidth because processors can talk one at a time;
- NoC provides lots of bandwidth, but has a higher latency. Latency is also not uniform because it depends on the number of hops in the network.

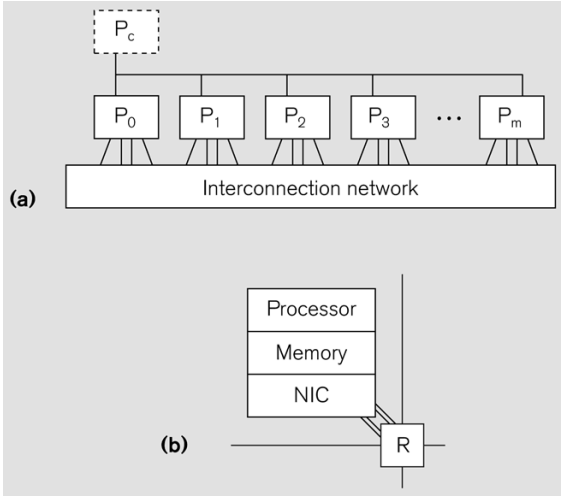


Figure 32: Shared bus vs NoC

Communication between processors cannot be avoided when executing a parallel application. Latency hiding is the only way to reduce its cost.

Latency hiding can be implemented in different ways:
- overlapping message sending with computation;
- data prefetch;
- switching to other task when a processor is idle;
- task **pipelining**

Every instruction of a program can be modeled with 2 types: fetching data from the memory and executing work. These two types of instructions can be **pipelined** to keep both the CPU and the MEM unit working:

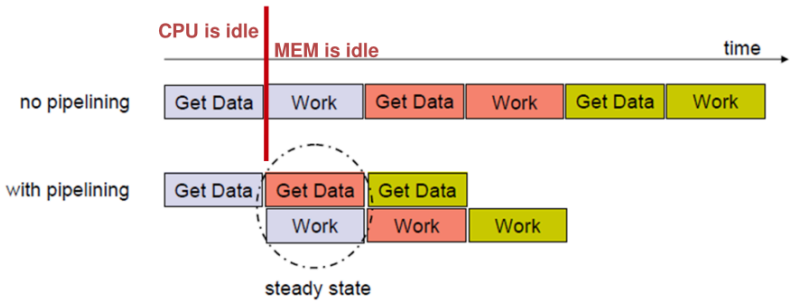


Figure 33: CPU and MEM utilization with and without pipelining

5. 3. Locality

In a shared memory multiprocessor, memory access can either be:
- uniform (UMA): there is a single, centralized main memory. Memory latency is the same for all processors;
- non-uniform (NUMA): memory is physically partitioned between different processors, but they all share the same address space. Memory latency is different for each processor

Partitioned global address space (PGAS) means that every processor is able to address any portion of the memory, but it's not able to access it physically. This is the case of **distributed** shared memory.

Even though processors are executing the same code, there could be **locality** problems because not all processors have the same latency in reaching the memory.

Two key aspect to optimize the memory hierarchy:
- reducing **memory latency**:
- maximize the **memory bandwidth**

Compiler back-ends help a lot in exploiting locality.

Improving locality in programs deals with **reusing data**:
- **temporal locality**: when reading a memory address, there is a high probability that the same address is read again after a short time;
- **spatial locality**: when reading a memory address, there is a high probability that in a short time also addresses near that one will be read

```
for (int i = 0; i < N; i++)
  for (int j = 1; j < N - 1; j++)
    A[j] = A[j + 1] + A[j - 1]
```

Listing 8: This loop uses both temporal and spatial locality

Programs should do most of their work on **local** data, because local memories can be much smaller (and thus faster).

When iterating over matrices, **loop permutation** can be benificial to improve data locality:

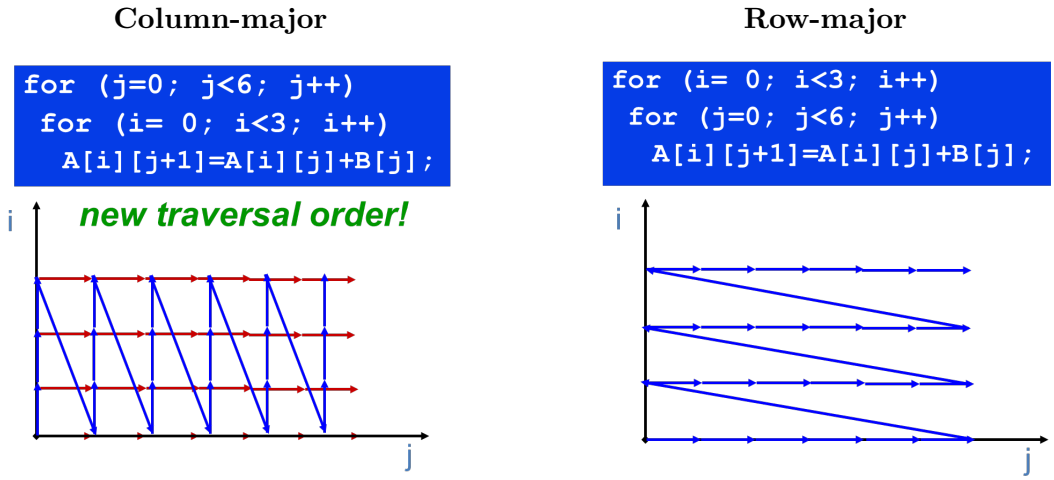


Figure 34: Row-major traversal vs column-major traversal

5. 3. 1. Tiling

Tiling (or blocking) is a way to reorder loop iterations when the data structure is too big to fit in a single cache line.

The iteration space is splitted in different blocks, where each one is small enough to fit in a cache line. This improves data locality also when the data structure is very big.

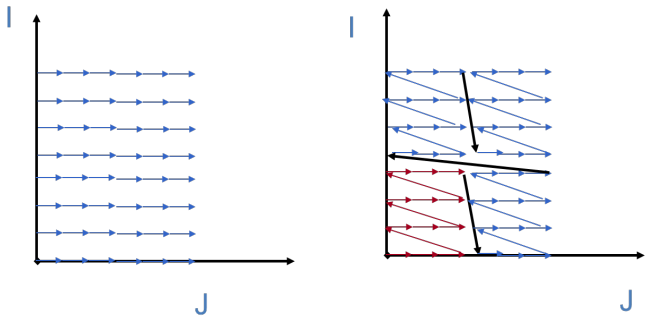


Figure 35: Tiling example

Tiling also allows to reduce the number of cache evictions because when a block of the data structure is read from the main memory, all of its content is used.

Tiling can help to achieve a **superlinear speedup** because it improves also the sequential part of the program.

5. 3. 2. Memory banking

Memory is often organized in several **independent banks**:

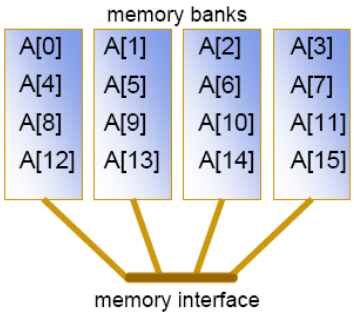


Figure 36: Memory banking

Each bank has a single port to access it, so different threads should use different banks. This can be implemented by changing the parallelization scheme or the memory layout of the data.

6. Parallel programming in practice

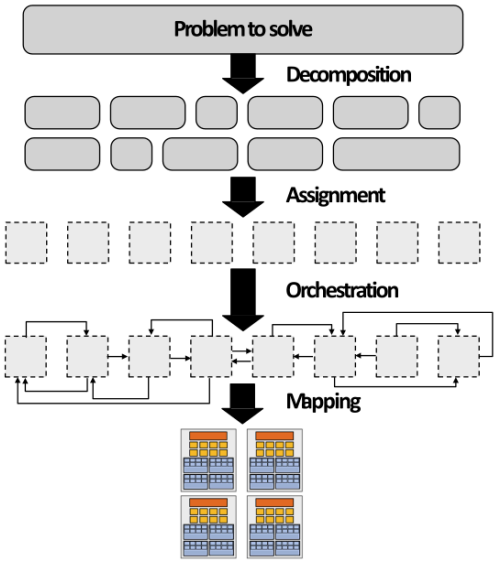


Figure 37: 4 steps of parallelization

6. 1. Decomposition

In most cases, decomposition must be manually performed by the programmer. Automatic decomposition by the compiler continues to be a challenging research problem.

Programs usually decompose naturally in **function calls** and **loop iterations**.

Task decomposition must be:

- **flexible**, both in the number of tasks and in the size of each task;
- **efficient**: each task should have enough work to do to amortize the overhead of parallelization;
- **simple**: the code has to remain readable and easy to debug

If the program uses some data structure (array, matrix, tree, etc.), **data decomposition** is likely possible. This decomposition executes the same task on multiple elements of the data structure at the same time.

Functions that deal with the same data structure can often be **pipelined**.

Bernstein's condition: Given R_i the set of memory locations read by task T_i and W_j the set of memory locations written by task T_j, T_i and T_j can run in parallel if and only if **all** of these are verified:

$$R_i \cap W_j = \emptyset$$
$$R_j \cap W_i = \emptyset$$
$$W_i \cap W_j = \emptyset$$

6. 1. 1. Decomposition example: A2D-grid based solver

In this algorithm, each row element depends on both the previous column and the previous row:

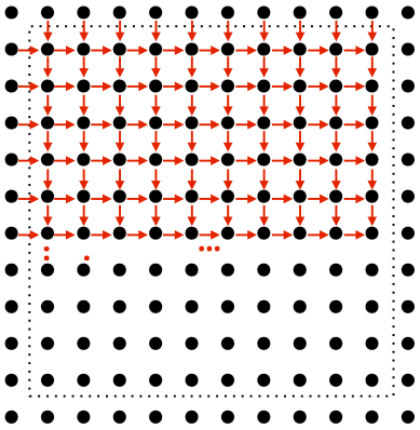


Figure 38: Dependencies in A2D-grid based solver

When looked in this way, it seems that no parallelism can be extracted from this algorithm. However a careful analysis can spot that parallelism does exist along matrix **diagonals**:

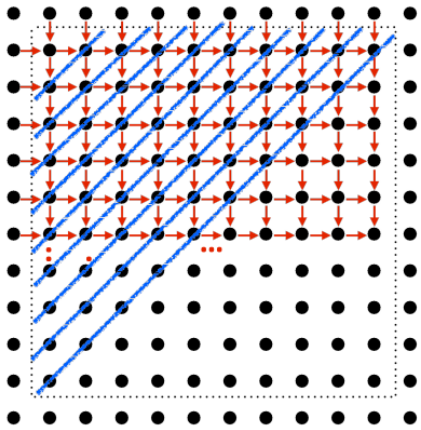


Figure 39: Parallelism along diagonals in the A2D-grid based solver

Task decomposition based on diagonal elements however is not optimal, because:

- each task has a different amount of work (each diagonal has a different length);
- code must be drastically changed to iterate through diagonals instead of rows/columns

A better approach is to change the order in which grid cells are updated:

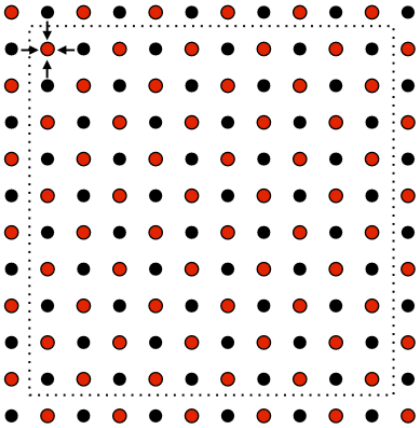


Figure 40: Red-black coloring

Red-black coloring first updates **red** cells, then it moves on **black** cells.

The result of these two algorithm (sequential and parallel) will not be the same, however in this context this is not an issue because Gauss-Seidel is itself an **iterative** algorithm which only provides an **approximation** of the result (not an exact result).

Decomposition requires the programmer to have a deep understanding of the problem.

6. 2. Assignment

There are many more tasks than cores/threads, so it's important to achieve a good **balancing**.

Figure 41: Example of poor balancing. P4 takes 2x longer to complete, which menas that 50% of program exeuction time is sequential

Although decomposition is often in charge to the programmer, assignment is typically performed automatically by the compiler or the runtime system.

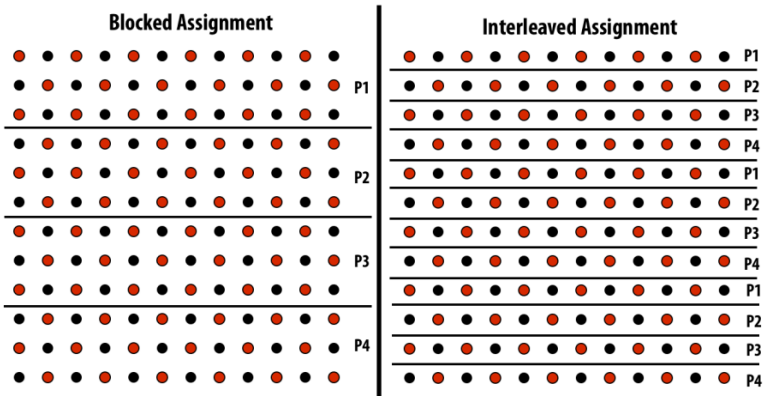


Figure 42: Two possible assignment strategies for the grid solver example. Which one is better depends on the system the program is run on.

Assignment can either be **static** or **dyamic**.

Static assignment does not have any runtime overhead, but it's applicable only when all tasks have a similar execution time or at least the execution time of each task is **predictable**, because this allows to have a good balance **on average**.

The simplest way to implement dynamic assignment is through a **centralized queue**: all tasks are pushed in this structure at the beginning of program's execution, then each threads pops one task from this queue when it's idle.

Task **granularity** is very important in dynamic assignment. Small tasks help achieve a better balancing, but they also generate more overhead.

Dynamic assignment should be smart: the runtime system should consider the state of other workers, expected execution time, dependencies, etc. when assigning a task to a worker (the runtime system should not choose the task to assign randomly).

Access to the task queue should be **synchronized** between processors. When there are a lot of them it may be benificial to use **private** work queues (one for each processor) instead of a global one. When a private queue is empty, the processor can "steal" work from another one.

6. 3. Orchestration

Orchestration means adding code to manage communication and synchronization between workers. The goal is to reduce the overhead of parallelization.

Orchestration depends on both the architecture and the cost of available synchronization/communication APIs (if they are cheaper, adding more of them it's not a big deal).

Back to the grid solver example, initial orchestration can be done like this:

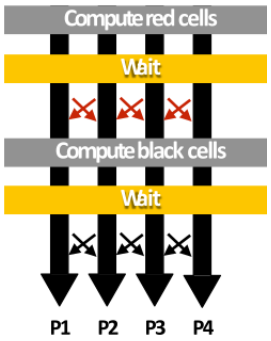


Figure 43: Grid solver initial orchestration idea

By looking at dependencies, it's clear that blocked assignment requires much less communication between workers, so this approach should be preferred:

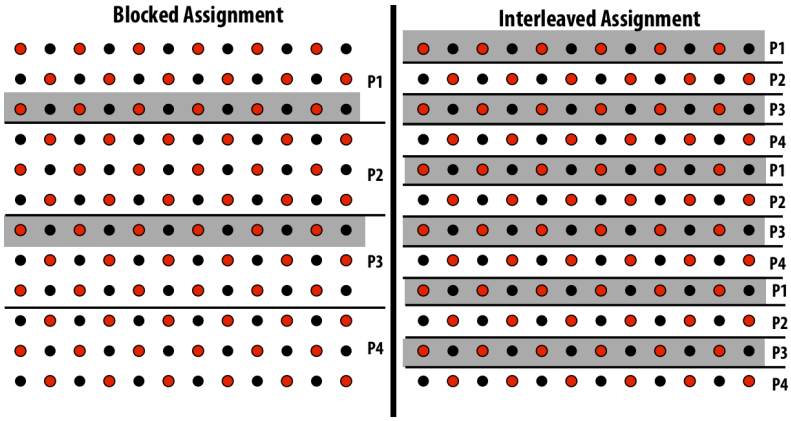


Figure 44: Blocked vs interleaved assignment communication

Orchestration is implemented differently based on the memory architecture of the system.

6. 3. 1. Shared address space

Here synchronization is programmer's responsibility, who should place barriers and locks properly.

Programmer can use the `getThreadId()` primitive to differentiate work between threads.

```
void solve(float* A) {
  int threadId = getThreadId();
  int myMin = 1 + (threadId * n / NUM_PROCESSORS);
  int myMax = myMin + (n / NUM_PROCESSORS);

  while (!done) {
    diff = 0.f;

    barrier(myBarrier, NUM_PROCESSORS);

    for (int j = myMin; j < myMax; j++) {
      foreach (red cell i in this row) {
        float prev = A[i, j];
        A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j],
A[i,j+1]);

        lock(myLock);
        diff += abs(A[i,j] - prev);
        unlock(myLock);
      }
    }

    barrier(myBarrier, NUM_PROCESSORS);

    if (diff / (n*n) < TOLERANCE)
      done = true;

    barrier(myBarrier, NUM_PROCESSORS);
  }
}
```

Listing 9: Pseudo-code example for grid solver

This code can be improved in different ways:

1. the critical section can be moved **outside** the `for` loop by accumulating on a **private** variable (instead of having all threads accumulating on the same shared variable);
2. one barrier can be removed by having **multiple copies** (stored in an array) of the `diff` variable to reduce dependencies between iterations. This trade off (more memory footprint to reduce parallelization overhead) is very common in parallel programming.

6. 3. 2. Distributed memory system (message passing)

The grid can be divided evenly between multiple threads. Each thread may have some **ghost cells**, which are cells that have been replicated from other threads and therefore are ownership of that thread.

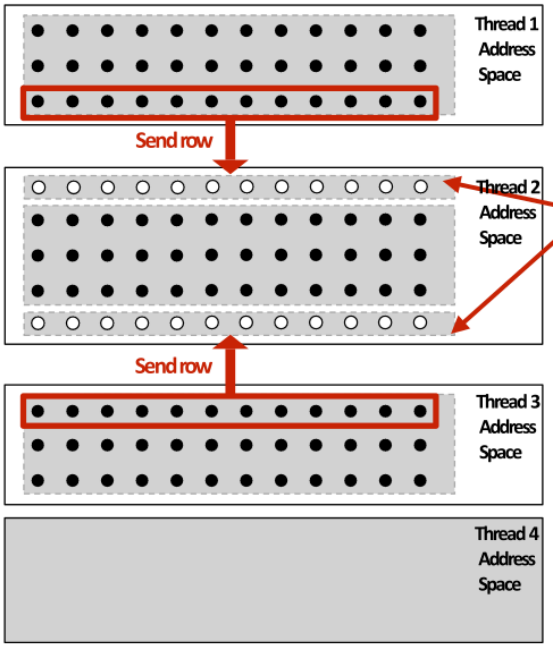


Figure 45: Ghost cells

Every thread must communicate to the next one if there are more iterations to do.

6. 4. Mapping to hardware

Mapping logical workers to hardware execution unit can be done by:

- the operating system;
- the compiler (e.g. by using OpenMP);
- the hardware (e.g. what CUDA does)

Placing related threads on the same processor may help in maximizing **locality** and reducing the overhead of communication. But also placing unrelated threads on the same processor may be beneficial in order to use the hardware more efficiently.

7. OpenMP programming

OpenMP is the standard programming model for **shared memory** systems. It consists in a collection of compiler directives, library APIs and environment variables.

OpenMP requires special support by the compiler. Both `gcc` and `clang` nowdays fully support it.

In the beginning, OpenMP focused only on **loops**, especially **DOALL** loops (loops without dependencies across iterations). OpenMP primarly targeted **expert** programmers who did not want to write low level code to handle parallelization.

Over time, OpenMP has broadened its focus also to things other than loops (e.g. tasks, accelerators/heterogeneous systems) and has spent lot of efforts in making parallel programming easier for everyone, also for non-expert programmers.

OpenMP adheres to the **fork/join model**:

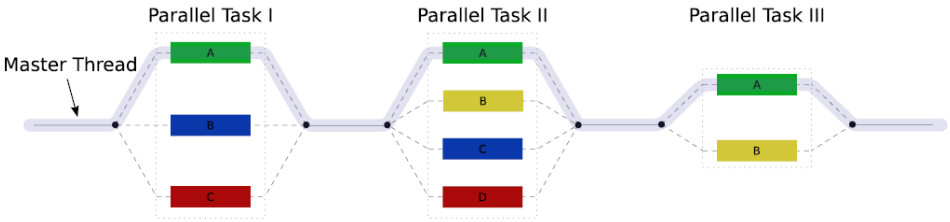


Figure 46: Fork-join model. The program is started by the **master thread**, which spawns new threads (**fork point**). At the end of parallel computation there is an implicit barrier (**join point**) where execution resumes on the master thread only.

OpenMP provides a set of `#pragma` **directives** to deal with:
- definition of **parallel regions**;
- **work sharing** (how work is distributed between threads);
- **synchronization**

Each directive may have zero or more **clauses** to better specify how the directive should work.

OpenMP provides also a set of runtime APIs (e.g. to get the current thread identifier).

7. 1. `parallel` directive

OpenMP fundamental construct to outline parallel computation within a sequential program.

When this directive is encountered, OpenMP creates a **parallel region**. This directive by itself does not create parallelism however, because work is simply **replicated** on all threads. Work-sharing directives must be used to have different threads execute different things.

```
#pragma omp parallel
{
  printf("Hello world!");
} // this is an implicit barrier
```

Listing 10: OpenMP `parallel` directive

When OpenMP encounters a parallel region, by default it creates one thread for each CPU core. This can be changed using the `num_threads(n)` clause.

7. 1. 1. Memory view inside parallel region

In an OpenMP parallel region a variable can either be **private** to the current thread or **shared** between all threads.

OpenMP provides a set of **data sharing clauses** to specify how each variable should be considered.

If no data sharing clauses are used:
- variables defined **inside** the parallel region are treated as **private** variables;
- variables defined **outside** the parallel region are treated as **shared** variables:

```
int a = 5; // shared
#pragma omp parallel
{
  int thread_id = omp_get_thread_num(); // private
  printf("Thread ID: %d\n", thread_id);
}
```

OpenMP provides also 2 additional data sharing clauses:

- `firstprivate`: variable is private, but it's initialized using a shared value (**copy-in** semantics);
- `lastprivate`: variable is private, but when it's written the change is propagated to all other threads when the parallel region ends (**copy-out** semantics)

In both cases storage is **private**, so cache coherence is not a problem.

7. 2. `for` directive

OpenMP `for` directive is a worksharing directive that parallelizes a `for` loop.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
  c[i] = a[i] + b[i];
} // implicit barrier
```

Listing 12: OpenMP `for` directive. An implicit barrier is added after the `for`.

The `nowait` clause may be used to avoid adding the implicit barrier:

```
#pragma omp parallel for nowait
for (...) {
  ...
}
```

Listing 13: OpenMP `nowait` clause

7. 2. 1. `schedule` clause

OpenMP default loop scheduling is **static**: loop's iteration space is divided evenly between all threads.

Programmer can change loop scheduling to **dynamic** by using the `schedule` directive.

Loop scheduling may specify a **chunk factor**, which determines the **unit of work** (i.e. how many iterations are executed together by a single thread).

Static scheduling	$\left\lceil \frac{N}{N_{\text{threads}}} \right\rceil$
Dynamic scheduling	1

Table 11: Default chunk factor for static and dynamic scheduling

Chunk factor determines parallelization **granularity**.

chunk size	T1 iteration space	T2 iteration space
default: $c = \frac{22}{2} = 11$	from `i = 0` to `i = 10`	from `i = 11` to `i = 21`
5	• from `i = 0` to `i = 4`; • from `i = 10` to `i = 14`; • from `i = 20` to `i = 21`	• from `i = 5` to `i = 9`; • from `i = 15` to `i = 19`

Table 12: Example of static scheduling with a different chunk factor (2 threads and 22 iterations). With a chunk factor of 5, work is slightly unbalanced (T1 executes 12 iterations, while T2 only 10).

With static scheduling you know in advance exactly which thread will execute a particular iteration. This is not possible instead with dynamic scheduling, because iterations are assigned to threads as they finish their work.

7. 3. `barrier` directive

Can be used to place an barrier **explicitly**.

7. 4. `critical` directive

Marks a **critical section** i.e. a portion of code that only **one thread at a time** can execute.

```
double area = 0;
int n;
#pragma omp parallel for shared(area)
{
  for (int i = 0; i < n; i++) {
    double x = (i + 0.5) / n;
    #pragma omp critical
    area += 4.0 / (1.0 + x * x);
  }
}
double pi = area / n;
```

Listing 14: OpenMP `critical` directive example

7. 5. `reduction` clause

Implements a **reduction**, a programming pattern in which a partial result is **accumulated** into a variable.

`reduction` clause can be used to avoid the `critical` directive, and therefore to achieve better performance because the code is no longer sequential.

Inside the parallel region, accumulation is done on **private** variables automatically created by OpenMP. When the parallel computation ends, each one of these private variables is summed together and the result is written to the shared variable.

```
double area = 0;
int n;
#pragma omp parallel for shared(area) reduction(+:area)
{
  for (int i = 0; i < n; i++) {
    double x = (i + 0.5) / n;
    #pragma omp critical
    area += 4.0 / (1.0 + x * x);
  }
}
double pi = area / n;
```

Listing 15: OpenMP `reduction` clause example

7. 6. `master` directive

Denotes a portion of code that can be only executed by the **master** thread. No synchronization is implied, because all other threads simply skip this code.

7. 7. `single` directive

Denotes a portion of code that can be executed only by one thread, which may not be the master thread.

`single` is different from `critical` because the first will execute the work **only once**, while the latter will execute the work many times as the number of threads.

A barrier is implicitly added at the end of a portion of code marked with this directive.

7. 8. `sections` directive

This directive was OpenMP first attempt to exploit **task parallelism**. Each `section` directive detones a task which can be executed in parallel with others:

```
#pragma omp parallel sections
{
  #pragma omp section
    v = alpha();
  #pragma omp section
    w = beta();
  #pragma omp section
    y = delta();
}
x = gamma(v, w);
z = epsilon(x, y);
```

Listing 16: OpenMP `sections` example

7. 9. Tasking model

`sections` directive allows to exploit task parallelism, but each task must be **statically** outlined in the code. This is not always feasible, for example when the body of a loop (or of a recursive function) is marked as a task:

```
for (Node* n = l->first; n != NULL; n = n->next) {
  process(n);
}
```

Listing 17: `sections` directive cannot be used here because loop **trip count** is not known in advance

This code can still exploit task parallelism by combining `parallel` and `single` directives:

```
#pragma omp parallel
{
  for (Node* n = l->first; n != NULL; n = n->next) {
    #pragma omp single nowait
    process(n);
  }
}
```

Listing 18: Task parallelism using `single` directive

But this too is not feasible, because the code is counter-intuitive and has performance issues.

```
void visit_tree_node(Node* n) {
  #pragma omp parallel sections
  {
    #pragma omp section
    if (n->left != NULL)
      visit_tree_node(n->left);
    #pragma omp section
    if (n->right != NULL)
      visit_tree_node(n->right)
  }
  process(n);
}
```

Listing 19: Tree traversal is another example of code hard to parallelize using `sections`

The problem in tree traversal is that tasks are created **recursively**. Recursive task creation may not be supported by all compilers and, besides that, creating too many tasks generates a severe overhead.

For these reasons, OpenMP introduced its **tasking model** in its specification 3.0.

Task parallelism in OpenMP allows to parallelize irregular problems, like unbounded loops and recursive algorithms.

OpenMP tasking model is composed of different parts:

- **creating** tasks;
- data scoping (which variables are private, shared, etc.);
- **synchronization** between tasks;
- execution model: how task are created, managed and executed in the system

An OpenMP task is a unit of work which execution may be deferred (but it may also start immediately).

OpenMP tasks are composed of 3 parts:

1. code to execute;
2. data environment;
3. control variables

```
void traverse_list(List* l) {
  for (Node* n = l->first; n != NULL; n = n->next) {
    #pragma omp task
    process(n);
  }
}
```

Listing 20: OpenMP tasks are defined using the `task` directive

A thread that encounters the `task` directive will create a **task description**, which packages code and data environment, for that task.

OpenMP tasking model is highly composable: tasks can be nested and can be used together with other OpenMP constructs (`for`, `sections`, etc.).

There is no synchronization between OpenMP tasks by default. In the example above, when `traverse_list` returns it's not guaranteed that all tasks have been executed. This can be changed using the `taskwait` directive, which acts like a barrier for tasks:

```
void traverse_list(List* l) {
  for (Node* n = l->first; n != NULL; n = n->next) {
    #pragma omp task
    process(n);
  }
  #pragma omp taskwait
}
```

Listing 21: OpenMP `taskwait` example

A **team** is a set of threads belonging to the parallel region that encloses a code block. Tasks are executed by a thread of the team that created it.

Typically, the thread that creates the task is not the one who will execute it, but this distinction is not mandatory (the same thread may both create and then execute the task).

When a parallel region begins, one implicit task is created for each thread, so there are always at least as many tasks as threads. There is also always a first-level task that encloses all the others.

Threads can **suspend** (and then resume) tasks. Task scheduling is managed by OpenMP runtime and it's transparent to the programmer.

7. 9. 1. `depend` clause

Used to specify **dependencies** between tasks and therefore have **point-to-point synchronization** between them.

A dependency may be in task's **inputs**, **outputs** or both.

```
#pragma omp task shared(x) depend(out: x)
x = do_something();
#pragma omp task shared(x) depend(in: x)
do_something_else(x);
#pragma omp task shared(x) depend(in: x)
do_something_else_again(x);
```

Listing 22: OpenMP `depend` clause example

7. 9. 2. `if` clause

This clause is not specifically related to OpenMP tasking model, but one of its main uses is to avoid task creation when the amount of work is very limited:

```
#define CUTOFF 20
int fib(int n) {
  int x, y;
  #pragma omp task shared(x) if (n > CUTOFF)
  x = fib(n - 1);
  #pragma omp task shared(y) if (n > CUTOFF)
  y = fib(n - 2);
  #pragma omp taskwait
  return x + y;
}
```

Listing 23: This example uses the `if` directive to avoid creating tasks with a small amount of work to do. This helps reduce the overhead because the number of tasks will be much lower.

7. 10. OpenMP accelerator model

OpenMP accelerator model deals with **heterogeneous systems**. One of its main goal is to unify the programming standard across different architectures (GPU, SIMD units, etc.).

Each architectural component is considered as a **device**. There is always one **host** device and there may be multiple **target** devices. System memory is divided between **host memory** and **device memory**.

Each device is composted of one or more **compute units**. Each compute unit is divided into one or more **processing elements**.

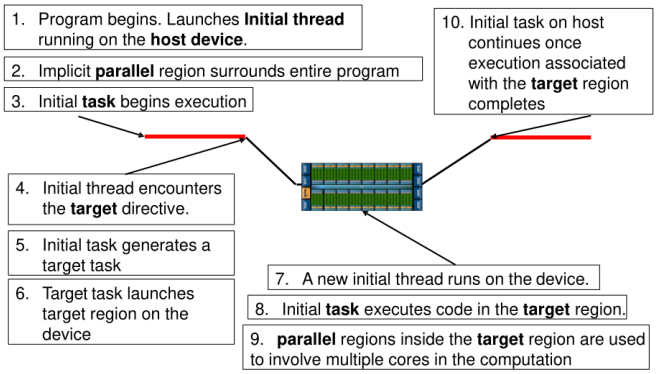


Figure 47: OpenMP execution model using accelerator

7. 10. 1. `target` directive

Denotes a portion of code that runs on the accelerator.

Similar to `parallel` directive, `target` directive by itself does not create parallelism: it **replicates** the same work on multiple accelerator threads.

When using the `target` directive, `map` clause allows to specify how variables are mapped between host and target data environments.

```
#pragma omp target map(to: b, c, d) map(from: a)
{
  #pragma omp parallel for
  for(int i = 0; i < n; i++)
    a[i] = b[i] * c + d;
}
```

Listing 24: `target` directive example

Offloading to the target is **synchronous** by default (the host waits until the target has finished). This can be changed by using the `nowait` clause.

7. 10. 2. `teams` directive

Worksharing directive that creates a **legue** of thread teams.

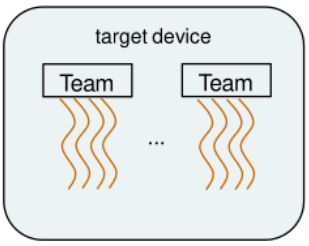


Figure 48: Legue of threads

Thread in different teams cannot synchronize with each other.

Only the master thread of each team executes the code inside the `teams` region.

7. 10. 3. `distribute` directive

Worksharing directive that distributes the iteration space of a loop across the master threads of each team executing the region.

Scheduling is static by default and can be changed using the `dist_schedule` directive.

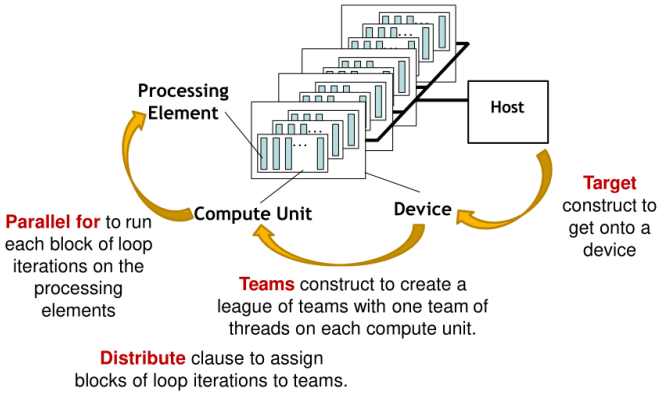


Figure 49: OpenMP accelerator model

7. 10. 4. `declare target` directive

It's not possible to invoke other functions inside a `target` region unless that function is marked with `declare target` directive:

```
#pragma omp declare target
int foo() {
  ...
}
#pragma omp end declare target
int main() {
  #pragma omp target
  foo();
  return 0;
}
```

Listing 25: `declare target` example

8. GPU acceleration

A GPU is an **accelerator** for graphical work. GPUs are designed to exploit the high amount of **data** and **pipeline** parallelism in these problems (e.g. increasing the brightness of an image is an addition of a constant to a matrix of pixels).

A GPU is an heterogeneous multi-processor chip highly specialized for graphical operations:

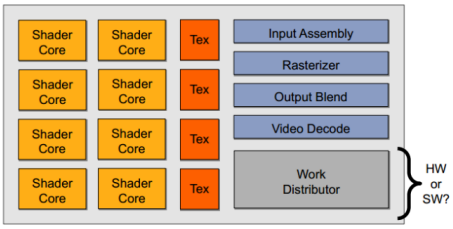


Figure 50: Overview of GPU architecture

Shader cores are **programmable processors**.

The work distributor distributes work among different shaders. It can be implemented both in hardware or in software.

Design of a GPU core starts from the CPU core:

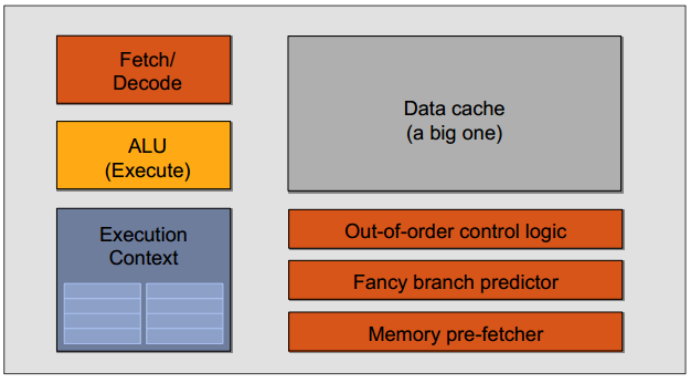


Figure 51: CPU core architecture

CPU core is much complex (branch prediction, out-of-order execution, etc.) and this limits the amount of them that can be on a chip. GPU cores cannot be so complicated, because there is the need to have **thousands** of them.

Therefore, a GPU core is like a CPU core slimmed down. There are multiple ALUs with a **single** fetch/decode unit for all of them:

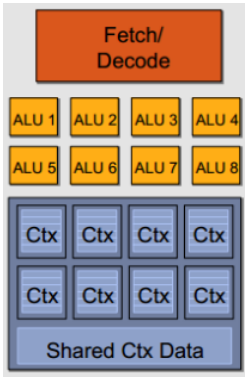


Figure 52: GPU core architecture

This architecture exploits **SIMD parallelism**: the same instruction is applied on different streams of data; there is no need to fetch/decode the same instruction multiple times.

Compilers transform scalar instructions into **vector instructions** to make them work on GPU hardware.

GPU cores can also use **task parallelism**, but the granularity is somewhat limited by the vector instructions.

SIMD processing does not imply SIMD instructions. Instructions should be transformed in SIMD form either manually by the programmer or automatically by the hardware.

Conditional execution is a pain for this type of architecture because the parallelism is broken: while one ALU is evaluating a conditional expression, others must wait for it to finish.

Stalls caused by dependencies between instructions are avoided using a large **context storage**. This helps having a large number of "waiting" fragments, so when one of them stalls there is a high probability that another one is ready to execute.

There is a tradeoff between the number of contexts and the size of each one. Higher context number means better latency hiding, because there can be more tasks in the queue, but it also mean that the size of each context cannot be too big.

Context switching can be performed either by hardware or software.

8. 1. Memory architecture

CPU cores have multiple levels of cache to reduce memory latency.

In a GPU the problem is no longer the latency of the memory, but is its **bandiwdth**, because GPU deals with SIMD instructions. Therefore GPUs need an entire different technology for their memories (e.g. GDDR5) optimized for bandwidth instead of latency.

Memory bandwidth is a critical resource when developing GPU applications. Programmers needs to keep low the pressure on the memory system, for example by increasing the arithmetic intensity (do more math instead of reading already computed results) or by using device memory (e.g. CUDA shared memory).

An efficient GPU workload should:

- have thousands of independent pieces of work, so all (or most of them) ALUs can be used and lots of context are available for switching;
- not be limited by memory bandwidth

9. CUDA programming

CUDA is a programming model for NVIDA GPUs. It's much more complex than OpenMP, because code has been changed more heavily than OpenMP.

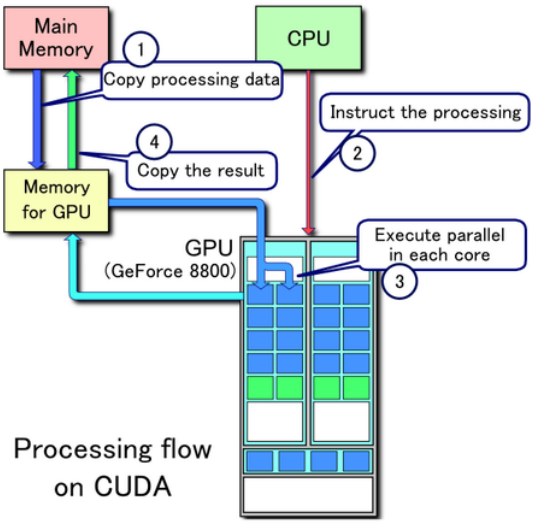


Figure 53: CUDA execution model

The CPU sees the GPU as an external processor, with its own memory, capable of executing lots of threads in parallel.

Data-parallel parts of the application are outlined within a **kernel** function that every thread on the device executes[4].

GPU threads are much more lightweight than CPU ones, but to have the maximum efficiency there must be thousands of them executing.

In a CUDA application, sequential code runs on the CPU, while parallel parts are offloaded to the GPU.

[4]SIMT paradigm: single instruction multiple threads.

```c
int main (int argc, char* argv[])
{
  int i;
  const int N = 1000;
  double u[N], v[N], z[N];

  initVector(u, N, 1.0);
  initVector(v, N, 2.0);
  initVector(z, N, 0.0);

  printVector(u, N);
  printVector(v, N);

  // z = u + v
  for (i=0; i<N; i++)
    z[i] = u[i] + v[i];

  printVector(z, N);
  return 0;
}
```

```c
void gpuVectAdd(const double *u,
          const double *v, double *z)
{
  // use GPU thread id as index
  i = getThreadIdx();
  z[i] = u[i] + v[i];
}

int main(int argc, char *argv[])
{
  ...

  // z = u + v
  {
    // run on GPU with N threads
    gpuVectAdd(u, v, z);
    // close threads
  }
  ...
}
```

Figure 54: Sequential program vs CUDA parallel program

CUDA threads are grouped in **blocks**. Blocks are in turn grouped in **grids**.

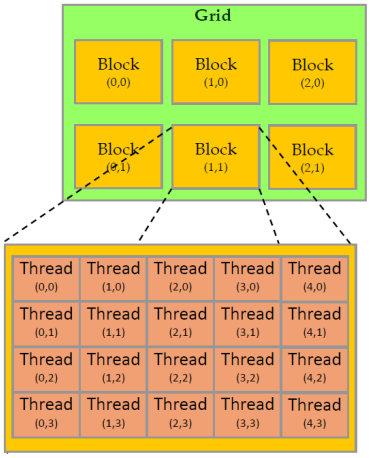


Figure 55: CUDA grid

Limits on the number of threads/blocks depend on the architecture.

9. 1. How to port sequential code into CUDA

1. identify parallel parts of the code and isolate them in kernel functions;
2. identify data needed by each kernel that must be transferred on the device;
3. implement the required CUDA kernels;
4. modify the host code to invoke CUDA kernels

Every CUDA thread executes the same kernel function, but it operates on different portions of the same data structure.

CUDA provides different qualifiers for kernels:

- `__global__`: executed **asynchronously** on the device, callable from the host only;
- `__device__`: executed on the device, callable on the device only;
- `__host__`: executed on the host, callable on the host only

`cudaMalloc()` and `cudaMemcpy()` APIs can be used to allocate and to copy data to/from the device respectively. These are synchronous APIs, but CUDA also provides an asynchronous variant.

When a kernel is launched on the device:

1. each block is assigned to a streaming multiprocessor;
   - there is no specific order in which blocks execute;
   - once a block is assigned to a SM it cannot migrate to other SMs;
2. threads in each block are grouped in teams called **warps**;
3. the scheduler selects a warp for execution among the available blocks;
4. idle blocks execute when another block stalls or has completed execution;
5. every CUDA core executes one of the threads inside a warp

CUDA kernels scale transparently over different architectures.

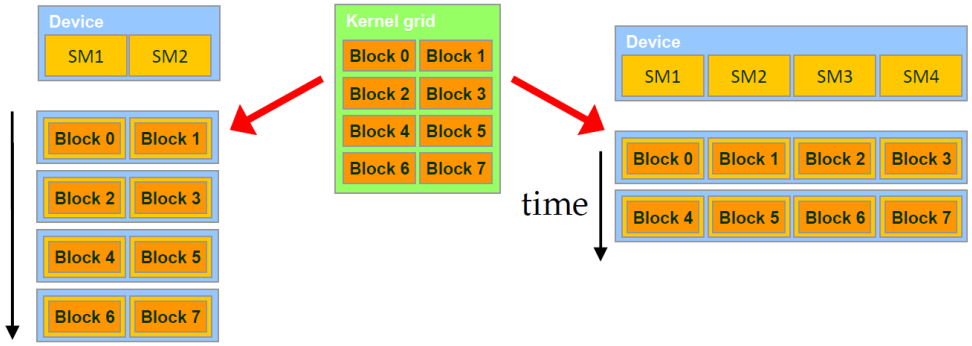


Figure 56: Example of high CUDA scalability

Mapping between blocks and SM is not determined statically; it depends on the available execution capacity.

Threads inside the same block execute concurrently on the same SM, while threads of different blocks execute concurrently over different SMs.

Registers are a space where each thread can store its data. Registers are segmented by threads: the scheduler does not swap the content of the registers when performing context switching, but different threads access different sections of the register.

The number of registers required by a thread decreases the parallelism of the application, because there is less space to store data for other threads and therefore there is less space to perform context switching efficiently.

Example: if an architecture has 32.768 regsiters, each block has 32×8 threads and the kernel needs 30 registers, a single SM can run only $\left\lfloor \frac{32.768}{30 \times 32 \times 8} \right\rfloor = 4$ blocks.

9. 2. CUDA driver API

CUDA currently supports 2 APIs:

- CUDA driver API, which allows very low-level operations
- C for CUDA (CUDART), which is much more easier to use and it's based on the CUDA driver API

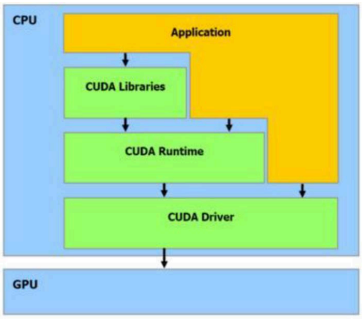


Figure 57: CUDA APIs scheme

```
// driver API
// initializeCUDA
err= cuInit(0);
err= cuDeviceGet(&device, 0);
err= cuCtxCreate(&context, 0, device);

// setup devicememory
err= cuMemAlloc(&d_a, sizeof(int) * N);
err= cuMemAlloc(&d_b, sizeof(int) * N);
err= cuMemAlloc(&d_c, sizeof(int) * N);
// copy arrays to device
err= cuMemcpyHtoD(d_a, a, sizeof(int) * N);
err= cuMemcpyHtoD(d_b, b, sizeof(int) * N);

// preparekernellaunch
kernelArgs[0] = &d_a;
kernelArgs[1] = &d_b;
kernelArgs[2] = &d_c;

// loaddevicecode (PTX or cubin. PTX here)
err= cuModuleLoad(&module, module_file);
err= cuModuleGetFunction(&function, module,kernel_name);

// executethe kernelover the <N,1> grid
err= cuLaunchKernel(function, N, 1, 1, // Nx1x1 blocks 1, 1, 1, // 1x1x1 threads0, 0, kernelArgs, 0);
```

```
// runtime API
// setup devicememory
err= cudaMalloc((void**)&d_a, sizeof(int) * N);
err= cudaMalloc((void**)&d_b, sizeof(int) * N);
err= cudaMalloc((void**)&d_c, sizeof(int) * N);

// copy arrays to device
err=cudaMemcpy(d_a, a, sizeof(int) * N,
cudaMemcpyHostToDevice);
err=cudaMemcpy(d_b, b, sizeof(int) * N,
cudaMemcpyHostToDevice);

// launchkernelover the <N, 1> grid
matSum<<<N,1>>>(d_a, d_b, d_c);
```

Figure 58: This example shows how much CUDA runtime API is easier than the driver API

9. 3. CUDA compiler

CUDA compiler processes both host and device code, then it splits them.

CUDA compilers transforms device code into **PTX**, which is CUDA's ISA.

9. 4. CUDA memory model

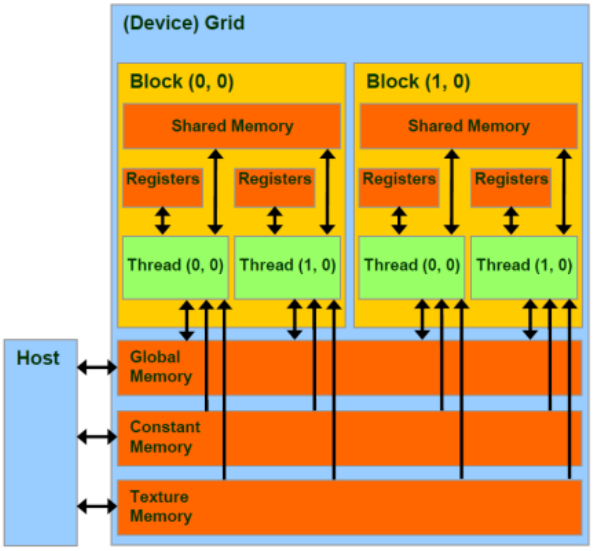


Figure 59: CUDA memory scheme

9. 4. 1. Global memory

Largest R/W memory available on the device shared among all blocks. It's a high-latency high-bandwidth memory.

`cudaMalloc` allocates memory in this region.

Global memory is **column-major**. Threads should access it using an **offset** instead of a **stride**.

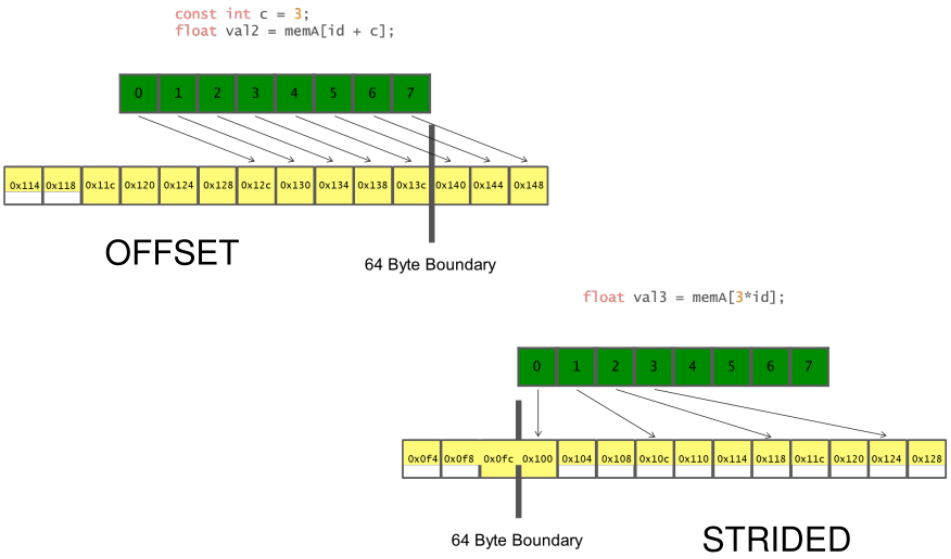


Figure 60: Offset vs strided memory access

Between global memory and SMs there are 2 cache levels:
- L1, private to the single SM;
- L2, shared among all SMs

Load/store operations in global memory are handled by the load/store unit. Every load/store operation is started from all threads of a warp at the same time.

9. 4. 2. Shared memory

Fast memory within the SM. "Shared" is misleading here, because this memory is shared only between threads of the **same block**.

Data in this memory is not preserved between different kernel executions.

Shared memory is typcally used as a **cache** for the global memory and to communicate with other threads in the block.

Each thread can access the whole shared memory address space, so writes on it must be **synchronized** between all threads in the same block.

Shared memory is:
- **multicast**: if multiple threads of the same warp access the same element, the access is done in a single transaction;
- **broadcast**: if **all** threads of a warp access the same element, the access is done in a single transaction

Shared memory is organized in **banks**. Access to the same bank is serial: if multiple threads access **different** address of the same bank, a **bank conflict** will occur.

To allocate a variable defined **inside the kernel function** on the shared memory, its definition must be preceded by `__shared__`:

```
__global__ kernel(...) {
  __shared__ float arr[SIZE];
}
```

Listing 26: `arr` is allocated to device shared memory

Shared memory can be used also **dynamically** when its size cannot be determined at compile time. The size (in bytes) of the shared memory for each block must be specified with the optional third parameter when invoking the kernel:

```
kernel<<<blocks, threadsPerBlock, sharedMemorySize>>>(...):
```

9. 4. 3. Thread synchronization

Threads **in the same block** can be synchronized using `__syncthreads()` API.

This API should be used with care to avoid deadlocks:

```
if (threadIdx.x < 16) {
   ...
   __syncthreads();
} else {
   ...
}
```

Listing 27: `__syncthreads()` here will produce a deadlock for the first 16 threads in the warp. They will wait forever because the other 16 threads will never enter the `if`, and thus will never reach the `__syncthreads()`.

Synchronization between **different** kernels is automatically handled by the system: before invoking the second kernel, the system waits until all writes of the first one are completed.

CUDA provides also **atomic operations** that must always be used when writing data into global or shared memory. These APIs are expensive in CUDA, so they must be used with care.

9. 4. 4. Constant memory

Read-only memory that holds data not known at compile time.

This memory is initialized by the host and then it's used by CUDA threads.

Actually, constant memory it's just a portion of the global memory, but its access uses a dedicated cache.

This memory must be used when all threads of the same warp access the same data. When a warp reads a vale from the constant memory, the value will be put in the constant cache by the system (if not already present), so access to the same data by other warps will be much more efficient.

9. 4. 5. Registers

Registers hold thread-local data.

Registers are not unlimited, so their usage should be keep low. The higher the register pressure, the lower is the achievable parallelization with CUDA, because the lower is the number of blocks that can execute on a SM.

CUDA compiler optimizes register usage.

9. 4. 6. CUDA local memory

Not a physical memory, but just a **view** of its private memory for a thread.

Tries to use register, then it fallbacks on the global memory when there is no space left.

9. 5. Data transfers in CUDA

Communication between host and device is complicated because the memory is accessed differently between the two systems:

- modern CPUs use **virtual** memory addressing;
- GPUs use **phyiscal** memory addressing

Virtual memory on the host is managed both by the CPU and the OS. When started, each process gets an amounts of virtual memory which typically is much more than the physical memory available. Translation of virtual addresses in physical addresses is done by the CPU and the OS.

Virtual memory can be mapped also to the secondary storage (e.g. the disk) if there is not enough space in the DRAM.

9. 5. 1. Pageable memory transfer

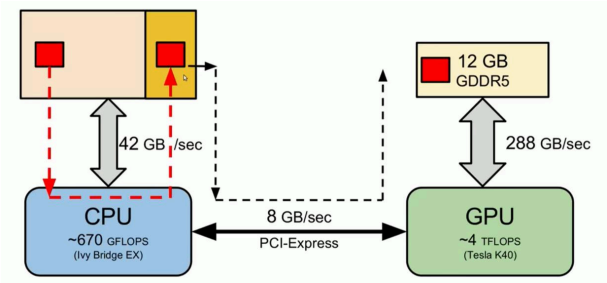


Figure 61: Pageable memory transfer

This type of transfer is subject to high overhead, because:

- it may be swapped to disk;
- requires multiple copies, one for each page

A plain `malloc` will allocate memory in this area.

```
w0 = (float*)malloc(SIZE);
cudaMalloc(&w0_dev, SIZE);
cudaMemcpy(w0_dev, w0, SIZE, cudaMemcpyHostToDevice);
kernel<<<...>>>(w0_dev, SIZE);
cudaMemcpy(w0, w0_dev, SIZE, cudaMemcpyDeviceToHost);
```

Listing 28: Pageable memory transfer example

9. 5. 2. Pinned (page-locked) memory transfer

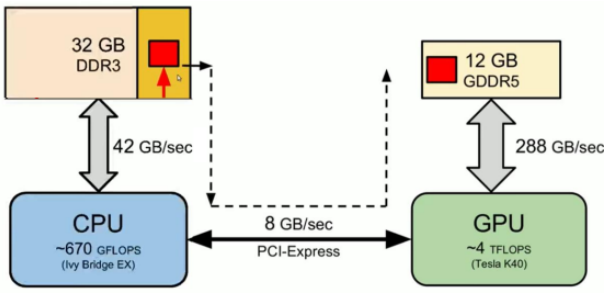


Figure 62: Pinned memory transfer

This transfer performs better than the pageable one, but it reduces the amount of physical memory available for the OS.

```
cudaMallocHost(&w0, SIZE);
cudaMalloc(&w0_dev, SIZE);
cudaMemcpy(w0_dev, w0, SIZE, cudaMemcpyHostToDevice);
kernel<<<...>>>(w0_dev, SIZE);
cudaMemcpy(w0, w0_dev, SIZE, cudaMemcpyDeviceToHost);
```

Listing 29: Pinned memory transfer example

9. 5. 3. Unified Virtual Memory (UVM)

UVM combines host and device memory in a single **logical** view.

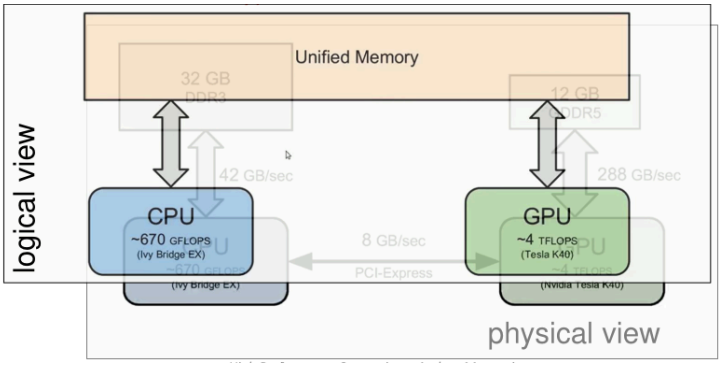


Figure 63: UVM is just a simplified logical view of the more complex physical memory system

```
cudaMallocManaged(&w0, SIZE);
kernel<<<...>>>(w0, SIZE);
free(w0);
```

Listing 30: UVM usage example

While it's not visible in the code, a data transfer still takes place with UVM. Data transfers with UVM happen through multiple small transfers, so while it's easier to write code using UVM, it has a quite big overhead.

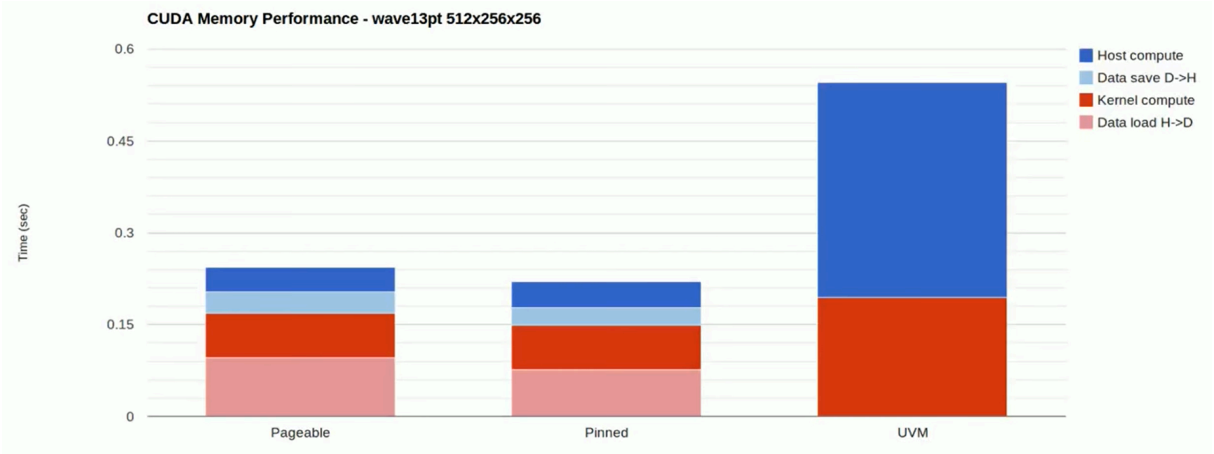


Figure 64: Overhead of different data transfer types

In general, UVM is useful only if:

- GPU is integrated with the host;
- data is loaded on the GPU **only once** and kernel **operation intensity** is high

9. 5. 4. Asynchronous data transfers

CUDA provides also asynchronous data transfer APIs (e.g. `cudaMemcpyAsync()`).

Asynchronous data transfers requires pinned memory or **streams**.

9. 5. 4. 1. CUDA streams

A CUDA stream is a FIFO queue of GPU operations (mainly kernel launches and memory copies) that must be executed in a specific order (e.g. insert order in the queue).

CUDA defines a **default stream** (stream 0) which is used transparently to the user when no other streams are defined.

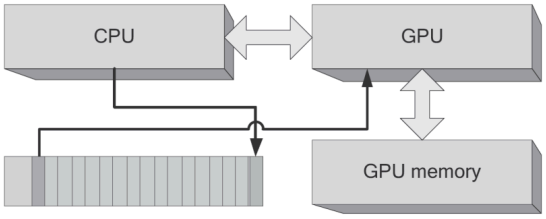


Figure 65: Default stream usage

Commands in the same stream are executed sequentially, but commands in different streams are executed concurrently. Streams can therefore be beneficial to optimize application performance.

Streams can be synchronized both explicitly, using CUDA APIs, or implicitly. Streams are always synchronized when using blocking memory allocation APIs (e.g. `cudaMalloc`, `cudaMallocHost`).

CUDA streams help in overlapping computation and memory transfers **within the device**.

Double buffering is a technique that exploits multiple streams execution to improve concurrency. It's very similar to CPU pipeline:

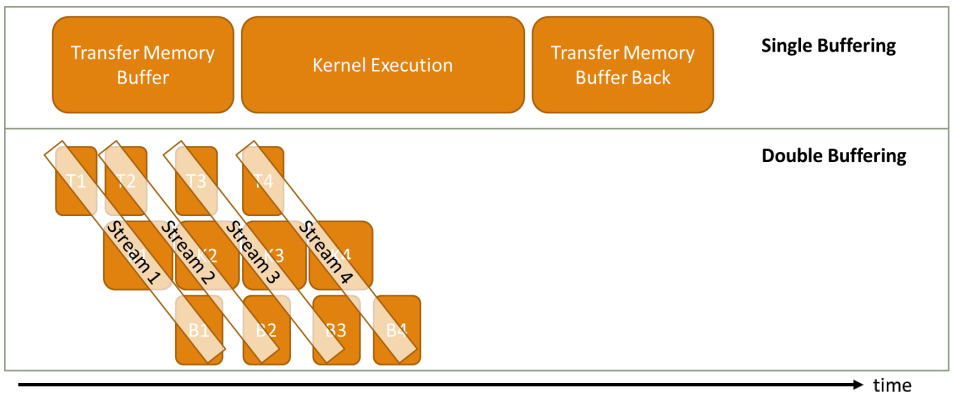


Figure 66: GPU double buffering example

10. FPGA (Field-programmable gate arrays)

FPGA is a computer architecture that combines the flexibility of software with the high performance of hardware. It's an intermediate approach between full-custom hardware (ASIC) and something that is **programmable**.

FPGA is custom hardware, but the way it's implemented does not rely on the physical representation of transistors, but just on **reconfiguring** some logic.

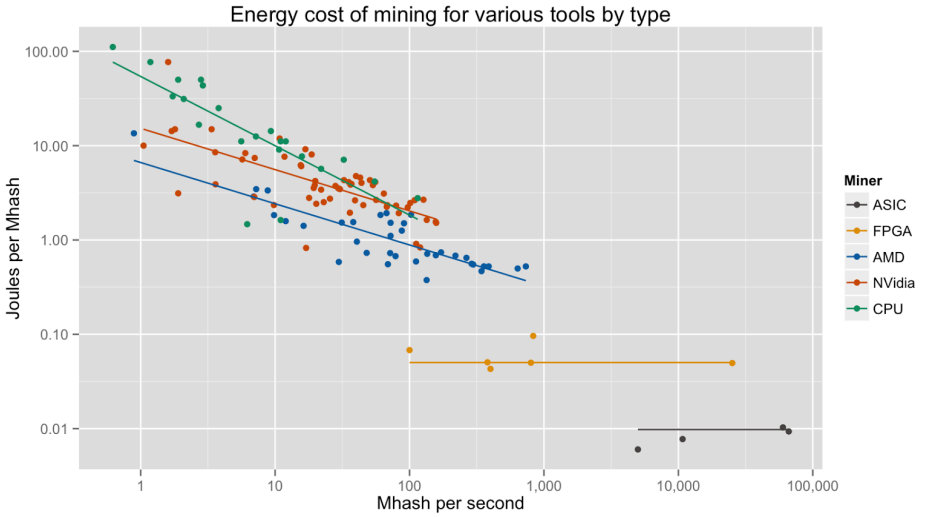


Figure 67: Performance and energy efficiency of diffferent architectures.

CPU and GPU are bound to the Von Neumann architecture: every program is executed in the same exact way. This is required to be able to execute **any** program, but it's not the optimal solution to extract performance from the system.

ASIC and FPGA, since they can implement completely custom logic, are the most efficient architectures.

An FPGA is a matrix of **configurable logic blocks** (CLB):

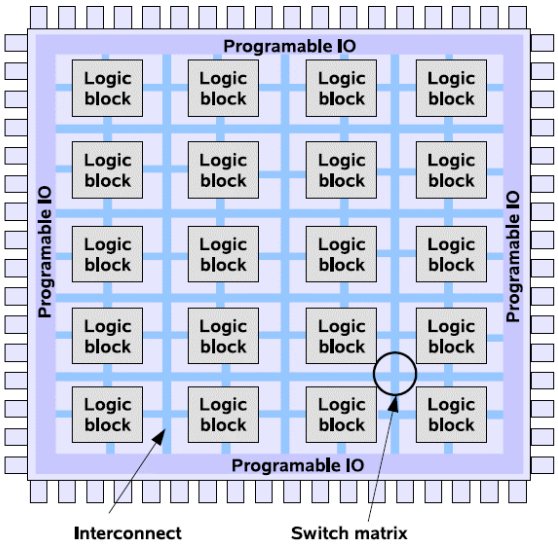


Figure 68: CLB matrix

Each CLB is made of two components:

- **slices**, made of **lookup tables**. These perform logic operations;
- **flip flops**, used as a memory storage

A lookup table is basically a multiplexer that evaluates a **truth table** stored in the configuration.

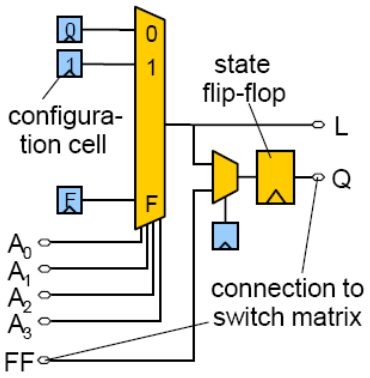


Figure 69: Lookup tables and flip flops

Wires are used to interconnect multiple CLBs together. **Connection boxes** and **switch matrices** are the fundamental components of these interconnections.

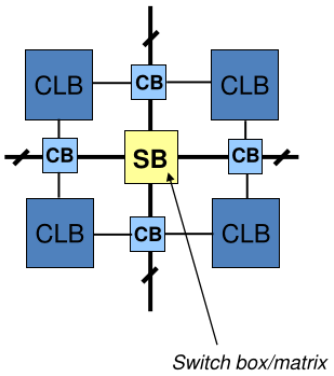


Figure 70: Routing in FPGA

By configuring connection boxes and switch matrices, it's possible to allow (or forbid) a certain direction of communication.

Switch boxes are actually **bitmasks** that open/close a door.

FPGAs have access to the system's DRAM, but they also have two types of internal memory:

- **block RAM**: SRAM memory connected to the circuit using switching boxes and switching matrices;
- **distribute RAM**: interconnection of several flip-flop within the same CLB

FPGA also have components to perform **I/O operations**.

FPGAs can either be **integrated** (on the same chip of the CPU) or **discrete**.

10. 1. Place & route

Place is the process of deciding which parts of a logic circuit are mapped to which CLBs in the FPGA.

Route is the process of defining the interconnection of CLBs to implement the logic circuit functionality.

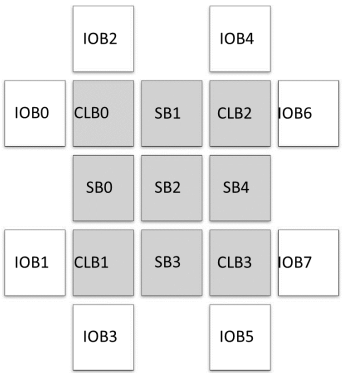
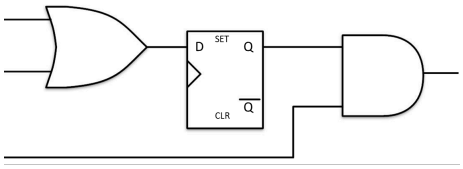
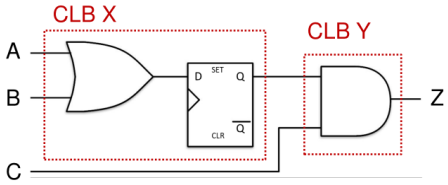


Figure 71: Simpled 2×2 CLBs FPGA

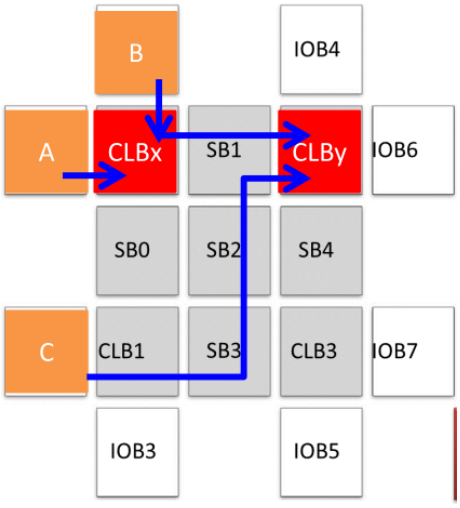
If this is the circuit we want to implement in our FPGA:



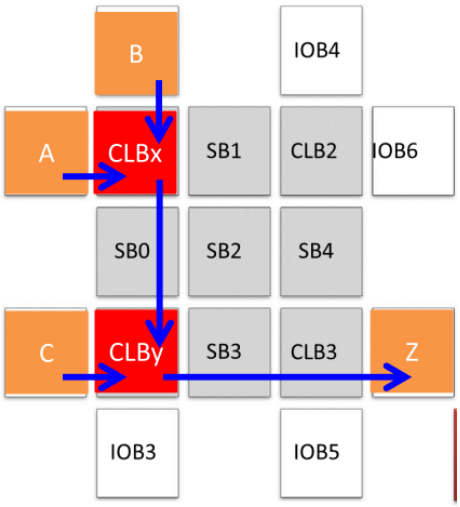
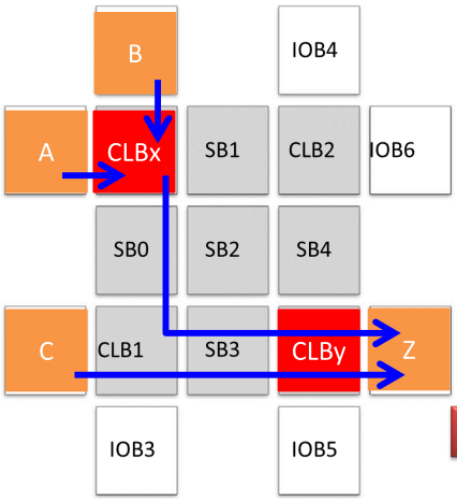
we can identify 2 separate parts. Each one of them will be mapped in a dedicated CLB:



The next step is to place the CLBs in the FPGAs. We have just 2 CLB to place in a 2×2 grid, so there are different alternatives:



This placement is not optimal because routing is complicated (3 switching boxes used).

	This is the optimal solution because it uses just 2 SBs and 2 wires.
	This is very similar to the previous one, but it uses one more wire, so it's not optimal.

10. 2. FPGA configuration

FPGA reconfiguration is about changing switching blocks configuration through a **bitstream**. Reconfiguring the FPGA is the process of loading this bitstream on it.

During the development phase, the FPGA device is programmed using utilities such as the Vivado IDE. On production hardware, the bitstream is placed in non-volatile memory and the hardware is configured to program the FPGA when powered on.

Reconfiguring FPGA takes time, so it's crucial to find a balance between **complete** reconfiguration and **partial** reconfiguration.

FPGA reconfiguration can be **static** (must be done **before** execution, it cannot be done on a working environment) or **dynamic** (can be done also during execution).

Reconfiguration can be **external** (the bitstream comes from an FPGA external source) or **internal**.

11. HLS - High Level Synthesis

HLS is a (very) abstract manner of describing how the hardware of an FPGA should behave.

FPGAs are configured using an hardware design flow:

1. desired hardware behaviour is described using an **Hardware Description Language** (HDL);
2. HDL is turned into a configuration bitstream using tools like Vivado

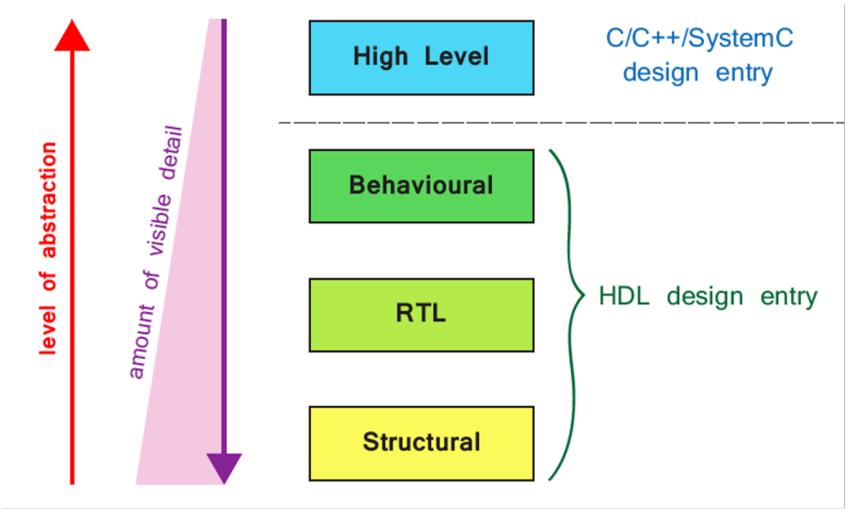
HDL is very similar to a programming language:

```
module exercise(
        start, clk, ready, rst
    );

    input  wire start, clk, rst;
    output wire ready;

    reg  [4:0] reg_val;
    wire [4:0] reg_in;
    wire and_reg, sel_mux;

    always @(posedge clk) begin
        if (rst) reg_val <= 5'd0;
        else if (and_reg) reg_val <= reg_in;
    end

    assign reg_in  = (sel_mux)      ? (reg_val - 5'd1) : 5'd8;
    assign sel_mux = (reg_val != 0) ? 1'b1             : 1'b0;
    assign ready   = (reg_val == 0) ? 1'b1             : 1'b0;

    assign and_reg = sel_mux | start;

endmodule
```

Figure 72: HDL example

Synthesis is the process from which a **gate-level netlist** is generated from hardware description.

Designing hardware requires an understanding of how logic gates work. Not all FPGA users have those skills, so its design must be simplified in some way.



Source: The Zynq Book

Figure 73: Different levels of abstraction when designing an FPGA

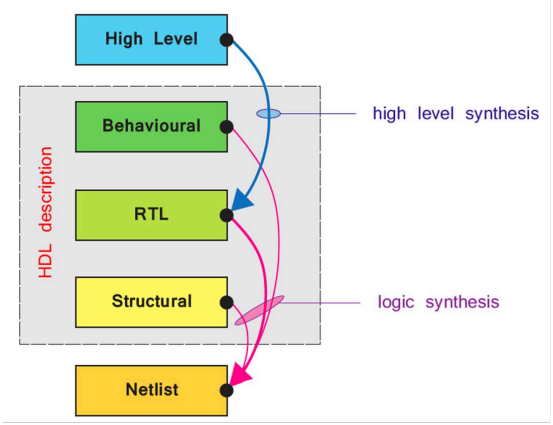
FPGA must be designed in two directions:

- **behaviour**: the algorithm which runs on the FPGA;
- **structure**: registers, gates, flip flops, etc.

High level synthesis (HLS) is an automated design process that transforms a high-level function specification to a register-trasnfer level description suitable for hardware implementation.

HLS allows to describe the hardware using a high-level language such as C/C++.

HLS covers the first layer of hardware synthesis:



Source: The Zynq Book

Figure 74: HLS vs HDL

Control logic and datapath can be extracted directly from the C/C++ code.

Scheduling and **binding** are the core of HLS:

- scheduling determines in which clock cycle an operation will occur;
- binding determines which **core** will execute an operation

Each C/C++ function is translated into a dedicated RTL block. Function arguments become ports on these RTL blocks.
Functions may also be **inlined** to dissolve their hierarchy.

HLS largely uses the **loop unrolling** and **loop pipelining** optimizations to expose higher parallelism and reduce the latency.

HLS describes hardware as a **finite-state machine** that performs different atomic operations.

Producing the final bitstream requires time, therefore developers rely often on **simulators**. Memory performance of these simulator is not realistic, but still simulation is useful when developers care only about executing a function in the FPGA and do not care about its integration in the system.

The **initialization interval** (II) is the number of clock cycles to wait before the computing of the next iteration of a loop can start. It depends on the availability of resources (e.g. number of ports to access the memory).

Labs

12. Profiling

Profiling is not only useful for measuring performances, but also for **debugging** and **optimizations**.

Profiling is used in many context (e.g. web applications), not only in hardware.

Steps to create a parallel program:
1. identify parts of the program (**hotspots**) that can be parallelized;
2. parallelize the code on multiple threads;
3. **assess** that parallelization had a benefit on performance (and also that the new parallelized version still produces correct results)

Profiling can be useful in all these steps, because:
- it helps in identifying hotspots;
- in can be used to check the overhead of parallelization, the cost of memory access, etc.;
- it can be used during the assessment to check if parallelization had a benefit on performance

Different metrics for different aspects of the system:
- **throughput**: operations per seconds, measured in FLOP/s (floating point operations);
- **memory bandwidth**: measured in GB per seconds;
- **energy efficiency**: this metric has become very important in the last years, not only for **environmental** aspects but also for **efficiency** (doing more with less energy waste)

The **Roofline model** is a metodology to describe performance of a system which can display also the maximum achievable performance:

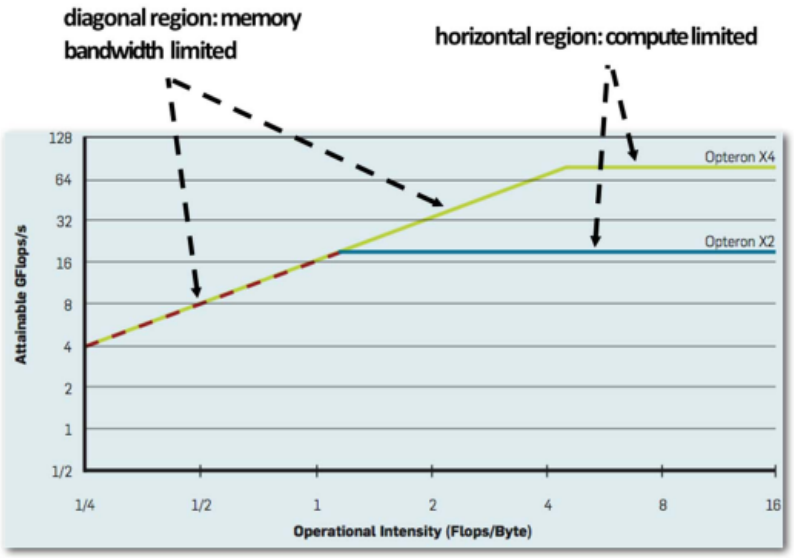


Figure 75: Roofline model

The x axis measures the **operational intensity**, which is the computational cost of the operation relative to the amount of data it needs. The y axis measures the throughput of the system.

Example of operational intensity for `c = a + b`:
1. read `a`;
2. read `b`;
3. calculate `a + b`

3 instructions in total with 2 data accesses, so the operational intensity is $\frac{1}{3}$.

The roofline model helps in identifying **memory bottlenecks** because it can show if a program is memory-bounded.

The graph provided by the roofline model is very important. **Before** going through optimizing the code, the programmer must know what is the maximum achievable performance in order to not waste time in optimization.

Profiling is a **dynamic** program analysis that measures different aspects:
- memory usage;
- time complexity;
- instruction usage;
- frequency and duration of **function calls**

Profiling is **dynamic** because different operations takes different amount of time, so the profiler cannot simply see the (static) source code of the program.

There are different techniques to collect profiling data:
- **performance counters**: typically implemented at **hardware level** by CPUs;
- code instrumentation;
- hardware interrupts: execution is interrupted every t seconds to check for a metric;
- OS hooks (scheduling point, context switch);
- ISA simulators: can be used when there is no bare-metal hardware

12. 1. Profiling tools

perf	• uses hardware and OS counters; • very low-level and fine-grained statistics; • highly dependent on the architecture
gprof	• measures **function** execution time; • requires special compilers annotations; • does not actually measure the time spent in a fuction; this time is just **inferred** by taking "snapshots" of the application at regular intervals; • not suited for applications with very small functions because their execution time might be lower than the sampling interval
Valgrind	• collection of tools, rather than a single one; • profiled code is executed in a **sandbox**. This is an advantage because there is no need of hardware counters or compiler annotations, but it also mean that performance when run on **real** hardware will be slightly different

Table 13: Comparison of common profiling tools

13. HLS

13. 1. Interfaces

AXI (or **mAXI**)	full-access interface
AXI-lite	lightweight interface, used for scalar and constant propagation
AXI-stream	very efficient, non-memory mapped interface

Table 14: AXI4 interfaces

Bundling is how interfaces are mapped to physical memory ports. The number of memory ports is limited and multiple accesses using the same port happen in sequence (not concurrently), so they should be used with care.

AXI master is a **bidirectional channel**: it's able to read and write concurrently.